Whitepaper

What is Holographic Processing?

Inside the Connective Logic Engine

www.connectivelogic.co.uk

© Copyright 2009 Connective Logic Systems Ltd.

Copyright © Connective Logic Systems Ltd 2009

Contents

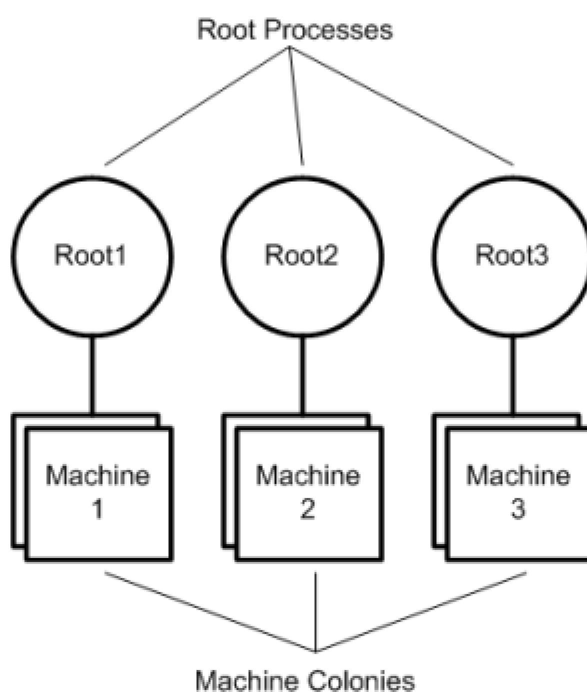
1 Introduction 1
2 Background 3
3 Explanation of Terms 4
4 The Problems Addressed 5
4.1 Inter-thread Communication 5
4.2 Coordination and Synchronization 7
4.3 Threading 11
4.4 Blocked Workers 12
4.5 Event Propagation 13
4.6 Flow Control 18
4.7 Accretion 21
4.8 Scheduling and Colonization 22
4.9 Memory Management 24
4.10 Dynamic Recruitment/Retirement 27
4.11 GUI Interfacing 27
4.12 Portability of the Runtime 28
5 Benefits 30
5.1 Portability 30
5.2 Performance Issues 30
5.3 Holographic Processing Benefits 30
5.4 Object Oriented Paradigm 31
6 Case Study 32

Copyright © Connective Logic Systems Ltd 2009

1 Introduction

The term ‘holographic processing’ refers to a Multiple Instruction Multiple Data (MIMD) technique that is primarily aimed at multi-core processing architectures. The name derives from the fact that most of the application functionality can be executed by most of the threads, and much of the data is optimally distributed (only copied when necessary). This means that individual machines can be recruited and/or retired at any stage of execution and in particular the application can be executed as a single thread of execution (although the real-time case may require additional threads in order to execute preemptively). Holograms can be broken into pieces, but all the information is preserved in each piece, and one of the main goals of the technology described in this note is to provide a similar resilience for distributed software applications (scale on demand).

This means that the application need not know (or care) how many CPUs (or even machines) are available at any one time, indeed the number could change at any stage of execution. As more CPUs are recruited the application can run faster, or in some cases, the fidelity of the calculation can increase (application design decision), if they are retired then the opposite applies. From a technical perspective, the concept of single machine SMP thread pools exploited by OpenMP and similar technologies is generalized to heterogeneous networks of multi-core processors but in a way that hides all of this from the application. The application sees the whole network as a single virtual process (SVP) with data accessed by reference, and tasks executing preemptively. In the general case, an application may contain multiple SVPs (each root process is distributed across an arbitrary number of machines).



This technique therefore requires a different programming approach as well as specific runtime capability (synchronization, scheduling etc – see Connective Logic). In particular, engineers need to think in parallel terms rather than sequentially. There are a number of issues that arise as a result of concurrent execution and these include; avoiding cases where two components update the same data simultaneously, making sure that components are not executed until their inputs are ready, launching one or more components when their shared inputs do become available, and last but not least providing an equivalent of automatic stack data so that data objects are relinquished when they are no longer referenced (happens automatically when a sequential program returns from a function that instantiates objects on its stack). But in this case, data objects are cached across the network and so when no longer referenced, the definitive and all cached instances need to be ‘freed’.

Copyright © Connective Logic Systems Ltd 2009

Equally importantly the runtime that provides these services must not use too many machine cycles otherwise scheduling overhead could become a serious drain on resources (limiting potential granularity). Also, for the general case of real-time execution, object deletion (garbage collection) has to be deterministic. If the solution is going to work in the general case of real-time systems, then preemption also needs to work at network scope rather than just individual machine scope.

Crucially, holographic systems (as defined above) make extensive re-use of the logic that their sequential equivalents provide and so migration from an existing application to a holographic equivalent does not involve extensive porting from one language to another (most code remains unchanged) and typically, the exploitation of multi-core is restricted to very course grain high-level modification, and as a consequence, very few engineers need to consider anything other than business as usual. As an aside, the approach suggests that multi-core only needs to create a new specialization rather than requiring everybody to re-learn their trade. This means that domain experts can develop parallel applications without needing to reason about threads, locks, semaphores, or any other low level synchronization issues.

However, the holographic execution of an entire application is not appropriate in all cases and so there also needs to be a means of mapping application functionality to one or more sub-systems and then optionally executing each of these holographically. The important issue is that it is necessary to distribute the functionality arbitrarily without needing to re-write any of the application's code; otherwise the application would be tied to a given platform topology and would not be portable. Also, it is often extremely convenient to be able to execute the whole system on a single CPU (as a single process) for development and maintenance purposes (greatly simplifies the logistics of debugging etc). The process of mapping the whole application to one or more executable processes is referred to as accretion, and typically most projects will have many accretions; some for debugging in slow time, some for debugging in real time, and of course the definitive release version. The goal is that the application must not change, only the build (see Accretion and Colonization).

In order to implement this higher level abstraction, it is necessary to consider exactly how application functionality is going to be scheduled by two or more processor cores (concurrent execution) and how this can be done without knowing (or caring) about the underlying processor architecture or (for distributed systems) topology.

It is also worth noting that this article aims to provide a high level overview of how holographic processing can be implemented, and is not necessary reading for engineers who simply want to use CLIP. However, as with most software technologies, an understanding of the underlying implementation is often useful when considering optimal use of the available functionality.

The connective logic technical article introduces a visual symbolism (CDL) for describing the coordination of event flow and in many of the examples that follow this is used in order to clarify the points being made. It is therefore recommended that the connective logic article should be read before this one.

Copyright © Connective Logic Systems Ltd 2009

2 Background

The Connective Logic Infrastructure Programming (CLIP) technology has been in development since 1993 and the current implementation began in 1995. The Concurrent Object Runtime Environment CORE (part of the CLIP technology) is an implementation of a kernel with a 'holographic' capability as described above, and has been in continuous commercial development and use since that time.

In earlier times it was used as a middleware, but more recently, with the development of the Concurrent Description Language (CDL) translator, it is even more effectively treated as a Visual Programming Language (VPL) runtime. CLIP was initially developed to address general concurrency (multiple threads and/or multiple processes) but in recent times its principal use has been with multi-core platforms (or networks of multi-core machines).

During this time the holographic processing model has been used for a wide range of projects that have ranged from CLIP's own development toolset, up to enterprise scale projects in areas like acoustics, seismology, simulation and surveillance. In recent times it has been used for a number of mission critical systems including the Royal Navy's Surface Ship Torpedo Defense system (SSTD), Unmanned Air Vehicles (WatchKeeper) and Synthetic Aperture Sonar (Artemus).

CORE is mature when compared to more recently developed solutions that address the problems raised by multi-core. Typical application sectors include military, aerospace, oil and gas, medical, surveillance and financial; but it has far more general application than this. Its successful use for distributed interactive military simulations suggests that it could be applicable to video games (especially multi-player implementations) and its inherent scalability makes it a candidate for general High Performance Computing (already demonstrated for finite difference time domain applications). See Case Studies.

Copyright © Connective Logic Systems Ltd 2009

3 Explanation of Terms

The following terms are used throughout this note.

Connective logic refers to a programming paradigm which is expanded in the accompanying document to this one. For full appreciation of this note, we recommend reading the connective logic document first.

In most contexts the term **blocking** refers to the case where a CLIP Method (see below) is unable to execute because either its inputs are not yet available, or it cannot yet get space for its output. In practice, the runtime worker threads will only block in the true sense if the total concurrency of the executing process falls below the number of worker threads (or logic specifically dictates that they should).

By default CORE creates one worker thread for each CPU core, and does so for each priority that methods are declared to execute at (so for non real-time applications this will usually be one). When their inputs and outputs are available, they are scheduled for execution by system owned worker threads, and in the general case these may actually be running on another machine. Passing a job for execution to another machine only involves sending its inputs (if the target doesn't already have them), and a 32bit identifier telling it which method to pass the arguments to (all processes are linked with all executable tasks so no code needs to be sent).

CLIP Methods are a lightweight alternative to conventional operating system threads and in practice almost all concurrent execution is achieved through these objects. Conventional threads are supported but they are only generally required for certain types of I/O where control can block outside of CLIP in the true sense. Although methods are almost entirely equivalent to threads, their 'blocking' does not usually result in workers 'blocking' in the conventional sense, and so their use can greatly reduce context switching overhead and considerably reduce total application stack requirements; this allows for extremely fine execution granularity. The fact that they are actually executed by workers is completely transparent to the application developer who will usually see them as a straightforward fully preemptive replacement for threads. They are in no way related to 'fibers'

Scheduling latency is an issue that affects many parallel applications. It refers to the situation where a component is logically schedulable (all its inputs and outputs are available) but because of sub-optimal implementation it cannot be scheduled. There are a number of examples of this in the connective logic document. It is not the same thing as Amdahl's effect (which results from a limitation of the algorithm being parallelized) but it has the same effect and prevents applications from scaling to their best theoretical limit. It can easily be mistaken for Amdahl's but (by definition) can be remedied by redesign of the application.

Leaf Providers are objects that provide events, but do not consume them. An example would be a transient store which provides two types of event; a 'ready for write', and a 'ready for read'.

Root Consumers are objects that consume events but do not provide them. An example would be a thread, a method or a GUI call-back (see below).

Developing CLIP applications is a multi-stage process. The first stage is to create an application which makes no assumption about topology (assumes a single virtual process). The second stage is to statically map that functionality to one or more executable 'processes'. This is referred to as **Accretion** (see below) and is totally separate from application development. In practice most projects will have a number of different accretions; a single process accretion for debugging/maintenance, and various other accretions for various targets. Finally, each of these 'accretions' can be distributed to **Colonies** of dynamically scheduled 'slave' processes.

Copyright © Connective Logic Systems Ltd 2009

4 The Problems Addressed

The reason for considering an alternative approach to concurrent systems stems from a number of problems that make the development of parallel programs very difficult in general, and given that processor manufacturers are universally moving to multi-core, this means that these issues now have a much higher profile.

It is generally agreed that threads, locks, and semaphores are equivalent to assembly language components in terms of their level of abstraction and although some specific problem areas have workable solutions (e.g. OpenMP and MPI for data parallel applications), the general case of irregular concurrency is arguably unresolved, and in particular, code that uses conventional techniques (especially locks) doesn't compose well and so it is very difficult to build large systems and/or re-use components.

This section considers what are believed to be some of the fundamental problems with conventional parallel programming approaches and explains how these are addressed by the CLIP technology.

4.1 Inter-thread Communication

This is one of the most fundamental issues in any concurrent system, and the reason it is an issue is that if two communicating threads are in the same address space then communication by reference is the simplest and most efficient means of exchanging data, but if they are in disparate address spaces (e.g. different machines) then this cannot work and so data has to be moved using a more complicated message-passing type scheme. And the problem that this then presents is portability. Unless we go for a lowest common denominator approach and make all inter-thread communication move data (very inefficient use of multi-core technology) then our application could be locked into a particular topology (e.g. 8 machines with 4 cores each).

Over the coming years core-counts are predicted to double every 18 months or so, from the current norm of 2 or 4, up to literally hundreds; and this means that cluster topologies will be likely to change as fewer and fewer machines are actually required (unless problem sizes exactly scale with core-count which is very unlikely). Once core-counts pass 32, many experts predict that the symmetric shared memory (SMP) model will have to change and this will only exacerbate the problem.

Applications that assume a particular topology will probably need continual maintenance. And addressing this is not as simple as it seems. If an application needed to run within a shared memory architecture it would probably want to use something like OpenMP and communicate by reference, but if it wants to schedule across a network it is more likely to want to use a message passing paradigm such as that provided by MPI. It is also possible that platforms like the IBM Cell may be the future and unless this issue is addressed, communication could become even more topology specific.

In order to write truly portable programs (rather than just O/S independent ones) we therefore need to abstract the platform topology in such a way that if two threads find themselves in the same address space at runtime, they will use reference, but if they find themselves in different memory spaces the data can be transparently moved without the application needing to do anything specific. This is a fundamental feature of holographic processing and the same code will run on any number of disparate SMP sub-systems. This is how the same application can be 'accreted' to any number of different topologies without the need to change the application itself.

The first thing that is required is an Application Programmer's Interface (API) that will work in both cases (shared and disparate memory), and again this is not as easy as it seems. The obvious question is why is it not possible to just use something like a Berkley socket interface, and in the case that the runtime detects that the recipient is in the same machine, just pass a reference rather than the data itself. This would seem to be a minor change to the API (the recipient would be given the address, rather than providing it) and all the runtime would need to do would be to pass the reference in the shared memory case; but move the data, create a buffer and return its address in the disparate case.

This almost works but doesn't address another very important issue; data life-cycles. If the software passes a reference to an automatic object that is created on the stack, then the executing thread cannot return from the providing function until the recipient has finished with the data and the simple API above doesn't have a way of letting the provider know when the consumer has finished accessing

Copyright © Connective Logic Systems Ltd 2009

the data. Even if the API is now modified so that the consumer is obliged to inform the provider that they have finished with the data, there is still the problem that the provider is gratuitously blocked, waiting for a response, when it could probably have continued to do some more useful work.

If a persistent storage scheme is used, rather than stack, then it solves the problem of not being able to return but the provider still needs to know that the consumer has finished with the data otherwise it could over-write an earlier message, and if it has returned then there needs to be some way of dealing with this that isn't synchronous (the provider should never need to wait for the consumer).

The simplest solution to this problem is to use a producer/consumer model and indeed this technique is commonly adopted by multi-threaded systems. So the communication code ends up looking something like;

Writer Code

```
buff = waitOpenWrite( store_id );  // Wait for writeable buffer ref
populate( buff );                  // Populate buffer ref
close( store_id);                  // Close and unblock reader
```

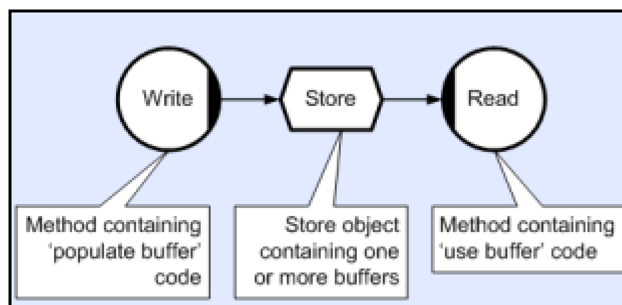
Reader Code

```
buff = waitOpenRead( store_id );   // Wait for readable buffer ref
useBuff( buff );                   // Use the input
close( store_id );                 // Close and unblock writer
```

Shared buffers can then be static (created once at runtime) or dynamic (providers allocate and consumers delete). CLIP provides this functionality through an object that is referred to as a transient store. The number of buffers (store depth) is configurable and writers are blocked if the store is full, and readers are blocked if the store is empty.

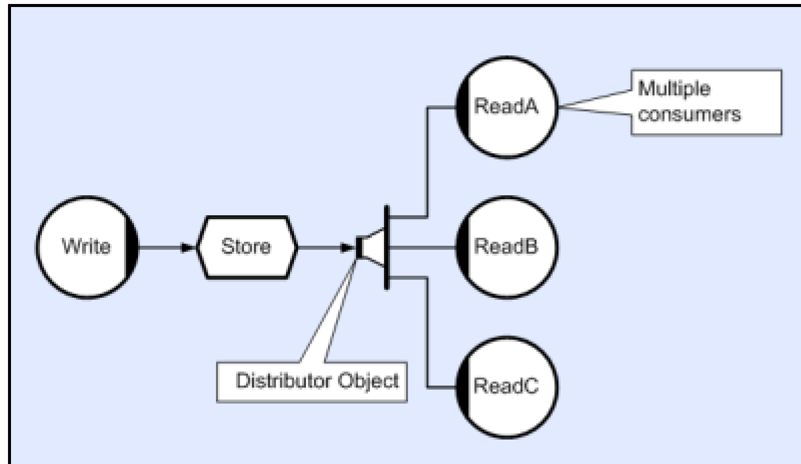
This 'blocking' flow control is transparently implemented between machines and so a thread reading from a store on one machine can unblock a thread waiting to write on another machine (and vice versa). Static, dynamic and other allocation schemes are also configurable. The above scheme can use reference passing if the threads are in the same address space, but transparently move the data if they are not.

The CDL equivalent of the above code is shown below. Note that the method code does not need to perform the open/close code because this is now performed by the circuitry (generated by the translator). So in this case, the write method would just consist of the developer supplied 'populate' call, and the read method would just consist of the 'useBuff' call.



But there is another problem to solve, and this is the case that arises when one provider has many consumers, and this introduces the concept of distribution (buffers need to be available until their last consumer has finished with them). This can be solved by a publish/subscribe type mechanism and CLIP provides this through the 'distributor' object.

Copyright © Connective Logic Systems Ltd 2009



In this case, each buffer remains valid until all three methods have used it, and when the last method returns, the buffer becomes re-write-able and allows the 'writing' method to re-execute.

4.2 Coordination and Synchronization

However, the provision of stores and distributors still doesn't solve the general irregular problem. Most functions have more than one input and it would be a very onerous constraint if the calculation of these inputs couldn't be scheduled in parallel. So consider the case of three threads A, B and C calculating three values 'a', 'b' and 'c' and then a fourth thread D, waiting until all three have completed and then calculating a fourth value 'd' (using 'a', 'b' and 'c'). The operation of waiting for multiple inputs is referred to as 'collection'. Each writer therefore executes similar code;

Writer Code (Threads A, B and C)

```
buffA = waitOpenWrite( storeA );  // Open store for write (e.g. A)
populate( buffA );                // Populate respective buffer
close( storeA);                   // Close respective store
```

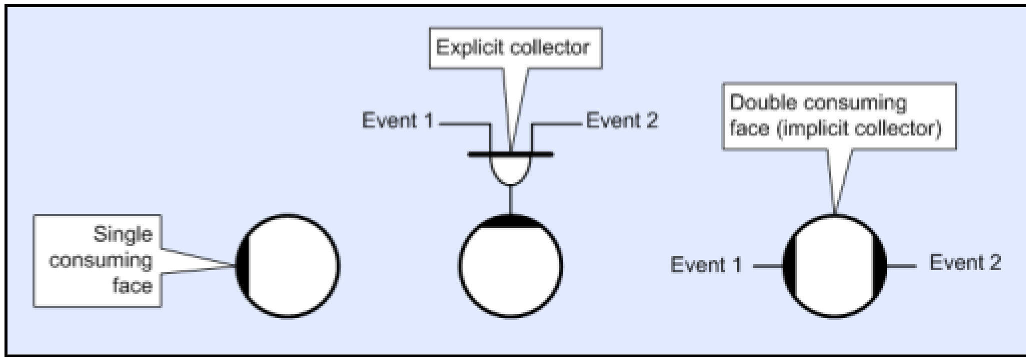
The reader executes the following;

Reader Code (Thread D)

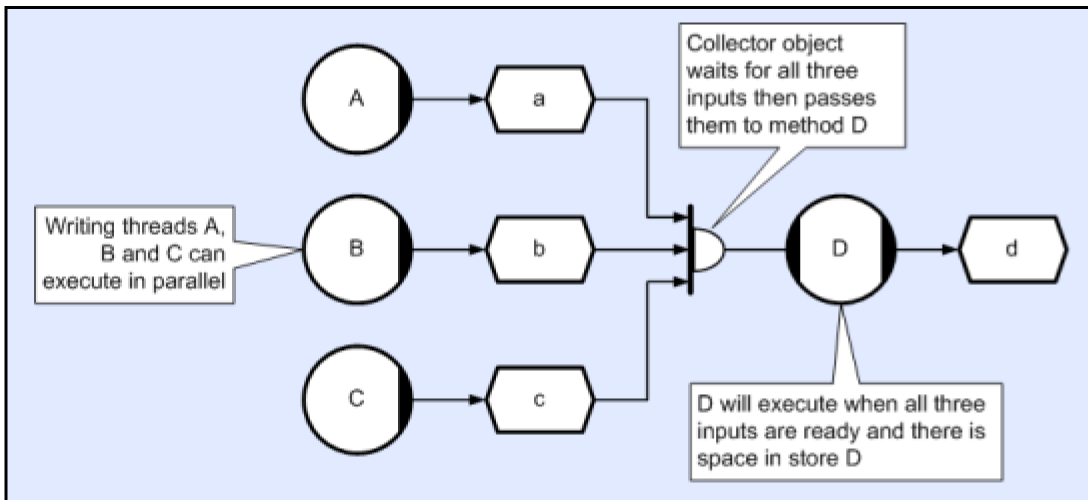
```
buffA = waitOpenRead( storeA );   // Wait for buffer in store A
buffB = waitOpenRead( storeB );   // Wait for buffer in store B
buffC = waitOpenRead( storeC );   // Wait for buffer in store C
buffD = waitOpenWrite( storeD );  // Wait for buffer in store D
PopulateD( buffA, buffB, buffC, buffD );  // Populate output
close( storeA );                  // Close and unblock writer A
close( storeB );                  // Close and unblock writer B
close( storeC );                  // Close and unblock writer C
close( storeD );                  // Close buffer D
```

This example introduces the compound method object which is identified by two consuming connection points. This symbol is equivalent to a method that collects both of its consumed events before executing and is provided as a form of symbolism 'shorthand' (see below).

Copyright © Connective Logic Systems Ltd 2009

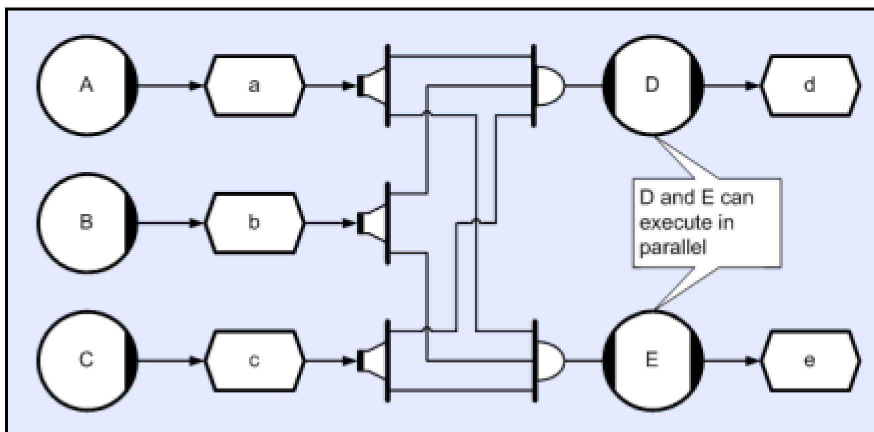


The CDL equivalent therefore becomes;



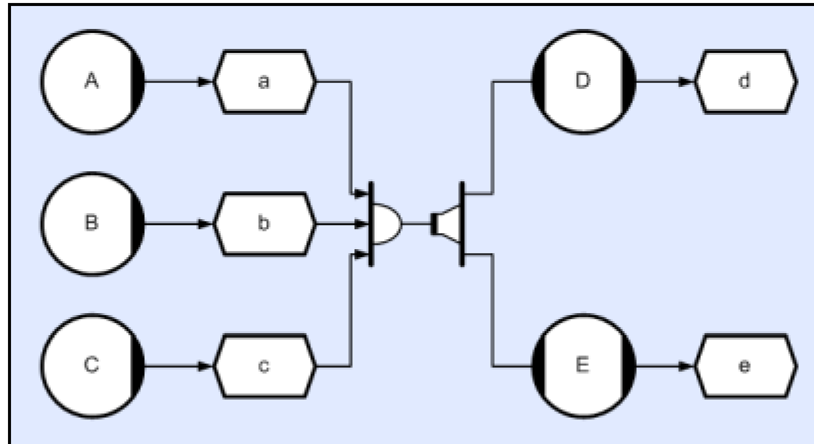
But now consider the case where two threads; D and E, are both required to execute with the outputs from the first three threads A, B, and C. Provided the application has a publish/subscribe style producer/consumer (store and distributor) then the problem can still be solved fairly easily.

Two CDL descriptions are shown below. The first reflects a typical thread solution where D and E would both collect from distributed stores, and the second shows an alternative CDL solution where collection occurs before distribution.



The solution above involves both collectors connecting to all three distributors and the resulting 'crossed-lines' could be confusing. Whilst this could be addressed through the use of conduits and/or labels, the solution below reverses the distribution and collection operations (leading to faster execution) and illustrates the fact that CDL operators are compose-able.

Copyright © Connective Logic Systems Ltd 2009



However, in order to have a completely generic solution, it is necessary to consider the case where a given input could be provided by one of a number of producers, and in particular it needs to work for the case that the actual provider cannot be predicted (equivalent to a UNIX 'select' or Windows 'wait for multiple objects' call). In CLIP terminology this operation is referred to as 'multiplexing'.

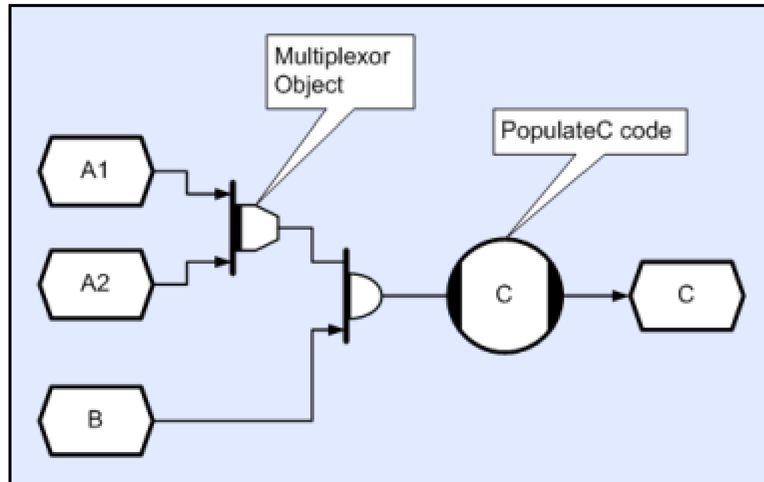
Suppose thread C will execute when its two inputs 'a' and 'b' are ready, but 'a' could be calculated by threads 'A1' or 'A2' (and we don't know or care which).

Pseudo-code for thread C could look something like;

```
// Wait for event from either store A1 or store A2
buffA = waitOpenMultipleRead( storeA1, storeA2 );

// Wait for event from store B
buffA = waitOpenRead( storeB );

// Open C for write
buffC = waitOpenWrite( storeC );

// Now calculate C from A and B
PopulateC( buffA, buffB, buffC );

// Now close whichever store was opened
closeMultiple( storeA1, storeA2 );

// Close remaining stores
close( storeB );
close( storeC );
```

The CDL equivalent would look like;

Copyright © Connective Logic Systems Ltd 2009



Finally, in order to solve the most general problems an exclusion mechanism is also required in order to prevent two or more concurrently executing threads from trying to access mutually shared data in an inconsistent manner (data races). More importantly, there needs to be a means of ensuring that exclusion can be implemented without causing deadly embraces and other deadlocks (this is dealt with in more detail in the connective logic document).

So in summary; the solution requires a number of basic operations and objects. These include;

1. Transient data stores to generalize the concept of 'automatic' (stack) data. These must be logically visible from any machine, and their buffers must be automatically invalidated when no longer referenced (from any machine). This requires a global memory management scheme.
2. The notion of distribution; so that events (especially data updates) can be passed to an arbitrary number of consumers (on any machine).
3. Collection, so that consumers can wait until all of their inputs are ready (from any machine and in any order).
4. Multiplexing, so that consumers can wait for events from any number of potential providers (on any machine and in any order)
5. Exclusion, so that two or more threads are prevented from accidentally updating data in an inconsistent manner (regardless of machine localities)

As will become apparent later, distribution, collection, multiplexing and exclusion need to be compose-able. So it needs to be possible for example to collect distributed collections of multiplexed excluded data events; and so on.

It will also become apparent that the objects that implement these operations must be passive (not require any hidden threads in order to execute). So for example when a request is made to a collector it needs to request all of the events that it requires and then record each one that is available at that time. After this time, each provider that delivers an additional event to the collector needs to check and see if it is the special provider that completes the sequence, and if so, needs to pass all of the collected events (as a single compound event) to whichever consumer is requesting from the collector (or queue it if no consumer has requested yet).

Furthermore it needs to happen asynchronously; so that in the distributed processing case, communications and processing can be over-lapped (otherwise latencies could become problematic). This also needs to happen across machines and has to be transparent to the application.

As an aside, whilst the operations described are complex in conventional terms; from a connective logic perspective all that the application developer actually needs to do is to specify the existence and inter-connectivity of a few fairly simple event operators; and experience to date suggests that engineers find it far easier to specify connectivity visually, rather than through a sequential text API (and they have had the choice). In fact the need for compose-ability etc makes the text API awkward, and most of the code that is now generated from diagrams is actually the creation and connection of objects (difficult to manage without a visual sketch of some sort).

Copyright © Connective Logic Systems Ltd 2009

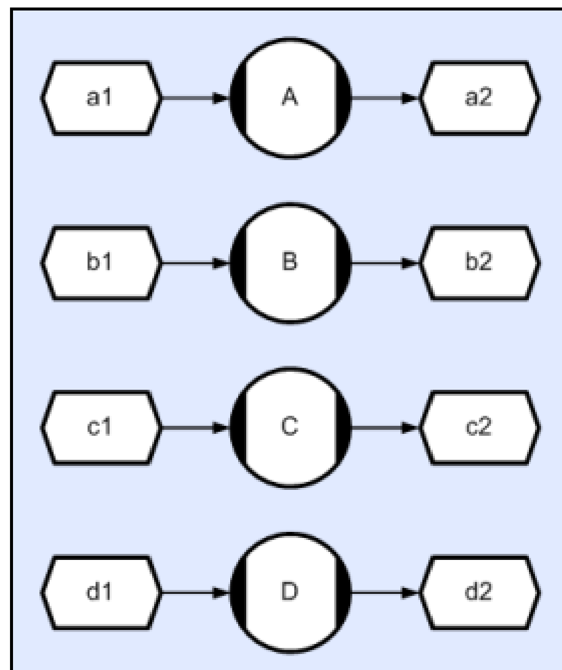
4.3 Threading

An obvious problem with the scheme as described so far is that it is likely to lead to arbitrarily large numbers of threads (especially for data-parallel applications) and worse than this; most of these will be blocked waiting for their collecting, multiplexing and so on; to complete. The operation of collection (waiting for multiple events) is also likely to incur multiple context-switching in order to complete and allow each thread to execute.

However, given that each thread is waiting for a sequence of operations to complete; there is actually no reason why each thread's blocking pattern (their complete sequence) couldn't actually be fed into one single multiplexor so that in actual fact the entire application could be executed by a pool of system owned worker threads (another reason that the basic operations need to be compose-able).

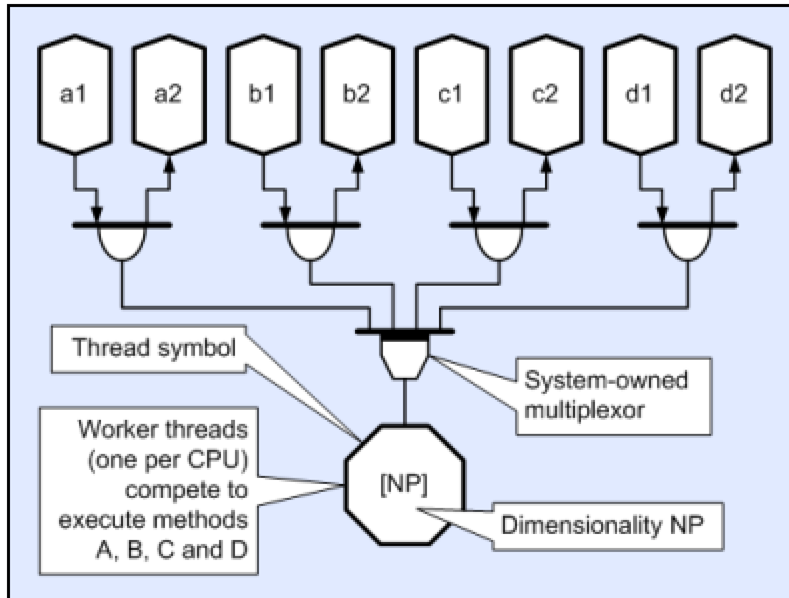
Once this can be achieved then at local SMP scope anyway, all that is required is a number of system owned worker threads that compete with each other to extract tasks (completed sequences of operations) and execute them.

The diagram below shows four CDL methods, each of which will execute when their inputs and outputs are available.



Below is a logical equivalent where the four compound method providers are fed into a single multiplexor, and worker threads compete to retrieve events from the multiplexor. In the CORE runtime case, each event is tagged with the multiplexor link number and so each worker can determine the appropriate method code to execute. Since every method in the application can be multiplexed into a single object, the application can therefore be executed by any number of worker threads (typically one per CPU).

Copyright © Connective Logic Systems Ltd 2009



This is the basis of the event manager described in a later section and is probably most easily understood by considering the CDL diagram above. This means that the bubble objects in the diagrams (CDL methods) are not threads at all but are in fact objects more akin to call-backs. It also means that although they may appear to be 'blocked' whilst collecting their inputs, in fact nothing is blocked (unless there are less executable tasks than workers), there is a considerably reduced context switching overhead, and each method only requires a few hundred bytes of storage rather than the enormous requirement that would result from large numbers of thread stacks. The reductions in context switching reduce the scheduling overhead and typically allow for finer grain concurrency.

In general there is no point in creating more than 'N' workers; where 'N' is the number of thread cores available to the executing process, because at a given priority there is no advantage to context switching to an adjacent task (unless the application is real-time). So at initialization, the runtime makes a call to determine the number of cores available and uses this to create its thread pool. Since this is transparent to the application, holographic applications need have no knowledge of their host's core-count and will simply scale to whatever platform they are executed on.

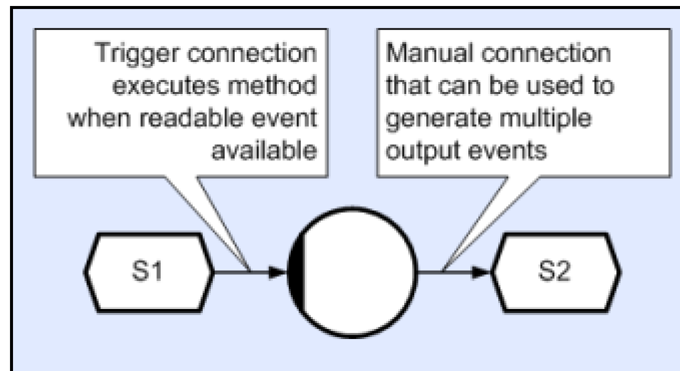
Most applications to date have required real-time preemptive execution and this can be achieved by assigning each CDL method a priority (as with threads). At initialization time, the runtime will transparently create a multiplexor and thread pool at each utilized priority. So the (default) total number of worker threads is PxN where P is the number of priority levels used by the application (seldom more than 3 or 4) and 'N' is the number of cores available to the executing process. Note that in the multiple machine case, the root master directs a distributed scheduler that ensures that the global thread pool (all workers in the distributed colony) execute preemptively.

4.4 Blocked Workers

As explained earlier, methods do not generally execute until all of their inputs and outputs are available, and this means that in general they do not need to block until completion. This means that all methods that share a given priority can usually be executed by a single worker thread (if required) and this has the benefit of repeatable execution, which in turn means that applications are far easier to debug. Typically, applications are first developed with a single thread, then multiple threads, and finally (if required); fully distributed.

However, in the special case that a worker-thread blocks on an operation that is a scheduling pre-requisite of adjacent workers, it is possible for the whole system to block with what is referred to as a 'log-jam'.

Copyright © Connective Logic Systems Ltd 2009



Consider the example above. If the method produces an arbitrary number of outputs each time it executes, then it is not possible to open all the outputs prior to execution, and it is therefore possible that the method could fill the output store and block. If this were executing with a single thread then the situation would be unrecoverable because there would be no available worker to read data out of the store and unblock the writer. This problem can be solved in several ways;

1. Ensure that the system has more workers than block-able methods. This solves the problem, but means that the application cannot be debugged with a single thread. It also means that whilst workers are blocked, the system may have less runnable threads than CPUs and this may affect performance.
2. Let the system detect potential blocks and spawn threads automatically. This is a safe scheme, is transparent to the application, and is the scheme adopted by the CORE runtime.

What this means in practice; is that only in the case that all method workers block outside of the runtime, can a log-jam occur. In order to address this pathological case, CORE therefore allows methods to be identified as having their own thread of execution. In practice, the concept of collection means that it is seldom necessary for workers to block for anything other than the purpose of I/O (e.g. reading from a socket).

4.5 Event Propagation

Although much of the detail has been skipped over, it should be apparent that event based execution can provide a concurrent alternative to conventional sequential stack based execution and that generality for data and task parallelism can be largely achieved by the provision of half a dozen or so arbitrarily compose-able event operations. Typically, the top level of the application will execute using an event based paradigm, but the lower level (finer grain) functionality remains sequential and stack-based.

When developing CLIP applications, one of the most important design tasks is to decide how far down to take the event based approach (CDL circuitry). Granularity is usually limited by the fact that event management is more expensive (in terms of CPU usage) than stack based execution and so there comes a point where the liberation of additional concurrency no longer justifies the event management overhead. So keeping the cost of event propagation low is absolutely essential and this section provides a (simplified) overview of how event propagation is implemented in the CORE runtime.

In most cases, there is a three stage operation; request, reply, rescind; and in almost all cases these simply involve unlinking tokens from one queue and re-linking them into another. Most event propagations also require one lock and one unlock operation but these are seldom contended.

Typically, if a consumer request cannot be satisfied, then the consumer leaves a token (referred to as a wait-frame), but does not block the executing thread (considerably reducing context switching). When (at some time in the future) the provider does have the requested event, it unlinks the wait-frame, points the wait-frame at the event, and then sends the populated wait-frame back down the consumer tree (this is the reply phase of the cycle). Eventually it reaches the root of the tree and if there is a waiting worker thread then the event is passed to the thread, which is then unblocked. If there are no workers waiting, then the event is queued for later processing by the next available worker.

Copyright © Connective Logic Systems Ltd 2009

Finally, when the worker has finished processing the event, it is returned to the object that provided it (and so on all the way back through the event tree). This is the rescind phase of the cycle, and once completed the system is back to the quiescent state that it was in before the earlier request, and is then ready to receive another request.

Note that the precise behavior of each object (distributor, collector, store etc) will be specific to their type and this will reflect the particular operation that the object performs. Although it is an advanced topic; most objects have a configurable reentrancy attribute that allows multiple request cycles to be active in parallel. So this might mean for example, that multiple consumers could be processing events retrieved from a multiplexor simultaneously.

In fact, as discussed earlier, CDL methods are implemented by multiplexing all method event trees, and doing so with a reentrancy of 'N' where 'N' is the number of worker threads (by default 'N' is the number of CPU cores available to the process). Also note that the term 'event trees' derives from the fact that in the general case consumers can collect collections and distributors can distribute to distributors; in fact as mentioned earlier, all event operations are necessarily compose-able.

The event model (as opposed to typical data-flow models) means that data only needs to be moved if the producer/consumer pair are in disparate memory spaces, and this considerably improves performance for many applications. Gratuitously moving data around main store is an expensive high latency operation particularly when data is shared by multiple consumers; each of which requires a separate gratuitous copy.

Event management may not be easy to follow from the description above and is probably best illustrated by example. Consider the case of two threads (methods will be considered later); the first writes to a transient store, and the second reads from the same store. In this example, the store is assumed to have a buffer depth of two (double-buffered). Below is the writer and reader code;

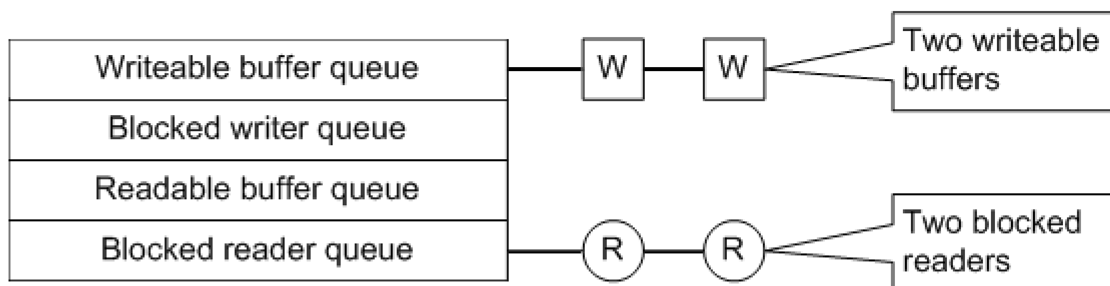
Writer Code

```
buff = waitOpenWrite( store_id );  // Wait for writeable buffer ref
populate( buff );                  // Populate buffer ref
close( store_id);                  // Close and unblock reader
```

Reader Code

```
buff = waitOpenRead( store_id );   // Wait for readable buffer ref
useBuff( buff );                   // Use the input
close( store_id );                 // Close and unblock writer
```

Essentially, a transient store consists of four queues and some additional state that doesn't concern this example.



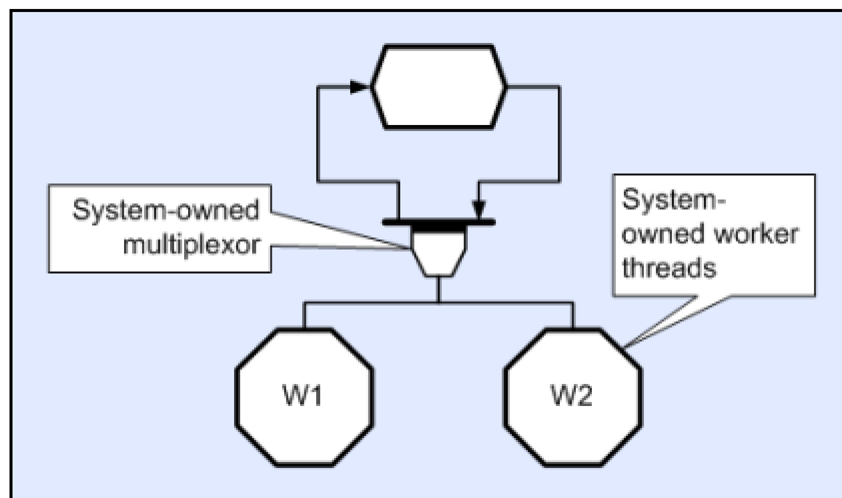
In practice, it is not possible to determine the order in which the write and read threads become runnable (in the multi-core case this may be simultaneous) and the exchange must work in all cases. This can be considered as a four stage operation;

Copyright © Connective Logic Systems Ltd 2009

1. The writing thread issues a write request to the store. If a buffer is available it is unlinked from the writeable buffer queue and its address returned to the writing thread, which can then continue to execute. If no buffer is available, then a wait-frame that references the executing thread is added to the blocked writer queue. In this example, a thread that could not retrieve a buffer would block in the conventional sense (this will not be the case for methods).
2. The reading thread issues a read request to the store. Again, if a buffer is available it is unlinked and its address returned to the reading thread which can also continue to execute. If no buffer is available then a wait frame is queued and the executing thread blocked.
3. When thread ‘A’ has completed writing its data to the buffer it ‘closes’ (rescinding the buffer). If the reading thread has already requested (and its wait-frame is queued) then the wait-frame is assigned a reference to the buffer (which is now readable) and in this case the reading thread is unblocked and executes. If however, the reading thread does not yet have any outstanding requests, then the buffer (now readable) is queued in the readable buffer queue and this means that when the reading thread next requests, it will have an available buffer (and its request will immediately succeed).
4. When the reading thread has completed reading its data from the buffer it also ‘closes’. But in this case; If the writing thread is queued (waiting to write) it will be passed the ‘now-writeable’ buffer and unblocked; otherwise the writeable buffer will be queued so that the writing thread’s next write attempt will succeed without blocking.

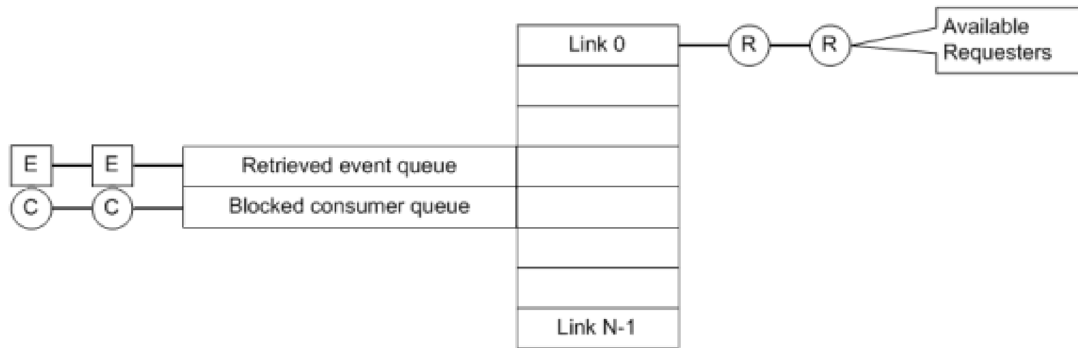
In this example (two threads and two buffers); both read and write threads could execute in parallel, with the read thread processing data from iteration ‘n’ whilst the write thread is actively writing the next ‘n+1’ iteration.

In practice, it would be very unusual for a CLIP application to use threads to perform the kind of read/write construct described above; it would be far more likely to use CDL methods. As explained earlier, these are essentially call-back functions that do not need to block in the conventional sense, and are implemented through process wide multiplexors (one for each active priority), so effectively the example above is actually realized using the following circuitry;



Before considering how this example would work, it is worth considering how multiplexors themselves work. The diagram below shows the structure of a multiplexor. In this case, each link owns one or more wait frames that it uses to request events from its connected provider in the manner described earlier. It also has a queue for ‘already-retrieved’ events that have yet to be consumed, and a queue for ‘blocked-consumers’ that are ‘as-yet’ unable to retrieve events from the multiplexor (clearly they cannot both be populated and so in practice only one queue is required). The reentrancy is ‘N’ where this is the number of worker threads, and in this case, links are ‘requested-from’ using a round robin (as opposed to prioritized) ordering scheme.

Copyright © Connective Logic Systems Ltd 2009



Any consuming object can retrieve events from a multiplexor, and the process involves the following stages;

1. The consumer requests an event using a wait-frame. If the multiplexor's 'already-retrieved' queue contains an event then the consumer unlinks it, references it from the wait-frame, and then processes the populated wait-frame (queues it locally or passes it further down the consumer chain).
2. If the multiplexor does not contain any 'already-retrieved' events then the consumer must traverse each link until it either retrieves an event, or reaches a situation where all links are blocked. In the latter case, the consumer adds its wait-frame into the 'blocked-consumers' queue and returns a blocked status (but crucially does not block the executing thread in the conventional sense). In the former case (where an event is retrieved on a given link), the wait-frame is assigned a reference to it, and the consumer then processes the populated wait frame.

So the request stage of the cycle may or may not successfully return an event, but does not block the executing thread (still free to continue requests lower in the tree and/or execute any runnable methods).

However, the multiplexor's request stage typically lodges a number of wait-frames (those owned by the links) that are now queued on their providers (objects connected to each of the multiplexor's consuming links). The next stage is therefore the reply stage and this is activated when a link's provider that was not available during the request cycle, subsequently becomes available.

1. The provider will find the multiplexor link's wait frame in its 'blocked-consumer's' queue and so it needs to unlink the wait-frame, assign a reference to the provided event, and then pass the populated wait-frame back to the multiplexor link that requested it.
2. If the multiplexor owning the link has wait-frames in its own 'blocked-consumers' queue then the first one is unlinked, a reference to the link's wait-frame (which is referencing its providing event) is assigned, and the populated consumer's wait-frame is then passed to the multiplexors consumer.
3. This continues all the way down the event tree until an object is reached that does not have a consumer, and at this point the accumulated event tree is linked into the objects 'already-retrieved' queue for later consumption.
4. In the special case that an actual blocked thread is the root consumer, then the event is referenced in the thread's wait-frame, and the thread unblocked in the literal sense (so it becomes runnable in an operating system sense and wakes up with a successfully retrieved event).

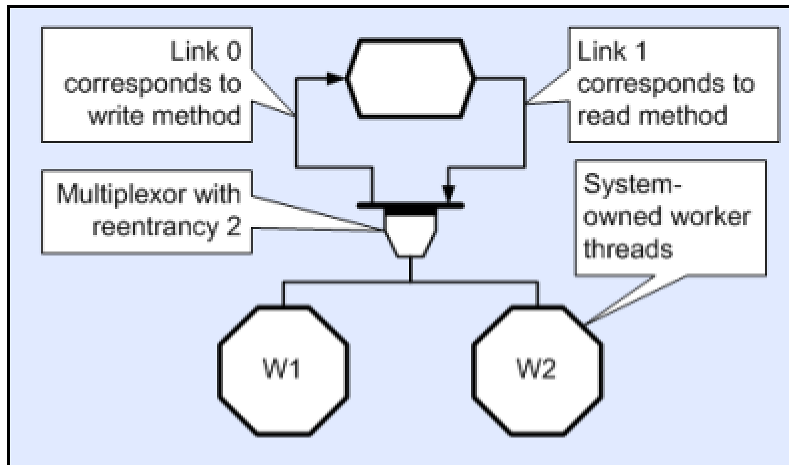
The net result of the request/reply cycle is the retrieval of an 'open' event tree. Once, this has been processed by its root consumer, the event tree can be 'closed' (the rescind stage of the cycle). This is typically executed automatically when control returns from user method code and in the case of the multiplexor, involves the following stages.

1. The first task is to close the event that was provided to the closing link. This sets off a recursive chain of close operations that means that leaf providers are actually closed first.

Copyright © Connective Logic Systems Ltd 2009

2. If the multiplexor has any ‘blocked-consumers’ then the request cycle is repeated for the closing link and if successful, the first blocked consumer in the queue is passed the event.
3. If the multiplexor has no ‘blocked-consumers’ then the link wait-frame is marked as quiescent so that subsequent requests to the multiplexor can launch a request on the closing link.

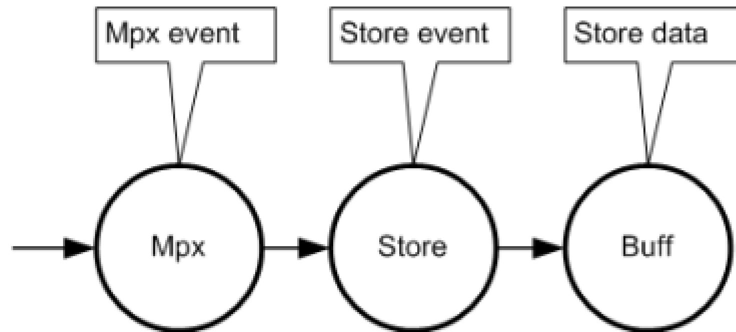
It should now be clearer how the example would work. In the explanation that follows, the ‘writing’ method is connected to link 0 and the ‘reading’ method to link 1; two worker threads are assumed. The exact order of events will depend upon timing, but the following would be typical;



1. The first worker to execute requests an event from the multiplexor. This causes a request to be issued on link 0 which requests a write to the store. First time in, the store will have two write-able buffers available so the first is unlinked and propagated back to the worker that can then start executing. As an aside, the retrieved link number (in this case 0) is used by the worker to index into a table of call-back addresses and in this case the worker will therefore execute the ‘writing’ method code.
2. The second worker now executes (in parallel if there are two or more cores) and first tries to retrieve on link 1 (the multiplexor in this case uses a round-robin scheme). Assuming that the first worker is still writing, there will be no read-able buffers populated yet, so link 1’s wait frame is queued in the store’s blocked reader queue.
3. The second worker now continues to the next link (wraps round to link 0). What happens next depends on the reentrancy assigned by the user to the ‘writing’ method. If it has reentrancy greater than one, then the multiplexor will issue a second request on link 0 and acquire a second write-able buffer. However, for the purposes of this discussion, it is assumed that the ‘writing’ method is not re-entrant and so the multiplexor will not issue a request and the second worker will be added to the multiplexor’s ‘blocked-waiter’ queue and will then block in the conventional sense.
4. At some point, the ‘writing’ method will complete execution (having populated the write-able buffer) and return from the user code associated with the method. The worker that executed the write will now ‘close’ its multiplexor event. This then closes the store that was opened for write as described earlier. In this case, the store will have a blocked reader (from the second workers earlier request) and so the now read-able buffer will be passed to the waiter and this will propagate down the event tree until it reaches the blocked second worker. The second worker is assigned the event tree (which contains a multiplexor wait frame and a read-able buffer) and is unblocked. The second worker now begins execution of the ‘reading’ method code.
5. Having completed its first task, the first worker now requests another event from the multiplexor and in this case will retrieve a second writeable buffer (the store is double-buffered in this example). It now executes the writing method for a second iteration and at this stage both worker threads are executing in parallel; the first is writing the second iteration, and second is processing the first iteration.
6. What happens next will depend upon which method completes next, but it should hopefully be clear that the resulting execution is exactly the same as it would have been in the case of two dedicated threads executing the reading and writing methods directly.

Copyright © Connective Logic Systems Ltd 2009

Whatever order the events occur in, the net result is that consuming threads end up with a reference to a multiplexor event (populated multiplexor connection), and this in turn has a reference to its provider (in this case a transient store event), which has a reference to its retrieved data buffer. In order for the thread to access the data buffer it simply has to traverse (in this case) three pointers. In practice, the translator will automatically generate this code, and the thread will actually be returned a reference to the store event. In the case that the multiplexor and/or store are actually hosted on a remote machine, the runtime will transparently move the data between machines and so the application code will not be affected (the runtime will still return a reference to the event).



It should also be clear from the above, that in the typical case where there are many more executable methods than workers, that most of the time the system owned multiplexor will have ‘available-events’ and the worker threads will not block. Also, note that in this specific example, both methods could have been given a reentrancy of ‘N’, the store could have been given a depth of ‘2N’ and the result would have been a concurrency of ‘2N’ (potentially keeping 2N cores active).

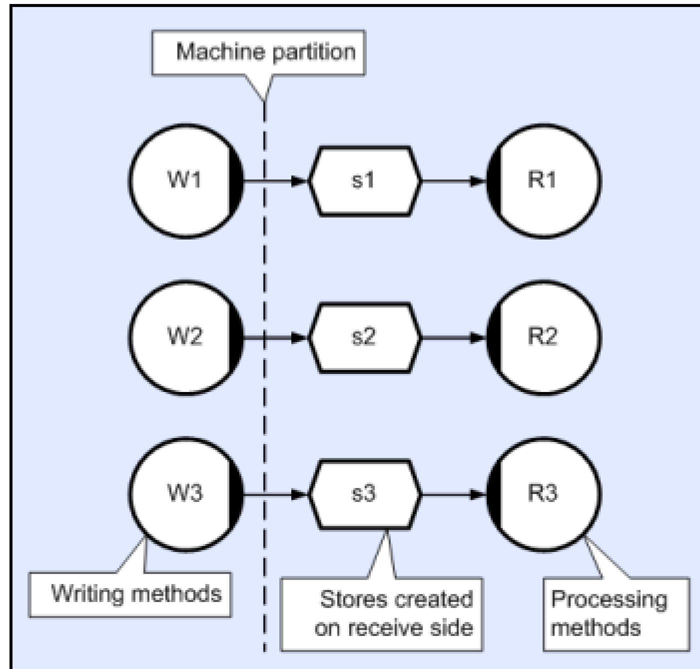
In practice leaf providers like stores and/or semaphores are seldom more than two or three objects away from root consumers like methods, call-backs or threads and few CLIP applications to date have spent more than 2% of their CPU budget executing in the runtime (although this is clearly dependant on choice of grain size). This means that they can scale extremely well. Also note that the relatively small overhead required for event propagation, replaces to a large extent, the much bigger overhead that would have been required to context switch between conventional operating system threads.

4.6 Flow Control

The previous section addressed event propagation but did not consider the problems that arise when two adjacent objects (exchanging events) are located in different executable processes. This raises the general problem of inter-process flow control and is important for a number of reasons. It is essential for example that a process can be stepped in a debugger without loss of data from neighboring processes. This section addresses this specific aspect of inter-object communication.

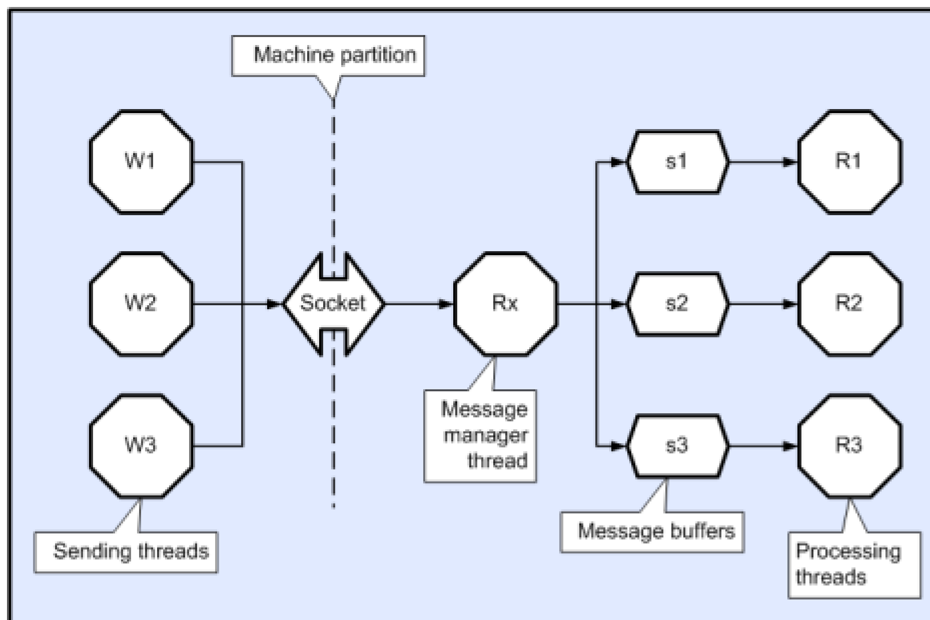
The diagram below illustrates the very simple case of three methods on one machine, writing to three methods on an adjacent machine and assumes that their exchanges are not synchronized (requiring three independent parallel lines of communication). Clearly, if one particular recipient is unable to receive due to a transient load or blockage, its neighbors must not be prevented from continuing to exchange events. This is a basic requirement but one that can raise issues.

Copyright © Connective Logic Systems Ltd 2009



Because sockets (and most equivalents) are relatively expensive resources, each process pair is typically constrained to use a single socket, even though logically there are many communicating pairs (as in the above diagram). This is probably a practical constraint for any generic scheme.

Typically, writing threads send to the socket directly, but in the receiving process there is usually a single 'message manager' thread that reads messages and passes them to the appropriate local thread (or processes them directly). In most cases, the message has a 'type' in its header, and the receive thread uses this to determine appropriate action. This scheme is illustrated below using CDL notation, although in practice distributed CLIP applications do not need to use this kind of approach unless they are connecting to a legacy non-CLIP application (they only need the simple method code above).



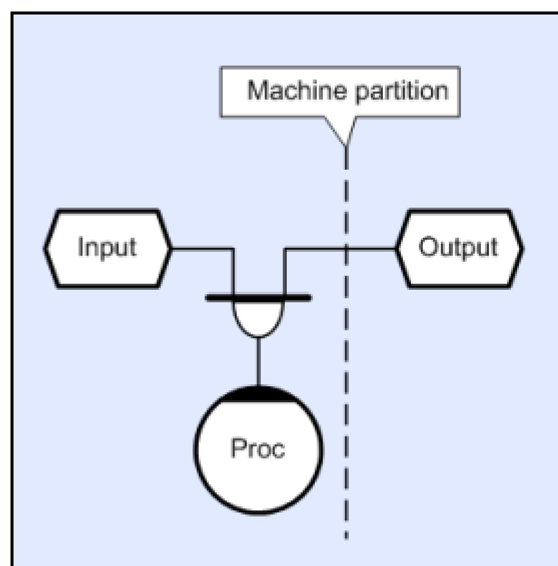
There are a number of problems with this approach. If for example, one of the local processing threads is busy or blocked, then the receive thread could also block because when the receive buffer eventually fills, it has nowhere to write subsequent incoming messages; this in turn will block all of the incoming messages behind it. So if one particular line of communication is blocked, but adjacent lines must continue to flow, then the blocked line's data must still be read from the socket regardless, and must therefore be put somewhere until its recipient is ready to receive it.

Copyright © Connective Logic Systems Ltd 2009

Unfortunately, in the general case, there is no limit to the number of messages that could be sent to a given receiver, and so there is no guarantee that eventually the receiving process will not run out of storage space. What this means, is that the receiver must have a finite number of available receive buffers, and the sender must not send until the receiver has a buffer that is available to receive; and the problem that this then raises is latency. It might not be practical for the sender to issue a request to send and then wait for an acknowledgement each time it needed to send because the turn around time would probably be large compared with the elapsed transmission time (introducing latency).

Without solicitation, the receiver cannot simply send the store's 'ready for write' event each time a buffer becomes available because in the general case a store may have many writers and there is no way to determine which one should be sent the event without knowing the identity of the senders that intend to send (and these could be more numerous than the number of available buffers). Obviously the 'ready to write' cannot be sent to more than one potential sender without the risk that they will all send simultaneously.

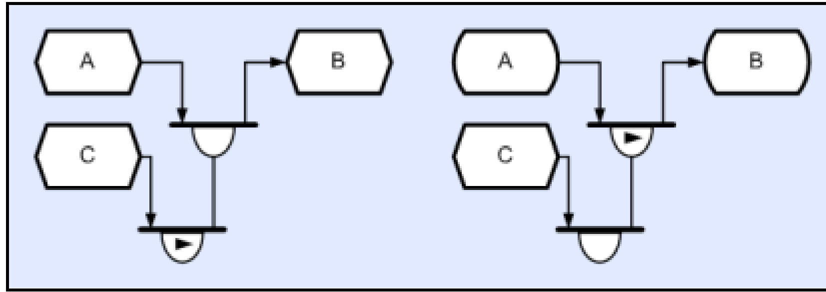
Collectors (introduced earlier) provide a solution to this problem. Consider the case of a method that reads input from a local store, and then writes output to a store that is assumed to be located on a remote machine. The diagram below shows this case with the collector drawn explicitly (to make the point clearer).



There are two collection schemes; sequential and random. In the former case, events are requested in strict connection order (to allow for the controlled acquisition of locks), and in the latter case they are collected randomly (first come first served). It is generally useful to have write-access to transient stores early in the sequence; this will reduce latency, especially if the target store is on a remote machine. What this means is that methods typically request and acquire, write-access, before their 'read-able' inputs arrive (hiding latency). In practice, the runtime will generally start requesting the next iteration's write access as soon as a method returns from its current iteration, in fact it is possible to run-ahead and request more than one write (but this requires extra buffers).

Similarly, circuit logic can be used to ensure that write requests are not issued until a method has the potential to execute (so that write-able buffers are not gratuitously requested). An obvious example would be to organize collection so that particular write requests are not issued until critical read/write requests have completed. The first example below illustrates a compose-able case where random write collection will only begin when a first critical read has arrived. The second example illustrates a random collector, that requests from a child collector that in turn ensures that a set of arbitrated stores are collected in a specific order.

Copyright © Connective Logic Systems Ltd 2009



As discussed above, there is one further property of collectors (referred to as reentrancy), which allows them to have more than one asynchronous collection sequence active at any given time. This means that a given method for example, can issue write requests for the next 'N' invocations in advance, and this technique has been used for a number of latency-critical real-time applications.

It should be noted however, that if there are more requestors than buffers then there is no guarantee that latency will be reduced optimally; but crucially, the scheme will 'limit' memory consumption, will never block, and usually realizes minimum latency (in most cases it hides it altogether).

4.7 Accretion

'Accretion' provides a means of statically partitioning an application into one or more component processes. 'Colonization' (the master/slave model) then provides the option to dynamically schedule any of these 'processes' across any number of 'slave' machines.

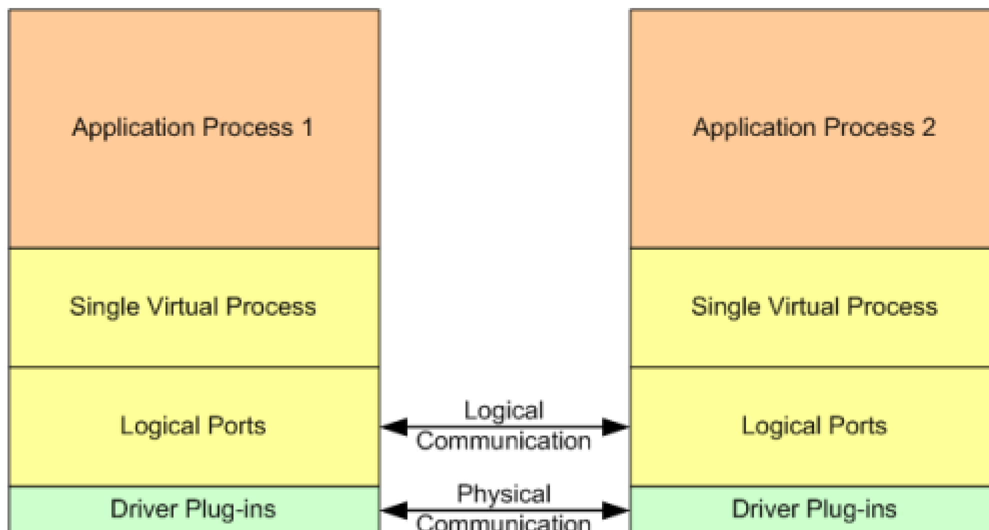
As discussed earlier, CLIP applications comprise one or more interconnected circuits and as such do not have any inherent notion of discrete executable processes. In order to allow maximum flexibility, the mapping of logical circuitry to physical processes is a separate second-stage operation. Each of these processes can then be executed autonomously, or if required, can be executed holographically through the recruitment of nest-able colonies. This therefore requires a number of runtime services.

When each process first executes it needs to locate each object that it is responsible for connecting to. By default consumers connect to providers but this is a configurable option. The runtime therefore needs to provide a discovery service and this can use a dedicated registry server, or for turn-key systems can nominate any accreted process to act as registrar.

The first stage is to locate and connect to the registrar (unless the executing process is the registrar). The second stage is to instantiate all circuitry and to register public objects (visible to adjacent processes). The third stage is to locate and connect to all server processes. Note that connection to adjacent processes is a driver issue and the runtime implements a logical port abstraction that is implemented from the available platform protocol. In the case of asymmetric shared memory systems like the Mercury platforms communication is implemented using data passing (sender writes directly into receivers memory), in the case of networks (typically TCP/IP) data is transmitted and received through sockets.

The important point is that these are driver issues and the runtime simply sees a logical port abstraction that can be implemented from more or less any communication mechanism (to date anyway). The application that sits above the runtime's logical port layer only sees a single virtual process that logically communicates by reference.

Copyright © Connective Logic Systems Ltd 2009



The result of this is that if two objects find themselves in the same symmetric address space they will communicate by reference, if they are in separate asymmetric spaces they will move the data directly to the target destination (only one physical move, typically by DMA), and if they are separated by some serial connection (e.g. Ethernet) then they can communicate using a socket style scheme. Because these are driver issues, communication can be optimally implemented for any given platform.

4.8 Scheduling and Colonization

The earlier 'Event Propagation' section provided a brief overview of how the event manager works, and in fact at SMP (autonomous symmetric shared memory process) scope, load balancing is fairly straightforward because most target operating systems will schedule the worker threads to available CPUs very efficiently without intervention. Also, as explained earlier, methods at a given utilized priority will have their own prioritized worker threads and so preemptive execution is also taken care of.

However, in years to come, as the number of cores increases, bus bandwidth is expected to become a bottleneck and cache utilization will therefore become ever more important. This may mean that the operating system scheduler may need to be over-ridden so that the CLIP runtime can use its knowledge of circuit connectivity to make optimal use of this cached data. In fact this is already an issue at network scope, and this section explains how the distributed scheduler utilizes knowledge of method connectivity to manage processor load balance and also minimize network bandwidth for the more general case of distributed applications (more than one executing process). It should be apparent from the explanation that follows, that the same algorithm could be applicable to bus bound shared memory platforms.

In the general case CLIP applications are accreted to more than one sub-system and each of these is then scheduled holographically. Each of these is referred to as a 'colony'. However, each colony can also be scheduled as a colony of colonies and so on. This section describes the issues that arise from the scheduling of a single sub-colony.

Essentially, there are three categories of task (implemented through methods etc). The first of these are tasks that are constrained to execute on particular machines (static scheduling). Typically, this occurs because the task needs particular resources that are only available on certain platforms and these might include; storage devices, accelerator boards, GPUs and so on. Under these circumstances the programmer can instruct the scheduler to pin these particular tasks so that they always execute within the specified logical process.

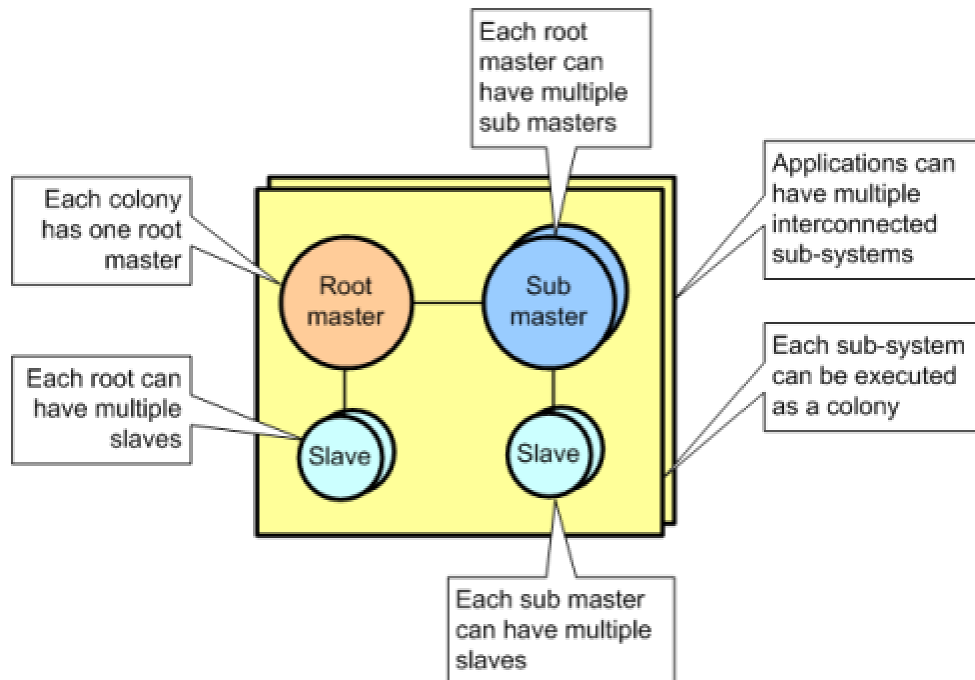
The second type of task would be those that are not physically constrained to execute on any particular machine, but have a high investment in state (accumulated from earlier executions). An example would be a Finite Difference Time Domain (FDTD) calculation that needs to keep all of the data from the previous time-step in order to calculate for the current time-step. These tasks can be pinned to a particular machine but given a hysteresis so that they only reorganize periodically or when directly instructed. In the FDTD case this would occur if a slave were recruited or retired, and this

Copyright © Connective Logic Systems Ltd 2009

would allow the scheduler to redistribute the calculation in order to accommodate the new load-balancing scenario and optimize scalability (see memory management below).

The third type of task would be those that can be 'floated'; and in this case the scheduler is free to dynamically allocate these tasks to any available machine in the colony. Note that this is possible because as the term 'holographic' suggests, all of the slave processors have all of the code to execute all of the tasks in categories two and three above.

As seen earlier, each task is uniquely identified by their system owned multiplexor link number, and so dispatching a task to a slave is a lightweight operation that does not involve the dynamic transfer of code (for most platforms). However, in order to execute a given task, the slave needs access to inputs and outputs, and in some cases state from the previous execution. This means that shared data also has to be distributed holographically and this issue is addressed in the memory management section below.

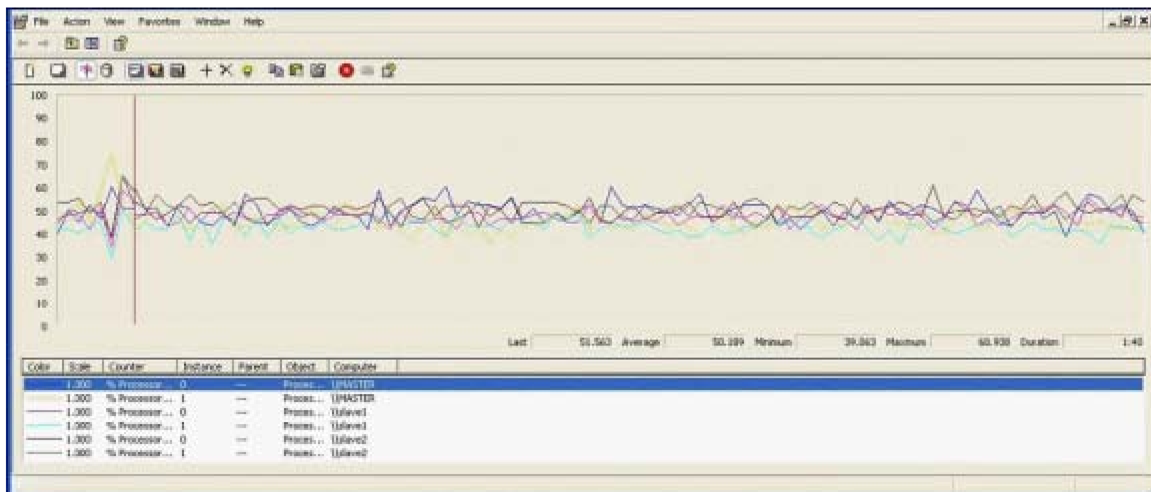


Each sub-system colony has one 'master' process that executes an instance of the scheduler which needs to keep track of the current distribution of data and the current distribution of tasks; and use this information to decide where to execute the floating task stream. The colony also contains a scalable number of slave processes that execute the task stream and keep the master informed of their progress. The scheduler has four goals;

Firstly, for real-time applications, it has to ensure preemptive execution. This means that there should never be an unscheduled task at priority 'N', if there is a machine in the colony executing at 'M' where 'N' is greater than 'M'. Typical real-time systems often need to execute more than one processing chain in parallel, and a standard technique is to raise the priority of the higher rate chains; so a 10Hz chain would have a higher priority than a 4Hz chain. This means that the 10Hz tasks aren't backed up waiting for the slower 4Hz tasks to complete (otherwise unacceptable latencies would be likely). Without colony scope preemption each chain would have to be given its own dedicated sub-set of the hardware and would lose many of the benefits of holographic processing (see 'benefits' below).

Secondly the scheduler needs to keep the total load as evenly distributed as possible and this is essential for good scalability characteristics. Since there are usually many possible scheduling permutations that meet the preemption requirement equally well, the scheduler can identify a sub-set that also keeps load as even as possible. The profiler output below shows a network application running on a heterogeneous collection of machines (different core speeds and core counts).

Copyright © Connective Logic Systems Ltd 2009



Thirdly, if there is still more than one way to meet both preemption and load-balance requirements the scheduler will try and identify a scheme that minimizes the resulting network traffic. In other words it aims to send tasks to data, rather than data to tasks. This is discussed in more detail in the next section.

Finally, in order to provide scalability, the scheduler also needs to hide (or at least minimize) communication latencies. These can be particularly problematic when scatter/gather type approaches are applied to non-deterministic calculations (where execution time is data dependent). In order to do this, the scheduler needs to be able to start dispatching the 'next' task whilst the slave is still executing one or more 'previous' tasks, and the number of tasks that a slave processor can be simultaneously allocated is referred to as 'task-overlap'.

By default, all slaves have an overlap of two; so that one task can collect its inputs whilst another is executing. However, in order to allow fine tuning for highly non-deterministic calculations this can be over-ridden by the application and set to any required value. If too many tasks are allocated then latency will be low, but there is a risk that the last slave to finish could in some cases, hold up adjacent slaves from progressing to the next set of tasks; and the result would be less efficient load balance.

Optimal performance is achieved when the task overlap is sufficiently big to hide latency, but no bigger; thus giving the scheduler the most freedom to achieve a good load balance. Clearly, there are applications where communication bandwidth is the limiting resource and under these circumstances it is not possible to hide latency altogether, although it can at least be minimized.

One last point that is worth making is that because the scheduler seeks task affinity (sends tasks to machines that already own their inputs), load balance is skewed (biased towards particular machines) until the system becomes heavily loaded. This is considered the best strategy because it minimizes network usage and there is little point in gratuitously spreading the load until the 'whole' task requires it. Underutilized networks may not therefore appear to show balanced load, but this is intentional.

4.9 Memory Management

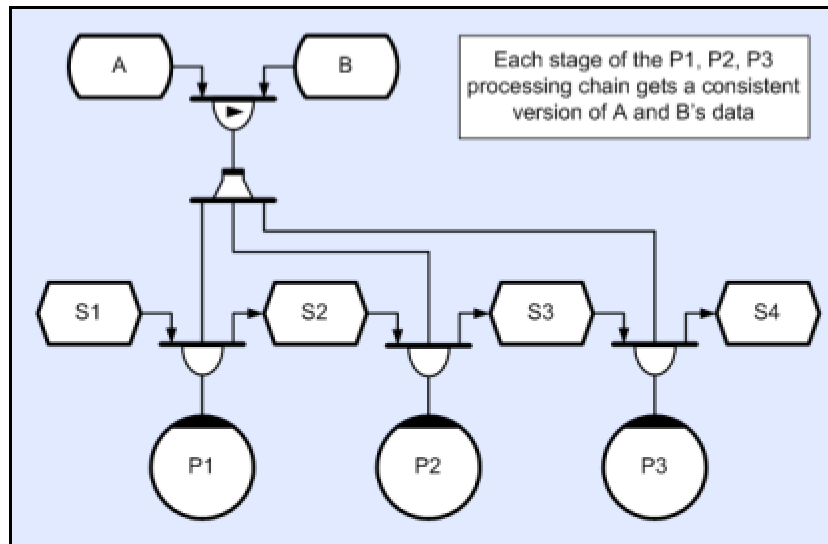
CLIP applications provide four types of data object and this section explains how the distributed memory manager ensures that each process in the application maintains a consistent view in each case.

The first object to consider is the arbitrated store which deals with shareable persistent storage (an equivalent for class state, file-scope data etc.). There are several modes of operation. In the first, remote references (accesses from processes that do not own the store) are directed to the definitive instance (typically owned by the colony master). If the remote process is updating the store, then it may need to transiently lock it. The manager has to keep track of this, so that if the client process exits, the lock can be automatically relinquished and avoid leaving the server in an unrecoverable state.

In the second mode of operation, the client can subscribe to the store's updates which are then transparently broadcast to all subscribers. This latter mode can save bandwidth for cases where store reads outnumber store writes significantly. Also, if the carrier protocol supports a multicast, then this

Copyright © Connective Logic Systems Ltd 2009

can be a very efficient alternative to direct read access. However, because of communication latencies there is no guarantee that all readers will see the update simultaneously. This may not matter for some cases such as updating a GUI screen to reflect an updated value or values, but in other cases it does.



If it is important that multiple consumers have a consistent view of a particular instance of persistent shared storage then the best way to ensure consistency is to have each consumer access the store through a shared distributor object. The data 'being read' will not be update-able until all consumers have finished with the data, but if it is suitably buffered, updaters will not be blocked if they attempt to access the store while the distributor holds it open. As soon as the distributor closes, the updated data instance will automatically become the current data instance.

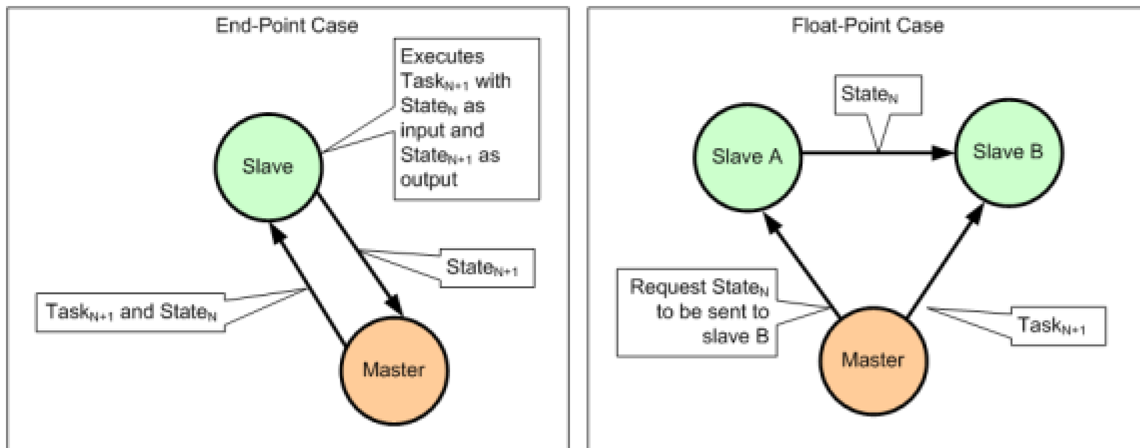
There are also two types of non-shareable persistent storage than can be associated with a method instance and these are referred to as 'workspace' and 'state' respectively.

Workspace is provided for two cases of operation. The first is where the method needs scratch workspace that does not need to persist after execution, but cannot be created on the standard stack (maybe variable-sized or too large); and the second is where storage is used to create write-once /read-many data such as tables of coefficients that are populated at initialization and persist. In both cases, this data can be re-instantiated within each slave process and does not need to be moved (it exists wherever the method is scheduled).

The second type of non-shareable persistent storage is the kind that is updated and maintained between method invocations, and this is referred to as 'state'. Examples would be some filters, moving averages, random number generators and so on. This data needs to be present in the slave before a given method can be executed. So if a method is executed by slave1 for iteration 'N', and the scheduler dispatches the same method to slave2 for iteration 'N+1', then execution must be delayed until slave-2's state is up to date. There are two schemes, and these are configurable and mixable.

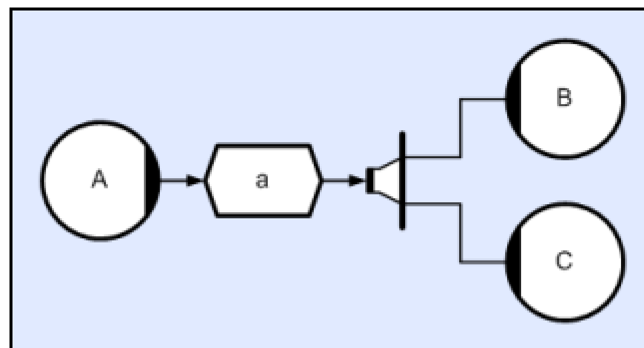
In the first 'endpoint' case, the master process maintains the definitive copy of the designated method's state, and after each invocation, the method's state object is flushed back to the master process. This is generally required for mission critical applications where losing state might be serious, and if the producing slave does exit before its state has been safely returned, the scheduler can reschedule the task to a surviving slave so that the state is simply recalculated. The scheduler now needs to send its copy of the state each time it schedules the method; although as described above, the scheduler will try and send the method invocation task to the slave that holds the most up-to-date data, and so if this results in the method executing in the same slave on successive invocations there will be no need to re-send the state. It is also worth noting that in practice most methods are actually stateless.

Copyright © Connective Logic Systems Ltd 2009



In the second 'float-point' case, the latest method state object is left in the last slave to execute it (and not flushed to the master). As with the previous case, a slave cannot execute a method until its state has arrived but in this case the operation may involve sending two messages; the first instructs the slave owning the definitive state to send it to the target slave, and the second involves sending the task itself to the target. Only when both have arrived at the target can execution begin. This scheme can potentially save bandwidth because it avoids the case where state is sent back to the master, and then resent again to some new target (replaced by one direct send from a method owning definitive state to a method not owning it). The limitation is that whilst slaves can still be gracefully retired, if they fail at a critical time, definitive state could be lost. Note that the latencies associated with the movement of state and data are usually hidden by the 'task-overlap' capability and slaves are seldom blocked waiting for data to arrive.

Finally, the fourth type of storage is shareable transient data; which is the type of data held in transient stores and provides a distributable equivalent for local stack data (invalidated when no longer referenced). The message manager needs to ensure that all readers access the latest instances consistently, but needs to do so without moving data gratuitously. As with method 'state', transient data can be designated as 'end-point' or 'float-point'. In the former case, remote writes are flushed back to the definitive object, but in the latter, they are not. The same advantages and disadvantages that applied to state data also apply in this case.



The scheduler and memory manager collaborate so that when the former allocates a task to a given slave, the latter can update its bitmaps to reflect the fact that the scheduled method's output is current and definitive in the given process. So in the example above, the definitive data that exists in the slave that executed method A (slaveA) needs to be copied to those slaves that execute methods B and C (both require 'a' as input).

The scheduler also uses the memory manager's information to select the 'best' machine to execute a given task. So in the example above it will attempt to execute methods B and C on the machine that executed method A (and thus reduce 'copying' bandwidth). And as with state, when the scheduler dispatches a task to a target slave, it will typically issue a number of simultaneous requests to adjacent slaves instructing them to send any outstanding copies of definitive data to the target, and execution will only begin when all data is up-to-date. In order to hide the latencies associated with these movements, the scheduler has a concept of task overlapping and this is discussed in the earlier 'scheduling' section.

Copyright © Connective Logic Systems Ltd 2009

In addition to coordinating the consistency of distributed shareable objects, the memory manager also needs to address a number of other issues and in particular needs to provide its own heaps (referred to as segments). There are a number of reasons for this. Firstly some platforms provide dedicated fast/slow memory and applications therefore need a portable means of allocating particular data objects to particular physical address spaces.

Secondly, many optimized libraries take advantage of data alignment, and so it is often necessary to guarantee object alignment in order to optimize performance. Also, some memory allocation schemes suffer from fragmentation and for real-time applications a deterministic free is also required.

For debugging purposes it is useful to have a concept of red-zoning which is a technique that has a high probability of detecting writes to a data object that have over-run or under-run and the manager implements this in a transparent manner.

Internally, the manager uses a technique that means that heterogeneous networks of symmetric and asymmetric processors can share references to distributed objects (segment and offset) globally, and again this is transparent to the application which only ever sees objects at a local virtual address.

4.10 Dynamic Recruitment/Retirement

A defining feature of holographic processing is that slave processing power can be dynamically recruited and/or retired at any stage of execution.

Recruitment is a relatively straightforward operation that involves clients connecting to servers, and slaves connecting to masters (for the particular instances being executed). The runtime provides a discovery service that means that a given application instance can be mastered on any machine and clients/slaves simply need to specify an instance name to the registry server (allowing multiple application instances to share the same resources if required). For release implementations, any application process can be nominated to act as registrar, which means that a dedicated discovery server is not required.

Retirement is a considerably more involved process and again there are two cases. In the case of a 'graceful' exit (slave retires, or is retired; without failure) the slave needs to re-distribute any definitive data, and relinquish all the critical resources that it holds (e.g. arbitrated store locks or subscriptions to distributors); it can then disconnect from the colony and exit.

In the case of a 'forced' exit however (e.g. switching off the power), the situation is more difficult. Even if the server keeps track of all resources, and even if all data is 'end-point', the scheduler and memory manager need to ensure that any/all partially calculated data owned by the 'exiting' process can be recalculated, and so the scheduler needs to be able to roll back and recalculate any data that is incomplete or lost. At the time of writing, algorithms that achieve this have been identified and their implementation is 'work-in-progress'.

Whilst this does reduce the chances of an unrecoverable problem, it still leaves masters as single points of failure. This can however be addressed by the application, and in fact CLIP was used to develop a redundant server baggage handler demonstrator that could recover from forced exit. It should however be possible to take a generic approach to the problem (based on method rollback and data sharing), and again this is work-in-progress.

4.11 GUI Interfacing

Graphical User Interfaces (GUIs) are required for a large proportion of applications and the arrival of multi-core may turn out to be an issue for a significant percentage of these. Some have argued that GUIs do not need the additional performance offered by multi-core and so it will be a non-issue. Others point out that sophisticated GUIs themselves were not required twenty years ago, but once a competitor exploited the opportunity; most had no choice but to follow. Certainly the CLIP toolset has benefited from its ability to offload various tasks to multiple cores.

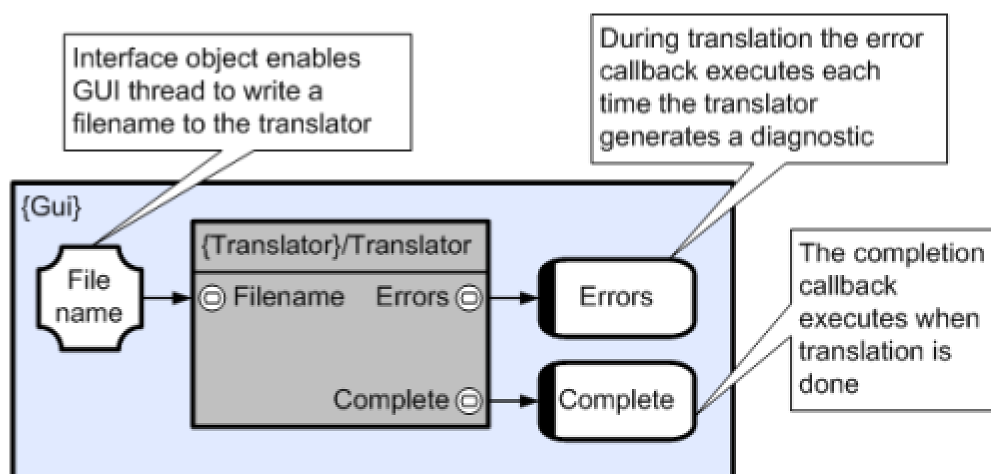
When all is said and done, if an application displays an hour glass for any significant period, then it could probably benefit from being able to offload processing to adjacent cores. Last but not least, it is worth bearing mind that at the time of writing processor clock speeds have already dropped from their peak (whilst core count is now typically four), and this trend may continue. This means that whilst single threaded performance has increased a thousand-fold over the last twenty years or so, it is now likely to decrease.

Copyright © Connective Logic Systems Ltd 2009

The issue stems from the fact that most GUI technologies (MFC, Windows QT, XT etc) are based on the notion of a message pump and are therefore intrinsically single threaded. What CLIP does is to provide a portable means of integrating message pumps with the worker thread pool allowing for a very straightforward offload model.

Essentially there are two cases to consider. Firstly there are events provided by the GUI to the application (referred to here as inputs). Secondly there are events provided to the GUI by the application (referred to here as outputs). The former are primarily controls and are typically initiated by mouse clicks or keystrokes. The latter typically comprise displayable information but also need to include notifications indicating external task status (e.g. an offloaded process has completed).

Inputs (where the GUI writes to the application) are relatively easy to deal with and CLIP provides the notion of an interface object that allows the GUI thread to connect to external circuitry and initiate blocking/non-blocking operations in the same way that any other method or thread can. Outputs are slightly more involved because each message pump has its own mechanism for integrating external events. The runtime therefore implements the notion of a 'GUI call-back' which is triggered by any circuit event and executes within the message pump in exactly the same way that a mouse or keyboard event would.



The circuit above illustrates how this works in practice and uses the example of the CDL editor's circuit translate function (somewhat simplified). The sequence starts with the GUI thread executing a native call-back in response to the 'translate' button being pressed. It disables the 'translate' button and then writes the filename of the application that requires translation to the remote translator circuit. The translator then executes asynchronously (scheduled across all the available worker threads in the colony). Each time an error is generated this triggers the error call-back which executes in the GUI thread allowing errors to be displayed on the screen. On completion, the translator writes a notification to the 'status' store. This triggers execution of the 'completion' call-back which executes back in the message pump. In this case it simply needs to re-enable the 'translate' button. The GUI thread can continue to execute other functionality throughout the translation operation.

4.12 Portability of the Runtime

It should be apparent from earlier discussion that the runtime is able to abstract most platform details from the application, but in order to provide application portability, it is necessary to implement the runtime itself on a broad range of targets. This section considers the portability of the runtime itself. The stack is actually very simple; the runtime contains most of the code, and calls down into the driver layer. In order to target a new platform, only the lower layer actually requires change. In order to run on a given platform, the application simply needs to link with the runtime and then the appropriate drivers. There are four categories of driver;

The first category deals with operating system abstraction and provides the runtime with a standard API that the driver implements from the target O/S. There are a few dozen entry points and these deal with issues like threading, determining the number of available cores, and other related functionality. At the time of writing the runtime has been ported to Microsoft Windows (including CE), most implementations of Linux, MacOS, VxWorks, Greenhill's Integrity, Solaris and Mercury.

Copyright © Connective Logic Systems Ltd 2009

The second category deals specifically with asymmetric shared memory platforms and includes mechanisms to map memory, translate addresses, move data and manage inter-process notifications. CLIP has been ported to Mercury and DY4 platforms.

The third category deals specifically with distributed memory (message based) platforms and includes mechanisms to connect, disconnect, multicast, send and receive. At the time of writing the drivers support any device with a Berkley socket API.

The fourth category deals with the standard GUI interface and the driver needs to execute a GUI call-back each time a CLIP event is sent to the GUI. At the time of writing there are drivers for MFC and Windows QT.

Copyright © Connective Logic Systems Ltd 2009

5 Benefits

This section summarizes the perceived benefits of the CLIP technology (connective logic and the CORE runtime).

5.1 Portability

There are a number of technologies that claim to provide portability, but in many cases this actually just means operating system level portability. The CLIP technology provides a means for ensuring that applications will execute repeatably on different architectures (symmetric, asymmetric and distributed memory) and crucially, will do so regardless of topology. This provides the additional benefit that applications can generally be developed and maintained as single processes, but deployed on any target platform without change; which has the knock-on benefit that hardware choice can be deferred until late in the program when CPU requirements have become more established.

For the reasons discussed in earlier sections of this article, generically re-mapping an application's functionality from 'N' machines to 'M' machines is a non-trivial problem; so much so that the UK MoD will not usually take delivery of mission critical real-time applications that utilize more than 50% of available CPU. This mitigates the almost inevitable problems that arise as functionality becomes more sophisticated (and cycle hungry) over time. Because CLIP applications are demonstrably scalable (CPU can be recruited at any stage to accelerate execution) this requirement was dropped for the UK's Surface Ship Torpedo Defense system (SSTD) and replaced by the less onerous requirement that there must be physical expansion slots available in the delivered platform.

5.2 Performance Issues

Although it must be true that application programmers operating with low level threading and socket APIs can at least match the performance of CLIP applications, in practice there are a number of optimizations that are simply not practical to implement within typical one-off application budgets. This is especially true in the case of distributed memory architectures (e.g. multiple networked machines) where major issues include; dynamic load-balancing, preemptive execution, task overlap, float-point data, smart scheduling (tasks sent to data rather than data sent to tasks) and shared data multicasting.

In fact for the reasons outlined in the earlier sections above it is actually quite difficult to write portable programs that don't gratuitously move data between threads in the same symmetric address space (very inefficient hardware utilization). It is also hard to develop programs that scale to the particular multi-core machine that they are executed on without proliferating unlimited numbers of threads and spending significant amounts of CPU context switching between them.

It is particularly difficult to write applications that minimize scheduling latency for the general case of irregular concurrency, especially when concurrencies reach four or more.

5.3 Holographic Processing Benefits

Holographic processing as defined by this article has a number of intrinsic benefits;

Firstly, the ability to dynamically recruit and retire CPU at any stage of the calculation and load balance across heterogeneous collections of machines with arbitrary numbers of differing speed cores means that applications can make best use of available hardware. It also means that migration to multi-core will not require continual adjustments (or worse still re-writes).

Machines can be retired from the colony at any stage of execution without loss of data and this has two benefits. Firstly, machines that form part of turn-key systems can be retired for routine maintenance, and secondly, any machines that are not in use during 'off-peak' periods can be recruited to perform batch calculations.

Network preemption means that when a high priority task becomes schedulable, all the machines in the network can be transiently recruited to its execution and this can significantly reduce turn-around times for latency sensitive tasks such as those in the financial and military sectors.

Finally, because all machines can provide all functionality there are less single points of failure.

Copyright © Connective Logic Systems Ltd 2009

5.4 Object Oriented Paradigm

One of the principal goals of the CLIP project was to give the user an object oriented programming paradigm, and in particular to provide a concurrent equivalent of a class. Although this is primarily an API issue, it does require the runtime to have certain properties. CDL provides the notion of a 'circuit' which has many of the properties of a concurrently executable class; aggregation, public and private state, multiple prototyped entry points, and a concept of inheritance.

Encapsulation is absolutely essential because without it, code can become dispersed and difficult to re-use (typically the case with conventional threading and messaging approaches). Another obvious benefit of encapsulating concurrency is that contiguous logic can be analyzed for potential deadlocks and races, and furthermore, this process can be automated.

When the concept of private state is combined with the runtime's collector object, locks only need to be exposed to programmers through access functions that ensure ordering, and crucially these sequences can be acquired asynchronously without having to block any threads of execution. In most cases deadlocks are completely eradicated.

Another property of classes (and hence circuits) is that their internal logic can be modified without impacting any external logic. This is particularly useful for parallel programs because it means that particular components can be optimized for specialized targets such as accelerator cards.

Perhaps most importantly of all however, because the basic operations that underpin CLIP are compose-able (and nest-able), so are circuits themselves. And this allows the development of libraries of re-usable components. A lot of the time, programming in CLIP is as simple as dragging and dropping pre-fabricated components and connecting them together to create a particular application. This extends the concept of re-use from sequential libraries to concurrent libraries, and the fact that a re-used component may actually distribute itself across the entire available network is completely transparent to the programmer who simply needs to connect the components inputs and outputs in accordance with its prototype.

Copyright © Connective Logic Systems Ltd 2009

6 Case Study

The UK's surface ship torpedo defense system was designed by a sonar expert with no programming experience whatsoever, but he was able to create the entire top level infrastructure for the system using CLIP. The result was that 40% of the deliverable code was generated from CDL diagrams, and although the original target was a PowerPC based asymmetric multi-processor running VxWorks, almost all development and debugging was carried out on standard Windows laptops. The system has been in service for several years (24/7) and has never experienced a deadlock or race of any kind. The 27 month development program was completed 6 months ahead of schedule.

Because it is portable and extendible and can be maintained on available desk-top hardware, SSTD has an extremely low cost of ownership and will shortly go into service with the Turkish and Australian Navy. It is also currently being trialed by the US Navy.

Copyright © Connective Logic Systems Ltd 2009