University of Bristol

DEPARTMENT OF COMPUTER SCIENCE

Hardware Migratable Channels

David May   Henk Muller   Shondip Sen

March 2000 | CSTR-00-005

Hardware Migratable Channels

David May Henk Muller Shondip Sen
Department of Computer Science, University of Bristol, UK.
`http://www.cs.bris.ac.uk/`

Abstract

In this paper we revisit the hardware implementation of channels. Channels as an essential part of the processors instruction set were first launched with the Transputer. Here we have revisited them and have made two major alterations: we have changed the semantics of the input and output instructions in order to overlap communication, and we allow channels to be communicated over channels (higher order communications). We show that all of those operations can be easily implemented in hardware.

1 Introduction

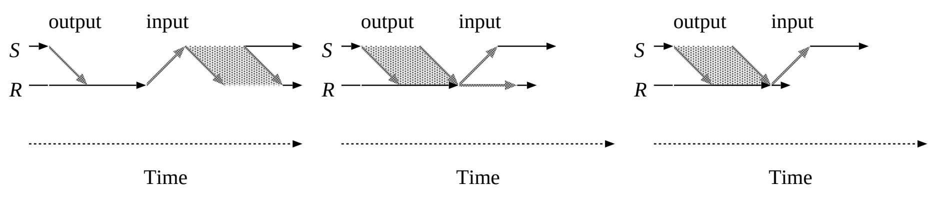
Communication is at the heart of concurrent systems. It is well known that in order for concurrent systems to work efficiently, we need efficient yet flexible communication primitives. In this paper we focus on a hardware implemented communication primitive (such as found in the Transputer family). We have changed the semantics of the hardware channels in two ways:

- Instead of fixed communication channels, we allow channel ends to migrate. The migration capability has been implemented in such a way that normal communication does not suffer a performance penalty.
- We have solved the buffer management issue by allowing (requiring) the compiler to define where messages received on a channel are to be buffered. This is specified on a per packet basis.

The rest of this paper is organised as follows. We start by giving a brief introduction of the specification of the input and output instructions in Section 2. This includes a description of how input data is buffered. The protocol used to communicate data and ports is presented in Section 3. The hardware implementation is detailed in Section 4.

2 Input and Output instructions

The traditional interface for input and output on on Occam style channels consists of two operations: `input` and `output` [1]. The input operation inputs a number of bytes on a channel on a given address, the output operation outputs a number of bytes on a given channel. In the classic implementation, the output operation blocks until the input operation is executed, whereupon the data is transferred between the two processes and both processes are released. This implements a synchronous transfer in hardware.



(a) Ask for permission first (b) Presend data (c) Presend and Preallocate

Figure 1: Progress of two processes S and R. Black arrows indicate computation, grey arrows data transport.

Although these semantics are very elegant, and useful for compilers and humans to reason about, they are not the most efficient semantics for implementing channels in hardware. In particular, with these semantics it is difficult to hide latency by pre-sending data on a channel. Suppose that we have two processes S and R that are connected by means of a (relatively) high latency channel, and that we wish to communicate data from S to R. An unoptimised communication protocol would operate as follows:

1. the sender will request the input process for permission to send data
2. the receiver will acknowledge that the input process is ready
3. the sender will transfer data and continue.

As sketched in Figure 1(a), the sender has to wait for a double latency before it can actually transfer the data.

An optimisation is to pre-send the data from the sender to the receiver, as shown in Figure 1(b). In this case the double latency can overlap with some of the computation. However, the common implementation of this will cause the receiver to have to copy the data from a buffer into its final destination using the following scheme:

1. the sender will send the input process the data as soon as it can.
2. the receiver will store the data in a temporary buffer if the receiving process is not ready to input yet.
3. when the receiving process is executing the input instruction, the receiver will acknowledge that the input process is ready, and copy the data.
4. the sender will continue once it receives the acknowledgement.

We will show that we can avoid copying altogether by redefining the input primitive so that it will store data in a *compiler defined* pre-allocated buffer. (Note that this is different from run time allocated buffers, such as used in Mach [2], or the use of scatter-buffers as used in Solaris [3].)

2.1 Input

We propose that we have an architecture where each port has an associated memory location where the data is going to be stored (the column labelled address in Figure 5), with an input primitive with the following syntax and semantics:

```
input port-register,address-register
```

This primitive will wait for the data to appear on the port and then perform the following actions

- It will swap the address-register and the current buffer location of the port.
- It will send an acknowledgement to the outputting process to signal that the I/O operation has succeeded.

Note that this instruction does not specify where the data of *this* input operation is to be stored, but it specifies where the data of the *next* input operation is to be stored. Typically, the compiler knows where the data will be needed, so it can store it directly in the right memory location. In the worst case, the compiler will have to use two global buffers to store the data alternatingly. This is no slower than the original copying scheme.

What makes this input operation unique is that it opens the door to implement many compiler optimisations:

- A loop reading data and processing it can be unrolled once. Renaming the variable where the data is stored will enable us to store the data of alternating iterations to be stored in alternating memory buffers. In pseudo code:

  ```
  int a, i ;
  channel in_channel, out_channel ;
  for( i=0 ; i<100 ; i++ ) {
    input( in_channel, &a, 1 ) ;
    /* Process a */
    output( out_channel, &a, 1 ) ;
  }
  ```

  Is translated into

  ```
  int a, b, i ;
  channel in_channel=&a, out_channel ;
  for( i=0 ; i<100 ; i+=2 ) {
    input( in_channel, &b, 1 ) ; /* Reads into a! */
    /* Process a */
    output( out_channel, &a, 1 ) ;
    input( in_channel, &a, 1 ) ; /* Reads into b! */
    /* Process b */
    output( out_channel, &b, 1 ) ;
  }
  ```

- A loop reading data in an array can trivially predict where the next data item is to be stored:

  ```
  int a[100], i ;
  channel in_channel, out_channel ;
  for( i=0 ; i<100 ; i++ ) {
    input( in_channel, &a[i], 1 ) ;
  }
  ```

Is translated into

```
int a[100], i ;
channel in_channel=&a[0], out_channel ;
for( i=0 ; i<100 ; i++ ) {
  input( in_channel, &a[i+1], 1 ) ; /* Reads into a[i]! */
}
```

Each port is initialised with an initial memory buffer where the first data item that is going to be read will be stored. Note that input is still strictly synchronous, and does not become asynchronous, such as is the case in, for example, the aioread call in Solaris [3]. We just separate synchronisation and data transfer, much like splitting a load instruction in a `prefetch` and `load`.

2.2 Output

In addition, we can split the output instruction into two parts, a pre-send instruction and a synchronisation instruction. The pre-send instruction actually transfers the data. This operation and the associated compiler optimisations are presented in a companion-paper [4] and [5].

3 Protocol

A port can be in one of 7 states: EMPTY (port is not part of a channel), IDLE (this port is part of a channel, but it is at present not used for input or output), INPUTTING (a process is attempting to input data on this port), OUTPUTTING (a process is attempting to output data on this port), BUFFERFULL (the companion port is attempting to output data on this port, the data has been sent speculatively), MIGRATING (the port is in the process of being moved to another node), and CONFIRMING (a port has been read from this channel, but the process cannot go ahead until the port has been wired up completely). The following are invariants for the states:

EMPTY No process has a reference to this port. If the state is not empty, then exactly one process has a reference to this port, exactly one other port in the system, c, has this port, t, as its companion port, and the companion port of t is c.

IDLE No process is performing an input or output on this port at present.

INPUTTING Exactly one process is performing an input operation on this port.

OUTPUTTING Exactly one process is performing an output operation on this port. The data has already been pre-send to the companion port.

BUFFERFULL Data has been received on this port from the companion port, but no process is performing an input on this port.

MIGRATING This port is attempting to migrate to another node. No input or output can be performed on this port (for it is being migrated). One other copy of this port may be around on the node where this port was sent to. The companion port

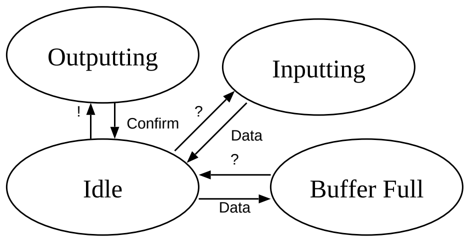


Figure 2: State machine for ordinary IO

will be informed of the migration, when informed, it will resent any pre-send data.

CONFIRMING The port has been used to read data from. The associated process cannot restart until the inputted port has been completely wired up.

The complete state machine is shown in Figure 3.

3.1 Data over Ports

The `outputdata` instruction will send the data, but will not affect the state of the port; only when the `output` operation is performed will the port move to the state OUTPUTTING.

When inputting data, the port moves to state BUFFERFULL when the data is received, when the input instruction is performed, the port will move back to IDLE. Alternatively, if the input instruction is performed before (or while) the data is transmitted, the port will first go to the state INPUTTING, and back to IDLE when the data is transferred. The partial state machine just incorporating the input and output operations is shown in Figure 2.

3.2 Ports over Ports

In order not to fix the communication graph, we need the ability to communicate ports. We allow ports to be treated as first class objects, similar to the pi-calculus [6]. However, because our channels are point to point, we can move them relatively easily. In a previous study we have implemented a moving channel end as a entity of its own [7]; regardless over the medium it was moved over. Although this gave us a good primitive to work with, embedding it in a hardware environment proved non trivial.

We have now designed a new protocol for moving virtual channels over a network of virtual channels that is suitable for a hardware implementation, such as the one proposed for MIDAS [8]. This protocol prevents chains of synchronisations and chains of sent data building up. At most two steps are to be performed on any state transition, enabling a trivial implementation using microcode or an FSM.

We assume that we have a situation as sketched in Figure 3.2. In this figure we see three nodes with one process on each node, and two channels connecting those three processes. A process is an entity that the programmer sees; the programmer uses channels to exchange messages between processes. Below that level, nodes are the

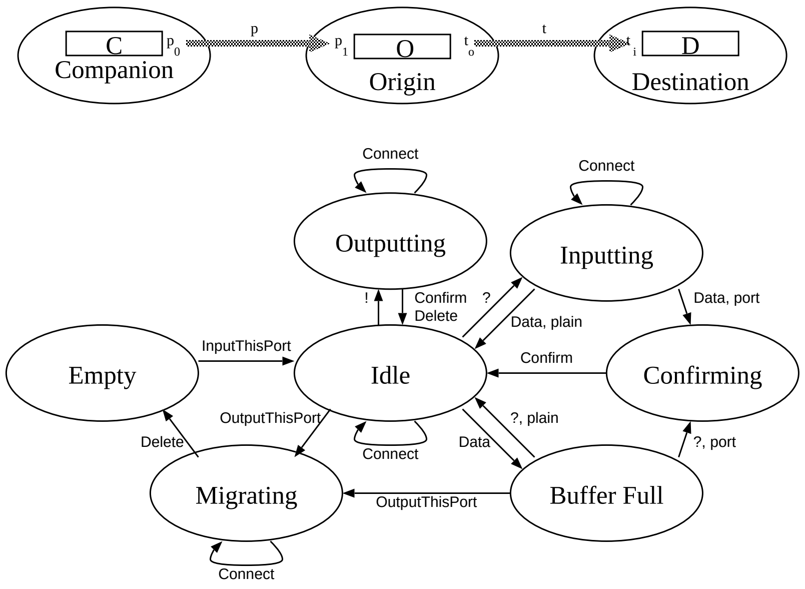


Figure 3: Complete state machine

physical processors that run processes. Nodes implement the protocol to communicate data between processes.

The nodes are Origin, Companion and Destination, and the respective processes are named ‘O’, ‘D’ and ‘C’. The port to be transferred is p_1 and resides on the origin node, it is connected via a channel p to a port p_0 on the companion node. The ports p_1 and p_0 are the input and output port of p; the protocol is symmetrical; either of p_0 and p_1 can be the input and the output port.

The channel that we use to transfer the data over is called t, and connects processes O and D on the origin and destination nodes. Channel t consists of ports t_i and t_o, for input and output respectively. All communication is synchronous, and all channels are guaranteed to be point to point. This implies that if a port is moving, no process can communicate over that port. Also, when a channel is used to transfer a port, the sending end of the channel cannot migrate, for it is locked up in an output operation.

3.2.1 Stage 1: prepare for move

The first stage starts when process O running on the origin node will attempt to output p_1 over t_o. This output will initiate three actions:

1. The state of the port to be moved, p_1 is set to the state MIGRATING. Because the port is being sent, it is impossible that someone wants to input or output on the port, hence p_1 must have been either IDLE, or BUFFERFULL. If p_1 was in the state BUFFERFULL, then that signalled that data was sent to this port; this data is simply ignored. At a later stage of the protocol, p_0 will be required to retransmit.

2. Port t_o, the port over which p_1 is migrating, is stored in the address field of the port-table at the origin. This allows a route for the confirmation message to be sent at a later stage.

3. A data message is sent over t consisting of the companion port of p_1, i.e. the address of p_0.

The system then stays in this state until a process on destination node actually inputs on t_i. It is essential that the protocol will not proceed. Proceeding would commit the port to be moved to this destination node, while it is not yet known whether a process on this node is going to read this port. It is not impossible that instead the port t_o itself will migrate before the data will be read.

3.2.2 Stage 2: input of the port

Stage 2 commences when the port transferred over t is inputted by process D on the destination node. This will initiate three actions:

1. A new port is allocated in the port-table. The state of the port is set to IDLE, and the companion-id is set to the data read from t_i. The index of the new port is stored on the stack of the process (ready to be used).

2. A message is sent to the companion node requesting port p_0 to be wired up to the newly allocated port. This message consists of the companion-id, and the newly allocated port-index.

3. port t_i is set to CONFIRMING to signal that the port has been read and is being wired up.

3.2.3 Stage 3: wiring up, and cleaning up

The third stage of the protocol starts when the companion node receives the request to wire-up. The following actions are performed:

1. the companion-id of port p_1 is overwritten with the newly received port-id received from the destination node. This will effectively complete the channel p between C and D.

2. If the state of the port was not IDLE, then some additional operation may have to be performed.

   - If the state was OUTPUTTING, then the companion node has to resent data to the new destination, for the original data has been thrown away. (It may be in transit to the origin node, in which case it will be thrown away on the origin).
   - If the state was MIGRATING, then there is trouble ahead: it indicates that both ports were migrating simultaneously. This case is dealt with in Section 3.2.5
   - If the state was not MIGRATING, then a Delete message is sent to the origin node, where the original port resides, indicating that the original port is to be deleted, and that the process sending this port can be restarted.

3.2.4 Stage 4: Final confirmation

When the origin node receives a Delete message for port p_1, it will delete the port from the port table (freeing this entry up for future use), restart the process that was waiting for this port found via t_o) and simultaneously send a confirmation message to the destination node, confirming that this port has been migrated successfully. This confirmation process will restart the process on t_i.

3.2.5 Stage 5: ports where both ends migrate

It is possible that both ends of a port migrate simultaneously. If that is the case, then both ports will always find the companion in a state "MIGRATING". The solution is simple: we forward the wiring up message to the new destination node of the companion node, where two cases may arise:

1. That node has not yet inputted its data, in this case we simply overwrite the data waiting on the port.
2. That node has already inputted its data, and a port has been allocated as a companion node. In that case the newly allocated port has been stored in the process structure of D, so we can update the companion-id of this port.

In either case we also send the Delete messages out to the origin node, which will cause a confirmation to be forwarded to the destination node, as in stage 4.

3.2.6 Deadlock freedom, progress

We do not (yet) have a formal proof that the protocol is deadlock free, but we can intuitively see why it is free. As far as network deadlock is concerned, each node will generate at most one message on acceptance of a message. If the network can always accept a message once a message has been delivered, then all messages will always be delivered.

The protocol itself is deadlock free because there is only one situation where the protocol can block: that is in stage 2, when the protocol waits for a process to input on t_i. This can only cause a deadlock if the program transferring ports deadlocks.

3.2.7 Incorrect optimisations

A couple of optimisations are tempting, yet incorrect. First, it is tempting to forget about the Confirmation message. The Conformation message is needed explicitly to prevent the destination from running away before the protocol has been completed. Without it, the end point of a channel could run away to a fifth node before it was wired up. This can be resolved with a (messy) forwarding mechanism.

Second, it is tempting to send the Confirmation message from the companion node, and not via the origin node. This is not sufficient, for it may take over the migration of another node. All messages are sent in a circle in our case (Figure 4), ensuring that all operations of the protocol are performed in-order (assuming that our interconnect guarantees in-order delivery).

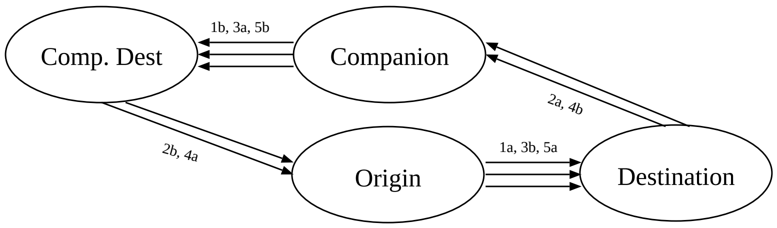


Figure 4: Two nodes sending ports away simultaneously; the numbers refer to the stages, the ’a’ and ’b’ refer to the two nodes initiating the transfers.

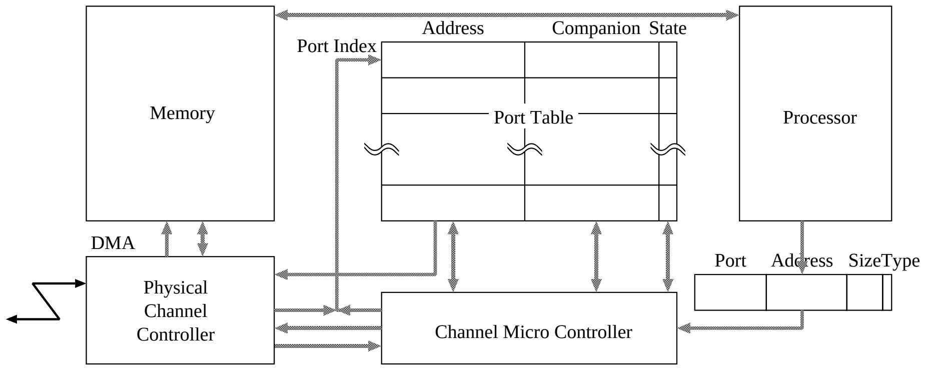


Figure 5: Implementation of port table

.

4 Implementation

We want to embed the channels at a low level in our processor, much like the transputer and its successor, but then integrating dynamically moving protocols in the hardware. Each node (processor) has a port-table. A port is represented by an index in the port-table. Each entry in the port-table consists of the port’s state and two extra fields:

companion-id An 32-bit identifier representing the companion port. A channel is made up of exactly two ports, so each port has one companion port. This is represented by a 32 bit identifier; 16 bits to identify the node and 16 bits to identify the port in the node. Note that in a stable situation a port’s companion companion-port will always refer to itself.

address A pointer in memory that is either storing the buffer where to input data (if the state is equal to IDLE); or it may hold the port index where this data has been transferred over (if the state is equal to MIGRATING).

The port table is controlled by a micro-controller as shown in Figure 5.

We except the channel controller to be very small indeed (the size of our software implementation indicates the simplicity). The main overhead is the port-table. Each port carries the absolute minimum amount of state needed: the remote port identifier and a pointer where to store the data. Still, if we want to implement a processor with room for 256 ports, then the port table will occupy an area equivalent to 2 Kbytes of static memory.

Note that the number of ports is fixed on each processor. However, thanks to the fact that ports can migrate, one can always migrate some of the software to another processor if more ports are to be employed.

5 Conclusions

In this paper we revisit the hardware implementation of channels. We have defined a protocol for transporting data (consisting of either ordinary bits or ports) over channels. The protocol speculatively pre-sends data to the receiver, where it is stored in a buffer. The instruction to input data must define where the next bit of data is to be stored. This allows data transport to overlap with computation, while retaining fully synchronous communication. Using this instruction the compiler can implement a zero-copying protocol without dynamic memory management.

If the input port is moved before the data is actually needed then the data will be resent. The protocol to transport a port from one node to the next performs the same speculative pre-send, but only when the data is actually accepted will the port finally move. This results in a protocol which can be implemented trivially in hardware. Any message coming in will result in a state transition and at most one message being generated. No chain of synchronisations or data are building up.

References

[1] INMOS. *The Transputer Databook*, November 1988.

[2] A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison Wesley, 1997.

[3] *Solaris Reference Manual*. SUN Microsystems, 1998.

[4] D. Towner and D. May. Optimising Concurrent Software Using Split Communication Transformations. Technical Report CSTR-00-LR (submitted for publication), Department of Computer Science, University of Bristol, Jan. 2000.

[5] M. Goldsmith. The Oxford Occam Transformation system, 1988.

[6] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, Sept. 1992.

[7] H. L. Muller and D. May. A Simple Protocol to Communicate Channels over Channels. In *EURO-PAR '98 Parallel Processing, LNCS 1470*, pp 591–600, Southampton, UK, September 1998. Springer Verlag.

[8] R. Kirk and A. Hunt. MIDAS–MILAN An Open Distributed Processing System for Audio Signal Processing. *Journal Audio Engineering Society*, 44(3):119–129, Mar. 1996.