Communicating Instances Of The Flight Programming Language

Charles Eric LaForest

April 2006
University of Waterloo
Independent Studies

Abstract

Some minor alterations to the Gullwing virtual machine and to the Flight language kernel allow for multiple processor to share a common memory. These changes make it possible for an instance of Flight to compile code in memory and have it executed by another processor under software control. This capability is then used to connect via serial channels multiple instances of Flight running on such multiprocessors and use the self-extensible nature of Flight to remotely compile code and make the system appear as a single instance of Flight with parallelism.

Contents

List of Figures

1 Background

The underlying hardware upon which the Flight programming language executes is the Gullwing processor architecture [LaF05a]. It is a simple, second-generation stack architecture without support for interrupts or predefined Input/Output protocols, using instead memory-mapped, individually controllable I/O lines which are abstracted here as serial communication channels. It is currently implemented only as an emulator on UNIX platforms.

The Flight programming language itself is a stack-oriented, typeless, dynamic language which is also its own compiler [LaF05c]. The kernel of Flight is very spartan, containing only what is required to make it self-extensible, thus most software makes use of a library of code which extends Flight into a more tractable form. The code for these extensions is listed in Appendix A and is described in detail in [LaF05b].

1.1 Gullwing Pairs

Some trivial alterations to the Gullwing processor and the Flight kernel allow for the conversion of the original uniprocessor system into a pair which shares a common memory (Figure 1). Each processor has complete, independent and simultaneous access to memory, and each has a pair of unidirectional serial communication channels to the outside world.

At reset, the Data Stacks of each processor is loaded with an initial value: Processor A gets a zero, while B gets a nonzero value. When the reset is released, both processors will begin executing at location 0 in memory, which contains a conditional branch-on-zero instruction. By default, A will enter the `NXEC` scan-lookup-execute loop of Flight, while B will enter the `SLAVE_LOOP` function, which endlessly reads the address contained in `SLAVE_TASK` and calls to it. `SLAVE_TASK` is initially set to run the `NULL_TASK`, which executes some no-ops and is also used as a coordination semaphore between A and B.

The example hardware platform used is shown in Figure 2. It is simply a chain of processor pairs, with code input into pair P_A, and outputs visible at P_A and $P1_B$.

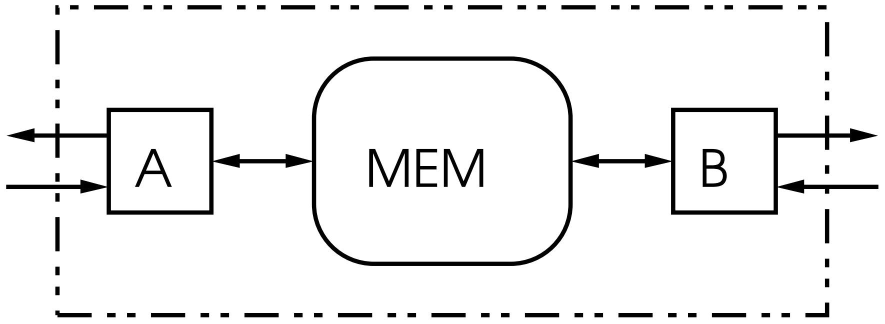

Figure 1: Pair Of Gullwing Processors

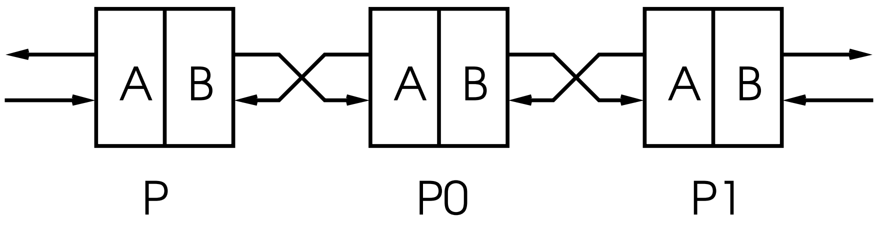

Figure 2: Chain Of Pairs

2 Shared-Memory Communication

Since processors A and B share the entirety of the memory, it is possible to divide a program between them. By this method features absent from hardware, such as interrupts and communication controllers, can be simulated (or at least, approximated) in software. This method is also the basis for serial channel communication, as shown in Section 3.

To distinguish between the processors and the tasks they run, whichever is running `NXEC` is known as the 'master', while the one which runs `SLAVE_LOOP` is know as the 'slave'. By default A is master and B is slave. It is possible to have two masters or two slaves, but these cases are not considered here.

2.1 Master/Slave Echo Demo

In this first demonstration, the master receives strings at P_A and instructs the slave to echo them at P_B. The slave reads from the same input buffer and executes the same fundamental code than the master would if it was echoing to P_A. All the source shown is received and compiled by the master.

The first function, `slave-wait`, simply sets the `SLAVE_TASK` to the address of the `NULL_TASK`.

```
: slave-wait
l# NULL_TASK l# SLAVE_TASK (!) ;
```

`write-task` calls the string output function, then reassigns itself to the null task so that it only runs once.

```
: write-task
c $>
j slave-wait
```

`slave-write` sets the `write-task` as the `SLAVE_TASK`.

```
: slave-write
l# write-task l# SLAVE_TASK (!) ;
```

This is simply an alias to make the code look consistent.

```
: master-read
j >$
```

The `NULL_TASK` is used as a semaphore. The master checks the address of the code that the slave is executing, and loops until the slave is idle.

```
: wait-for-slave
l# SLAVE_TASK (@) l# NULL_TASK (XOR)
if j wait-for-slave else ;
```

The master enters a loop of reading in a string, telling the slave to output it, and waiting for the slave to finish. If two alternating input buffers were used, the master and the slave could operate concurrently.

```
: run-through
c master-read
c slave-write
c wait-for-slave
j run-through
```

2.2 Master/Slave Task Demo

This second demonstration takes advantage of the fact that each processor simulation is a separate UNIX process to show the coordination between the master and the slave despite the non-deterministic run-time relationship between both.

```
: slave-wait
l# NULL_TASK l# SLAVE_TASK (!) ;
```

The next two functions output a counted string of length 1 containing the ASCII symbol '`A`' or '`B`'.

```
: A-task
n# 65 n# 1 c #> c #>
c slave-wait ;

: B-task
n# 66 n# 1 c #> c #>
c slave-wait ;

: slave-A
l# A-task l# SLAVE_TASK (!) ;

: slave-B
l# B-task l# SLAVE_TASK (!) ;
```

This is an instrumented version of `wait-for-slave` which outputs '`W`' while it is waiting, and '`D`' once done.

```
: wait-for-slave
l# SLAVE_TASK (@) l# NULL_TASK (XOR)
if
 n# 87 n# 1 c #> c #>
 j wait-for-slave
else
 n# 68 n# 1 c #> c #> ;

: run-through
c wait-for-slave
c slave-A
c wait-for-slave
c slave-B
j run-through ;
```

When `run-through` is called, the output from the master is

```
WDWWDWDWWDWDWDWDWWDWWWDWDWWDWWDWWWWDWWWDWW
```

and the output from the slave is

```
ABABABABABABABABABABABABABABABABABABABA
```

The masters ends up waiting for one to four loops in this example, while the slave executes in a fixed sequence.

2.3 Slave Control Library

This code is as before, except that a function to return the address of the task the slave is running at the moment has been added. It and the task-setting function now compile their code in-line for speed[1].

```
: (set-slave-task)
l# SLAVE_TASK c # c (!) ;

: (get-slave-task)
l# SLAVE_TASK c # c (@) ;

: slave-wait
l# NULL_TASK (set-slave-task) ;

: wait-for-slave
(get-slave-task) l# NULL_TASK (XOR)
if j wait-for-slave else ;
```

2.4 Slave Output Library

These two functions provide a way for the master to have the slave output a string from the input buffer.

```
: $>-task
c $> j slave-wait

: slave-$>
l# $>-task (set-slave-task)
j wait-for-slave
```

2.5 Master/Slave Swap Library

This set of functions allow the master and the slave to swap roles without race conditions. The first function, `(unwind)`, compiles in-place the body of `EXECUTE`, which 'unwinds the stack'[2] to forcibly return to its caller's caller. It is used here to remove from the stack the address of the function's caller, which is either the `NXEC` or the `SLAVE_LOOP` function, since it will never be returned to again[3]. If this was not done, successive master/slave role swaps would eventually overflow the stack.

```
: (unwind)
c (R>) c (DROP) ;
```

The slave simply sets itself idle, clears the return stack, and jumps to the Flight main loop, thus becoming the master.

```
: slave-become-master
c slave-wait
(unwind)
j NXEC
```

The master instructs the slave to change roles, waits for the slave to have begun by detecting it to be idle, clears the return stack, and jumps to the slave loop.

```
: master-become-slave
l# slave-become-master (set-slave-task)
c wait-for-slave
(unwind)
j SLAVE_LOOP
```

[1]It has become a convention in Flight to denote functions which compile their function instead of executing it by surrounding the name with parentheses. This is purely a naming convention.

[2]An expression borrowed from C programming. The stack is the Return Stack in this case.

[3]Actually, leaving the address in place could be useful for brief role swaps: the master and slave would then simply have to perform a function return at the top level to return to their previous roles. Some form of race condition avoidance would still be required though, else both processors may briefly end up in the same role and corrupt memory.

3 Serial Channel Communication

Because it is possible for the slave to output a string read in by the master, a bidirectional daisy-chain of pairs becomes the first network possible (Figure 2). Each pair runs an instance of Flight, therefore one instance can control another and thus distribute a program across pairs.

The software to accomplish this works by echoing a sequence of strings down the daisy-chain. It does so by modifying its own invocation and sending it to the next hop down the line, along with the carried message. At the hop just before the destination, the code removes its invocation from the stream and sends the raw carried message to the destination. This process can also be used to bootstrap the network by having the carried message be the source code of its carrier.

To keep the source listings manageable, previously defined and discussed source is replaced by a reference to its section, enclosed between angle brackets (`<...>`).

3.1 Send Library Base

This is the code which allows the library to call the instance of itself at the next hop. It simply sends out the required strings. The source works by `create`'ing a storage location, reading a string into it, then compiling code to place its address on the stack before executing the code following `does`, which in this case simply outputs the string.

The `)-string` function is a particular case. Since the '`)`' string is the delimiter for the carried message, it cannot occur in its body. This means that the Send Library, which tests for that string, must not contain it else it could not send itself! Therefore, instead of reading in the string at compilation, it is generated in place.

```
: send-n
create >$ n $>c
does j cs>

: send-send(
create >$ send( $>c
does j cs>

: )-string
create n 1 #>c ALIGN n 41 #>c ALIGN l HERE (@) #>c
does ;

: send-)
)-string # j cs>
```

3.2 Send Library

```
<Slave Control (Section 2.3)>

<Slave Output (Section 2.4)>
```

These function wrap the Send Library Base code into slave tasks for the master to initiate.

```
: send-n-task
c send-n j slave-wait

: slave-send-n
l# send-n-task (set-slave-task) j wait-for-slave

: send-send(-task
c send-send( j slave-wait

: slave-send-send(
l# send-send(-task (set-slave-task) j wait-for-slave


: send-)-task
c send-) j slave-wait

: slave-send-)
l# send-)-task (set-slave-task) j wait-for-slave
```

The `match)?` function compares the string just received with '`)`', and returns zero if there is a match. This is the test for the end of the carried message.

```
: match)?
l# INPUT (@) )-string # c COMPARE_STRINGS
(DROP) l# INPUT (@) c STRING_TAIL (XOR) ;
```

Read in a string. If the received string is not the delimiter, then echo it to the next hop and loop, else discard it.

```
: pass-until-)
c >$
c match)?
if
 c slave-$>
 j pass-until-)
else
 j POP_STRING
```

The number of hops to send the carried message is specified by an integer. This function decrements it by one and sends its string representation to the next hop.

```
: slave-send-next-count
(N-) 1 c #>$ j slave-$>
```

Finally, the main loop of the Send Library checks if the hop count is zero. If not, then send to the next hop the new hop count, the call to `send(`, the carried message, and the delimiter. If it is zero, then only send the carried message to the next hop.

```
: send(
(DUP)
if
 c slave-send-n
 c slave-send-next-count
 c slave-send-send(
 c pass-until-)
 j slave-send-)
else
(DROP) j pass-until-)
```

3.2.1 Demo

Assuming the chain depicted in Figure 2, where each pair initially only contains the Flight kernel, the following code will progressively bootstrap the network and allow code input at P_A to be run on P1, which will send its output to P_B. All code is input at P_A.

The first block compiles the Flight Extensions (Appendix A) and the Send Library on P.

```
<Flight Extensions From Appendix A>
<Send Library Base (Section 3.1)>
<Send Library (Section 3.2)>
```

The pair P can now forward messages to P1. The first messages sent are the same code block which was compiled on P.

```
n 0 send(
 <Flight Extensions From Appendix A>
 <Send Library Base (Section 3.1)>
 <Send Library (Section 3.2)>
)
```

Since P0 now has an instance of the Send Library, it is now possible to compile and execute code on P1.

```
n 1 send(
 <Flight Extensions From Appendix A>
 <Slave Control (Section 2.3)>
 <Slave Output (Section 2.4)>
)
```

Finally, P1 can now be remotely instructed to add 2 and 6 and send the string representation of the sum to P_B.

```
n 1 send( n 2 n 6 + #>$ slave-$> )
```

Here is how the previous command was rewritten as it travelled through the chain:

```
Input to $P_A$:         n 1 send( n 2 n 6 + #>$ slave-$> )
Output from $P_B$:      n 0 send( n 2 n 6 + #>$ slave-$> )

Input to $P0_A$:        n 0 send( n 2 n 6 + #>$ slave-$> )
Output from $P0_B$:     n 2 n 6 + #>$ slave-$>

Input to $P1_A$:        n 2 n 6 + #>$ slave-$>
Output from $P1_B$:     8
```

3.3 Return Library Base

This build upon the Send Library Base (Section 3.1), and adds the remote invocation for `return(`.

```
: send-return(
create >$ return( $>c
does j cs>
```

3.4 Return Library

```
: send-return(-task
c send-return( j slave-wait

: slave-send-return(
l# send-return(-task (set-slave-task) j wait-for-slave

<Master/Slave Swap (Section 2.5)>
```

This code functions identically to `send(`, but invokes `return(` instead and swaps the roles of the master and the slave after having sent the message. This *reverses* the direction of the network, allowing for data to be returned from a remote execution. Returning the data with `return(` restores the network to its original state.

```
: return(
(DUP)
if
 c slave-send-n
 c slave-send-next-count
 c slave-send-return(
 c pass-until-)
 c slave-send-)
 j master-become-slave
else
 (DROP)
 c pass-until-)
 j master-become-slave
```

3.4.1 Demo

```
<Flight Extensions From Appendix A>
<Send Library Base (Section 3.1)>
<Send Library (Section 3.2)>
<Return Library Base (Section 3.3)>
<Return Library (Section 3.4)>

n 0 send(
 <Flight Extensions From Appendix A>
 <Send Library Base (Section 3.1)>
 <Send Library (Section 3.2)>
 <Return Library Base (Section 3.3)>
 <Return Library (Section 3.4)>
)

n 1 send(
 <Flight Extensions From Appendix A>
 <Send Library Base (Section 3.1)>
 <Return Library Base (Section 3.3)>
 <Slave Control (Section 2.3)>
 <Slave Output (Section 2.4)>
)
```

This command will compile code on P1, execute it, and as a flourish cause it to erase itself afterwards when `forget` reads in `marker` and erases it and all code that comes after. The `send-message-back` function constructs the reply.

```
n 1 return(
 : marker ;

 : output-4-at-slave
 n# 2 n# 2 (+) c #>$ j slave-$>

 : output-5-at-master
 n# 2 n# 3 (+) j #$>

 : send-message-back
 c send-n
 n# 1 c #$>
 c send-return(
 c output-5-at-master
 j send-)

 : return-test
 c output-4-at-slave
 c send-message-back
 j forget

 return-test marker
)
```

Here is the flow of messages through the chain, starting with the last command sent:

```
Input to $P_A$:         n 1 return( : marker ;  return-test marker )
Output from $P_B$:      n 0 return( : marker ;  return-test marker )
(P swaps roles)

Input to $P0_A$:        n 1 return( : marker ;  return-test marker )
Output from $P0_B$:     : marker ;  return-test marker
```

```
(P0 swaps roles)

Input to P1_A:          : marker ;  return-test marker
Output from P1_B:       4
Output from P1_A:       n 1 return( 5 )
(P1 forgets code from marker onwards)

Input to P0_B:          n 1 return( 5 )
Output from P0_A:       n 0 return( 5 )
(P0 swaps roles)

Input to P_B:           n 0 return( 5 )
Output from P_A:        5
(P swaps roles)
```

The chain has now output `4` at the far end and `5` at the near end, from code input at P and executed at P1.

4 Multiple Flight Instances Demo

With the Send and Return Libraries, a local function can now execute code on and receive results from other processors. Multiple remote calls might be dispatched and the return values, if any, read back at a later time. This can make one instance of Flight into an intelligent interface for other instances, which all appear as a single large instance where the local code is executed sequentially, but with internal parallelism. This may be recursively true for the underlying instances of Flight also.

This section will demonstrate the division of the input, processing, and output functions of a simple stream cipher program across the chain shown in Figure 2, effectively pipelining its operation to operate all three parts at once while appearing local.

4.1 RANDU-based String Cipher

This function multiplies two 8-bit integers, returning a 16-bit product.

```
: 8x8
(>R)
(2*) (2*) (2*) (2*)
(2*) (2*) (2*) (2*)
(R>)
(+*) (2/) (+*) (2/) (+*) (2/) (+*) (2/)
(+*) (2/) (+*) (2/) (+*) (2/) (+*) (2/)
(>R) (DROP) (R>) ;
```

This is an unoptimized version of the RANDU linear congruential pseudorandom number generator (PRNG), reduced to 8 bits. The recurrence relation is $x_{i+1} = \left[\left(2^{n-2}+3\right)x_i\right]\ mod\, 2^{n-1}$ where $n = 8$ and x_0 is odd[4]. This is possibly the worst PRNG in existence, but its implementation is trivial.

```
: rand
n# 67 c 8x8 n# 128 c U/ (DROP) ;

: seed n 1 var ;
```

This takes the seed, which is x_i, computes the exclusive-or of it and an ASCII character in memory, and stores the new value back in the same location. It then computes a new seed value from the old one.

```
: process
(>R) seed # (@) (R>)
(@) (OVER) (XOR) (A!)
c rand seed # (!) ;
```

The `process` function is then used to build a version that iterates over strings.

[4]Actually, the first power of two should be $2^{\frac{n}{2}}$, but I goofed. It still works well enough.

```
: cipher
n 1 mapgen process
```

4.1.1 Demo

First the initial seed is set.

```
n 77 seed !
```

Then a string is read into the input buffer.

```
>$ testword
```

The address of the input buffer is obtained, and its content printed out, followed by a horizontal tab. The `DUP`plicate instruction creates a copy of the address on the stack for the next function.

```
l INPUT @
DUP cs> \t
```

The string is enciphered and printed, followed by a horizontal tab.

```
DUP cipher
DUP cs> \t
```

Applying the cipher, once the seed is reset to the same value, returns the string to clear text.

```
n 77 seed !
cipher
$> \n
```

The output is:

```
testword    9BF+*87k    testword
```

4.2 Load1

This first segment programs the first pair, P. It implements the input interface in a manner similar to the Send Library (Section 3.2) but since it will not be used to process itself, the delimiter workaround is not required and it is present in the source. The purpose of these functions is to initialize the cipher and watch for the message delimiter, '`EOL`', when forwarding messages.

```
<Flight Extensions From Appendix A>
<Send Library Base (Section 3.1)>
<Send Library (Section 3.2)>

: slave-send-set-seed
create >$ set-seed $>c
does c c>$ j slave-$>

: set-seed
c >$
c slave-send-set-seed
j slave-$>

: slave-send-encrypt
create >$ encrypt $>c
does c c>$ j slave-$>

: EOL-string
create >$ EOL $>c
does ;
```

```
: matchEOL?
l# INPUT (@) EOL-string # c COMPARE_STRINGS
(DROP) l# INPUT (@) c STRING_TAIL (XOR) ;

: encrypt
c >$
c matchEOL?
if
 c slave-send-encrypt
 c slave-$>
 j encrypt
else
 j POP_STRING
```

Here are some example inputs and outputs:

```
Input to PA:        set-seed 56
Output from PB:     set-seed 56

Input to PA:        encrypt This is a plaintext message EOL
Output from PB:     encrypt This encrypt is encrypt a encrypt plaintext encrypt message
```

4.3 Load2

This segment programs P0. It accepts a command and a string from P, encrypts the string, and forwards it to P2. It also sets the seed.

```
n 0 send(

 <Flight Extensions From Appendix A>
 <Send Library Base (Section 3.1)>
 <Send Library (Section 3.2)>
 <RANDU-based String Cipher (Section 4.1)>

 : set-seed
 c n
 seed # (!) ;

 : slave-send-write-out
 create >$ write-out $>c
 does c c>$ j slave-$>

 : encrypt
 c >$
 l# INPUT (@)
 c cipher
 c slave-send-write-out
 j slave-$>

)
```

Here are some example inputs and outputs:

```
Input to P0A:        set-seed 56
Output from P0B:     (no output, sets the seed to 56)

Input to P0A:        encrypt This encrypt is encrypt a encrypt plaintext encrypt message
Output from P0B:     write-out <ciphertext This> write-out <ciphertext is>
```

4.4 Load3

This segment programs P1. It receives the ciphertext from P0 and outputs its numerical representation.

```
n 1 send(

 <Flight Extensions From Appendix A>
 <Slave Control (Section 2.3)>
```

See the end of Appendix A for the source to `print$#`, which prints a string as numbers.

```
 : print$#-task
 l# INPUT (@) c print$# j slave-wait

 : slave-print$#
 l# print$#-task (set-slave-task) j wait-for-slave

 : write-out
 c >$
 j slave-print$#

)
```

Here are some example inputs and outputs:

```
Input to P1_A:       write-out <ciphertext This> write-out <ciphertext is>
Output from P1_B:    34 12 6 123 43 2 31 87 09
```

4.5 Demo

The input at P_A is:

```
set-seed 77
encrypt
This is a test sentence. EOL

set-seed 75
encrypt
This is a test sentence. EOL

set-seed 77
encrypt
This is a test sentence. EOL
```

The output, split into three lines for clarity, at $P1_B$ is:

```
25 79 92 44 52 36 36 123 8 116 33 76 24 89 17 10 99 4 16 49

31 73 74 90 82 2 114 13 78 50 119 58 126 127 7 124 101 2 6 71

25 79 92 44 52 36 36 123 8 116 33 76 24 89 17 10 99 4 16 49
```

5 Further Work

5.1 Nesting

It is possible in theory to nest `send(` and `return(` calls, which would allow more sophisticated network messaging (messages that send/receive messages), but in practise this would require the presence of the ')' delimiter in the body of the message. A possible solution would be to have the first string in the message be the current delimiter. Also, there is sometimes a need to only compile a call to those function for later use. The `send-message-back` function (Section 3.4.1) does handles both the delimiter and compilation problems by using the Send and Return Libraries to construct the required reply function, but a more elegant mechanism is needed.

5.2 Topology

The example network given in Figure 2 is merely linear, but if the A and B processors are provided with extra serial channels then other topologies become possible. The Send and Return Libraries would then need to be improved to replace the single hop count with a better coordinate system, but this is not a fundamental change. For example, if the pairs are arranged in a n-cube mesh then the hop count becomes multiple integers, one for each dimension, but otherwise functioning in the same manner by subtracting from the count at each hop. A more sophisticated algorithm would avoid the obvious shortest path if it was congested, incrementing the hop count when not moving the message directly towards the destination [Hil86, 7.2].

5.3 Congestion

If a message is sent via the `return(` function, the path taken by the message is unavailable for other messages until the destination provides a reply in the same manner. If the reply delay is long, and if the path happens to intersect another one, then multiple parts of the distributed program may stall. Worse yet, if another `return(` reply takes an intersecting path, at least one hop in the first reversed path will flip back to its previous direction, cleaving the path in two, preventing the reply from reaching its destination, and effectively corrupting the program which is spread over the network.

A solution to this problem would be to split the message transactions. The sender uses `send(` for the original message, and again at a later time to ask the receiver if the result is ready. Alternately, the receiver could notify the sender in the same manner. Finally, the sender uses `return(` to ask for the result, which is ready and returned immediately.

5.4 Efficiency

Sending remote commands by emulating string input is a simple, but grossly inefficient method. Not only must the string be sent and received symbol by symbol, but it must then be looked up in the dictionary to find the address of the code it refers to. This can easily add up to thousands of instructions executed before any useful (and often, short) code is run.

The solution lies in the fact that the code address is fixed and already known by the time the remote code can be called. The sender first sends a message to compile the code remotely, then sends another to request the address at which it was compiled. Once all the required addresses are known to the sender, it sends a message to the receiver telling it to execute a different main loop than `NXEC` (Section 1.1), which only reads in an integer and then calls to it. The sender now only needs to send a single symbol, an integer, to remotely call code. The receiver now only needs to call to that address and avoids the entire address lookup overhead. The message passing process remains otherwise unchanged, since it is the called code that will read in any arguments sent, and not the main loop.

Also, the length of a message is know once it has been read in, so the use of delimiters or repeated calls as in Section 4.3 is unnecessary. Instead, a recursive data structure, a string of strings, would reduce the delimiter comparisons to a simple count. This is not a trivial change to implement, since all the I/O and dictionary functions will needs to be changed to use the new structures.

5.5 Concurrency

In the simple examples given, the coordination between the master and the slave (Section 2.3) always leaves one of them idle and waiting for the other to finish. This is to prevent them from simultaneously operating on the same buffer and thus corrupting its contents. If separate buffers are used then the `wait-for-slave` function can be non-blocking, and simply return a value based on whether or not the slave is running the null task. This would double the throughput of the distributed cipher program in Section 4.

A Flight Extensions

```
SCAN : DEFN SCAN SCAN LOOK CMPCALL SCAN DEFN LOOK CMPCALL CMPRET
: l SCAN SCAN LOOK CMPCALL SCAN LOOK LOOK CMPCALL CMPRET
: $c l LOOK CMPCALL l CMPCALL CMPCALL CMPRET
: c SCAN SCAN $c SCAN $c $c CMPRET
: n c SCAN c NUMI CMPRET
: alias c l c SCAN c DEFN_AS CMPRET

alias   CMPJMP          (JMP)
alias   CMPJMPZERO      (JMP0)
alias   CMPJMPPLUS      (JMP+)
alias   CMPCALL          (CALL)
alias   CMPRET           ;
alias   NUMC             #
alias   NUMI             $n
alias   LOOK             $l
alias   DEFN             $:
alias   SCAN             >$
alias   WRITE1           #>
alias   READ1            >#
alias   TENSTAR          10*

: $j c $l c (JMP) ;
: j c >$ c $j ;

: $j0 c $l c (JMP0) ;
: j0 c >$ c $j0 ;

: $j+ c $l c (JMP+) ;
: j+ c >$ c $j+ ;

alias   CMPFETCHA        (A@)
alias   CMPSTOREA        (A!)
alias   CMPFETCHAPLUS    (A@+)
alias   CMPSTOREAPLUS    (A!+)
alias   CMPFETCHRPLUS    (R@+)
alias   CMPSTORERPLUS    (R!+)
alias   CMPXOR           (XOR)
alias   CMPAND           (AND)
alias   CMPNOT           (NOT)
alias   CMPTWOSTAR       (2*)
alias   CMPTWOSLASH      (2/)
alias   CMPPLUS          (+)
alias   CMPPLUSSTAR      (+*)
alias   CMPDUP           (DUP)
alias   CMPDROP          (DROP)
alias   CMPOVER          (OVER)
alias   CMPTOR           (>R)
alias   CMPRFROM         (R>)
alias   CMPTOA           (>A)
alias   CMPAFROM         (A>)
alias   CMPNOP           (NOP)

: @ (>A) (A@) ;
: ! (>A) (A!) ;
: XOR (XOR) ;
: AND (AND) ;
: NOT (NOT) ;
: OR (OVER) (NOT) (AND) (XOR) ;
: 2* (2*) ;
: 2/ (2/) ;
: + (+) ;
: +* (+*) ;
: DUP (DUP) ;
: DROP (DROP) ;
: OVER (OVER) ;
: NOP (NOP) ;
: #+ c # c (+) ;
: n# c n c # ;
: l# c l c # ;
: (@) c (>A) c (A@) ;
: (!) c (>A) c (A!) ;
: (OR) c (OVER) c (NOT) c (AND) c (XOR) ;

: \a n# 7 n# 1 c #> j #>
: \b n# 8 n# 1 c #> j #>
: \t n# 9 n# 1 c #> j #>
: \n n# 10 n# 1 c #> j #>
: \v n# 11 n# 1 c #> j #>
: \f n# 12 n# 1 c #> j #>
: \r n# 13 n# 1 c #> j #>
: \s n# 32 n# 1 c #> j #>

: negate (NOT) n# 1 (+) ;
: (negate) c (NOT) n# 1 c #+ ;
: -n c n (negate) ;
: -$n c $n (negate) ;
: -n# c -n c # ;
: (N-) c -n c #+ ;
: (N+) c n c #+ ;
: - (negate) (+) ;
: (-) c (negate) c (+) ;

: if n# 0 c (JMP0) l# HERE_NEXT (@) (N-) 1 ;
: if- n# 0 c (JMP+) l# HERE_NEXT (@) (N-) 1 ;
: else (>R) c NEW_WORD (R>) (>A) (A!) ;

: max (OVER) (OVER) (-) if- (>R) (DROP) (R>) ; else (DROP) ;
: min (OVER) (OVER) (-) if- (DROP) ; else (>R) (DROP) (R>) ;
: abs (DUP) if- (negate) ; else ;

: allot (DUP) if c ALIGN (N-) 1 j allot else (DROP) ;

: $copy
(>R) (>A)
(A@) (A>) (+) (N+) 1
: $copy-loop (A>) (OVER) (XOR)
if (A@+) (R!+) j $copy-loop
else (DROP)
(R>) (DUP) (>A) (A!) ;

: $>c
l# INPUT (@)
l# HERE (@)
(OVER) (@) (N+) 1 c allot
c $copy
c ALIGN
j POP_STRING

: c>$
(DUP) (@) c PUSH_STRING
l# INPUT (@)
c $copy ;
```

```
: cs>
(DUP) (@) (+) (N+) 1
(A>) (>R)
: cs>-loop (R>) (OVER) (OVER) (XOR)
if (>R) (R@+) c #> j cs>-loop
else (DROP) (DROP) ;

: $>
l# INPUT (@)
c cs>
j POP_STRING

: $print c l c c>$ j $>
: csprint c l j cs>

: match?
l# INPUT (@) c STRING_TAIL (XOR) ;

: end?
l NAME_END @ # (XOR) ;

: erase
(DUP) l# THERE (!)
(DUP) (N+) 1 l# INPUT (!)
(@) (DUP)
(N-) 1 l# HERE (!)
l# HERE_NEXT (!)
j ALIGN

: forget
c >$
l# THERE (@) (N+) 1 (DUP)
: forget-loop
l# INPUT (@)
c COMPARE_STRINGS
c match?
if (DROP) c STRING_TAIL (N+) 1 (DUP) c end?
 if (DUP) j forget-loop
 else (DROP) c POP_STRING ;
else c erase (DROP) ;

: #>c l# HERE (@) (!) ;

: <=
(-) (DUP)
if- (DROP) -n# 1 ;
else
 if n# 0 ;
 else -n# 1 ;

: >=
(-) (DUP)
if- (DROP) -n# 0 ;
else
 if n# 0 ;
 else -n# 1 ;
```

```
: 15x15
(>R)
(2*) (2*) (2*) (2*)
(2*) (2*) (2*) (2*)
(2*) (2*) (2*) (2*)
(2*) (2*) (2*)
(R>)
(+*) (2/) (+*) (2/) (+*) (2/) (+*) (2/)
(+*) (2/) (+*) (2/) (+*) (2/) (+*) (2/)
(+*) (2/) (+*) (2/) (+*) (2/) (+*) (2/)
(+*) (2/) (+*) (2/) (+*) (2/) (+*)
(>R) (DROP) (R>) ;

: divby0msg >$ DIV_BY_0_ERROR $>c
: divby0 l# divby0msg c cs> ;
: errcontext c \s c >$ c $> j errcontext
: divby0check (DUP) if ; else (DROP) (DROP) c divby0 j errcontext

: U/
c divby0check
n# 0 (>A)
: U/-loop
(OVER) (>R) (DUP) (R>) c <=
if
(DUP) (>R) (-) (R>)
(A>) (N+) 1 (>A)
j U/-loop
else
(DROP) (A>) ;

: minus?
if- n# 1 ; else n# 0 ;

: numstrlen
c abs (DUP) (N-) 10
if- (DROP) n# 1 ; else (DUP) (N-) 100
if- (DROP) n# 2 ; else (DUP) (N-) 1000
if- (DROP) n# 3 ; else (DUP) (N-) 10000
if- (DROP) n# 4 ; else (DUP) (N-) 100000
if- (DROP) n# 5 ; else (DUP) (N-) 1000000
if- (DROP) n# 6 ; else (DUP) (N-) 10000000
if- (DROP) n# 7 ; else (DUP) (N-) 100000000
if- (DROP) n# 8 ; else (DUP) (N-) 1000000000
if- (DROP) n# 9 ; else (DROP) n# 10 ;

: #>$
(DUP) c numstrlen (OVER) c minus? (+) c PUSH_STRING
(DUP) if- n# 45 l# INPUT (@) (N+) 1 (!) c abs else
(>R) l# INPUT (@) c STRING_TAIL (N-) 1 (R>)
: #>$-loop
n# 10 c U/ (>R) n# 48 (+) (OVER) (!) (N-) 1 (R>)
(DUP) if j #>$-loop else (DROP) (DROP) ;

: #$> c #>$ c $> ;

: create
n# 0 c # l# HERE_NEXT (@) (N-) 1 (>R)
```

```
n# 0 c (JMP) l# HERE (@) (N-) 1
l# HERE (@) (R>) (!) ;

alias else does

: var c create (>R) (N-) 1 c allot (R>) c does ;

: name-as c >$ c DEFN_AS ;

: mapgen
c (DUP) c (>R) l# STRING_TAIL c (CALL) c (R>)
n 1 # c # c (+)
c NEW_WORD
c (DUP) c l c (CALL) (>R) c # (R>) c (+)
c (OVER) c (OVER) c (XOR)
c if (>R) c (JMP) (R>)
c else c (DROP) c (DROP) c ; ;

: print# (@) c #$> c \t ;
: print$# n 1 mapgen print#
```

References

[Hil86] W. Daniel Hillis, *The connection machine*, MIT Press, Cambridge, MA, USA, 1986.

[LaF05a] Eric LaForest, *Unit 2-3 (IS 102B) report: The Gullwing stack computer architecture*, IS Unit report, University of Waterloo, Independent Studies Program, April 2005, Otherwise unpublished.

[LaF05b] ______, *Unit 2-3 (IS 301B) report: Extensions to the Flight programming language*, IS Unit report, University of Waterloo, Independent Studies Program, November 2005, Otherwise unpublished.

[LaF05c] ______, *Unit 4-5 (IS 101B) report: The Flight programming language*, IS Unit report, University of Waterloo, Independent Studies Program, April 2005, Otherwise unpublished.