Hardware Migratable Channels*

David May Henk Muller Shondip Sen

Department of Computer Science, University of Bristol, UK.
http://www.cs.bris.ac.uk/

Abstract. Channels as an essential part of a processor's instruction set were first launched with the Transputer. We have made two major alterations: the semantics of the input and output instruction are changed in order to overlap communication, and channels are allowed to be communicated over channels (higher order communications). All 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. First, instead of fixed communication channels, we allow channel ends to migrate. Second, we have solved the buffer management issue by requiring the compiler to define where each received message is to be buffered. This paper describes the instructions and protocol, full details including the implementation are in [1].

2 Compiler Directed Input Buffers

The traditional interface for communication over Occam style channels consists of two operations: `input` and `output` [2]. 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.

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, it is difficult to hide latency. In a naive implementation the sender would first ask the receiver for permission to send the data, after granting permission, the data would be transferred, incurring a quadruple latency.

* This work was partially funded by Hewlett Packard Research Laboratories Bristol, UK

Solutions solving the latency issue often copy the data; however, we can avoid both excessive latency and copying 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 [3], or the use of scatter-buffers as used in Solaris [4].) We propose an architecture where each port has an associated memory location where the data is going to be stored, 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: swap the address-register and the current buffer location of the port; and send an acknowledgement to the outputting process to signal that the I/O operation has succeeded.

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 specify the right memory location. In the worst case, the compiler will have to use two global buffers to store the data alternately. 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. For example, a loop reading data into an array can be transformed as follows:

```
int a[100], i ;                     int a[100], i ;
channel in_c, out_c ;               channel in_c=&a[0], out_c ;
for( i=0 ; i<100 ; i++ ) {   ⇒ for( i=0 ; i<100 ; i++ ) {
  input( in_c, &a[i], 1 ) ;           inputNEW( in_c, &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. Similarly, a loop reading data and processing it can be unrolled once. Note that the `input` operation is still strictly synchronous (unlike, for example, the aioread call in Solaris [4]). We separate synchronisation and data transfer, much like splitting a load instruction in a `prefetch` and `load`. A split output operation (presend + synchronise) is described in [5] and [6].

3 Communicating Ports over Ports

By allowing ports to be sent over ports, we are able to create a communication graph that is no longer fixed. This is accomplished by treating ports as first class objects, similar to the pi-calculus [7]. However, because our channels are synchronous and point to point, they can be moved relatively easily. In a previous study, a relocatable channel end or port was described as an entity of its own [8]; regardless of the medium it was moved over. Although this provided a good primitive to work with, embedding it in a hardware environment proved non trivial. We have now designed a new protocol for moving ports over a network of

virtual channels that is suitable for a hardware implementation, such as the one proposed for MIDAS [9]. 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.

Each port consists of an entry in a port-table and can be in one of seven states as shown in the state transition diagram, Figure 1(a). The following invariants hold. **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-sent 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 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.

3.1 Protocol

As a simple example, a migrating port is illustrated in Figure 1(b), with three nodes: origin, destination and companion with processes O, D and C on each node respectively. Two channels p and t connect the processes OC and OD. Our objective is to relocate the port p_1 from the environment of process O to D via the transport channel t (the port labels t_i and t_o stand for input and output), the end result being a channel p which connects C to D.

Stage 1: prepare for move When process O attempts to output p_1 over a port, t_0 the channel migration operation will be initiated: **(1a)** The state of port p_1 is set to MIGRATING from either IDLE or BUFFERFULL. This obstructs any other process at the origin from trying to communicate using this port. If the port was previously in the BUFFERFULL state, any buffered messages remain unacknowledged and are retransmitted later to the destination process D. **(1b)** 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 creates a route for the confirmation message to be sent at a later stage. **(1c)** 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 remains in this state until the process where t_i resides commits to input the port p_1. Ownership of t_i may change as it may itself migrate.

Stage 2: input of the port Stage 2 commences when the port transferred over t is inputted by process D on the destination node: **(2a)** A new port is allocated in the port-table. The state of the port is set to IDLE, and the companion-id is

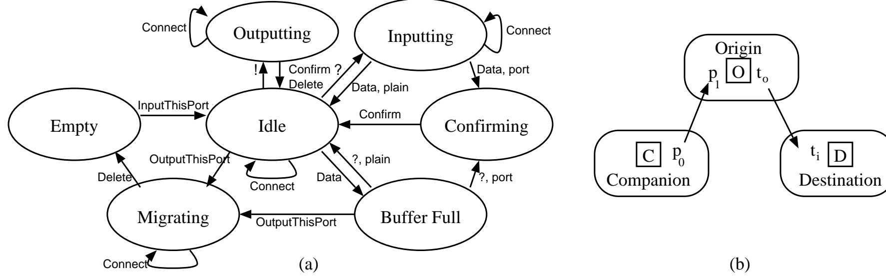

Fig. 1. (a) Complete state machine, (b) Nodes, processes and ports

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). **(2b)** 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. **(2c)** port t_i is set to CONFIRMING to signal that the port has been read and is being wired up.

Stage 3: wiring up, cleaning up and confirmation The third stage of the protocol starts when the companion node receives the request to wire-up. **(3a)** 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. **(3b)** If the state of the companion port was OUTPUTTING, then the unacknowledged message must be re-transmitted (see stage 1 part 1a). If the state is MIGRATING, then both ports of channel p are migrating simultaneously (this situation is discussed in the next paragraph). If the companion is in any state other than MIGRATING then a delete message is sent to the origin node causing deletion of the port-table entry of p_1 and notification for the sending process to proceed. Simultaneously a confirmation message is sent from the origin to the destination node notifying it that the migration process has been completed and that the process owning port t_i may be restarted.

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. Finally we also send the Delete message out to the origin node, which will cause a confirmation to be forwarded to the destination node, as in stage 3.

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. 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 only deadlocks if the program transferring ports deadlocks.

4 Conclusions

In this paper 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 defines where the next message 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. An on-chip implementation will have a fixed number of ports for each processor with an area overhead of around 2Kb of static memory for 256 ports. 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.

References

[1] D. May, H. Muller, and S. Sen. Hardware Migratable Channels. Technical Report CSTR-00-005, Department of Computer Science, University of Bristol, March 2000.

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

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

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

[5] 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.

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

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

[8] 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.

[9] 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.