An occam style communications system for UNIX networks

Kevin J. Vella

Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent CT2 7NF, U.K.
E-mail: kv4@ukc.ac.uk
Tel: ++44 1227 764000
Fax: ++44 1227 762811

Dept. of Computer Science & A.I.
University of Malta
Msida MSD06, Malta
E-mail: kvel@cs.unimt.mt
Tel: ++356 3290 2505
Fax: ++356 320539

December 9, 1995

Abstract

This document describes the design of a communications system which provides occam style communications primitives under a UNIX environment, using TCP/IP protocols, and any number of other protocols deemed suitable as underlying transport layers. The system will integrate with a low overhead scheduler/kernel without incurring significant costs to the execution of processes within the run time environment. A survey of relevant occam and occam3 features and related research is followed by a look at the UNIX and TCP/IP facilities which determine our working constraints, and a description of the T9000 transputer's Virtual Channel Processor, which was instrumental in our formulation. Drawing from the information presented here, a design for the communications system is subsequently proposed. Finally, a preliminary investigation of methods for lightweight access control to shared resources in an environment which does not provide support for critical sections, semaphores, or busy waiting, is made. This is presented with relevance to mutual exclusion problems which arise within the proposed design. Future directions for the evolution of this project are discussed in conclusion.

1 Scenario

An environment for executing parallel programs expressed in a safe, pure parallel language on a wide variety of architectures is a desirable tool. Various parallel programming environments exist in the form of libraries used in conjunction with a sequential language such as C or FORTRAN. Parallel Virtual Machine (PVM) [24] and Message Passing Interface (MPI) [18, 23] are two instances of this approach, the latter only specifying a standard programming interface, abstracting away from implementation decisions. Such combinations, however, make both formal and intuitive reasoning about a parallel program as a single entity rather difficult, since the semantics of parallelism are not part and parcel of the language, but stuck on as an afterthought in such a manner that communicating processes cannot readily be thought of as part of the *same* program (as described in [68]). Additionally, the coarse grain parallelism imposed as the basis of many such libraries limits the amount of readily identifiable parallelism at fine grain algorithmic level which can be exploited on suitable architectural platforms, and used as extra parallel slackness [62] to hide latency in other situations.

On the other hand, the occam language [34, 9] offers parallelism as an integral part of the language, and a stable formal basis for reasoning about parallel algorithms [54, 55, 11], based on CSP [28], or related process algebrae [50]. The fine level of granularity expressible through this programming model exposes sufficient parallelism for exploitation both by parallel execution, and for latency hiding [6, 4]. Sadly, a practical execution environment which also meets the requirements just laid down has not yet been implemented for occam, or indeed, any language possessing the same desirable properties.

2 Aims and Requirements

The aim of this project is the construction of a software *communications system* which efficiently implements the occam model of communication across interconnected computers, principally running UNIX [53], though not exclusively so, on top of TCP/IP [21], while including sufficient flexibility to add on support for whatever communications protocol is deemed suitable as an underlying transport layer. This scenario hints strongly at ethernet-connected workstation networks, which are in fact the initial target for this system, though no strings are attached to this particular case in the design.

Implementing a *run time system* (process scheduler and internal communications kernel) for occam programs as part of the project immediately binds the system to a particular occam compiler and single processor run time system. This is undesirable, as it limits the applicability of our system, so a strict dividing line is drawn as an interface between the single processor run time system, and the communications system. Our scope is limited to enabling the run time system to perform external communications, without interfering with the way in which processes are scheduled, internal communications are implemented, and memory is laid out. The only alterations needed to adapt an existing uniprocessor runtime system are:

- the addition of a simple comparison in the kernel routines for communications, by which the channel is identified as internal (so that the normal course of action is taken) or external (in which case the communication system is invoked accordingly), and
- the provision of a routine which is called by the communication system when an external communication is completed, so that the appropriate process is rescheduled as desired by the run time system.

Necessary for complete system operation are a *network description language* [38], and a corresponding *software configuration language* [39] for mapping programs onto distributed machines. One can envisage the extension of this system to heterogenous processor architectures, and heterogenous interconnects, which would necessitate the inclusion of relevant directives in the network description language regarding processor types, and underlying network protocols for which communications drivers have been included. While the former (heterogenous processor architectures) can be set aside for the time being, the case for support for heterogenous links is strong, as machines on the network can be interconnected in various ways (such as TCP/IP over practically anything, and raw devices like serial ports), and various connection types can be offered even over a single link type (for example, a TCP/IP link can be offered as a TCP stream, a UDP packet multiplexing service, or a packet multiplexing service built *over* TCP). In this vein, ATM networks [64, 52, 30] introduce a wide range of parameters to characterise a connection, which specify properties such as quality of service and guaranteed bandwidth. It is desirable to expose these in the network description language so that the quality and capacity of individual links can be decided by the author of the description, possibly on the basis of the anticipated loading on each link.

The software configuration language, together with software for loading executable code to processors on the network, depend heavily on the particular compiler and run time system. However, it is expected to be possible to provide a partial solution to this problem, which may be extended in a modular fashion to match particular systems' requirements by defining an interface similar to that between run time system and communication system, through which system specific extensions for loading and configuring code can be included.

It is a fundamental requirement that the interface offered to the run time system presents a direct means of implementing the occam channel communication primitives `?` and `!`, for normal communication under `PAR`, as guards in nondeterministic choice `ALT`, and as shared resources in the occam3 `CALL` channel and `SHARED` channel types. Interruptions of the occam computational engine must be shorn to the minimum possible, the internals of the communication system executing asynchronously with respect to the former. Optimisations pertaining to particular underlying communications protocols can be conveniently isolated in the protocol's driver.

3 occam and occam3

The occam language defines three primitive processes, analogous to statements in a sequential imperative language: assignment, input and output. Both parallel and sequential composition are presented as first class constructs to be used at the same level of granularity. The parallel construct `PAR` contains an implicit barrier synchronisation between its components at their termination. Message passing is through point to point, synchronous channels connecting pairs of processes. Consequently, communication is deterministic, since contention does not occur for normal occam channels. Strict usage rules which can be statically checked are enforced for variables and channels in parallel processes, excluding the possibility of any form of contention, and consequently nondeterminism which could be casually overlooked by programmers.

occam programmers can willingly introduce nondeterminism using the `ALT` choice construct [27, 33, 56, 31], which creates contention on a process on the recieving end of multiple channels, used as input guards, rather than on a channel. Only a weak form of fairness is guaranteed, so starvation can be observed which is peculiar to the particular occam implementation. By allowing the introduction of nondeterminism only explicitly, groups of processes can be guaranteed to be safe from nondeterminism, and the points which act as a source of nondeterministic execution can be pinpointed statically, thus allowing the partitioning of a program into deterministic groups of processes with nondeterminism at the boundaries. Priority choice can be expressed using the `PRI ALT` construct [8].

The proposed occam3 definition [9] augments the language with the introduction of shared channel resources [66], shifting the focus of contention, and consequently, the source of nondeterminism, to the channel. This time, a strong notion of fairness is guaranteed in the first in, first out management of access to the channel resource. One instance of this new concept is in the `SHARED` channel type, where any number of parallel client processes `CLAIM` the channel in question, and the single server process at the other end `GRANT`s exclusive channel access to the client whose request was first registered at the server end. The other instance of resource sharing in occam3 manifests itself in the `CALL` declaration, which provides a high level client-server construct, analogous to RPC (Remote Procedure Call). A process, with a single entry point, is explicitly and permanently bound to the server end of an implied '`CALL` channel', which acts as a medium for sending parameters and receiving results. Client processes effect calls to the server and are serviced one at a time, in a first in, first out manner.

3.1 Mapping programs onto parallel architectures

occam does not provide directives for placement of processes and channels on machine architectures, since this would tie down program code to that particular machine. Rather, a separate *configuration language* [39], defines the mapping. The configuration language needs a machine description, which is specified in a separate language known as the *network description language* [38]. This arrangement assumes the machine to be a message passing, distributed memory architecture, but one could well imagine a similar configuration language using a shared memory machine specification generated by a corresponding description language. With the development of appropriate tools which model channel loading and process execution patterns, optimising the mapping of critical paths in the process graph [25], the mapping could be automated to obtain configurations with reasonable performance. Another possible solution, for applications with dynamic execution patterns, is the use of a run time load balancing system [29, 32, 3]. However, it must be noted that with recent developments in communications technology [42, 63, 4], the importance of locality in communicating processes has diminished. Moving a step further, in a scenario where a processor farm can service processes from a single, shared pool without major performance penalties, the mapping problem is dissolved away [69]. However, such an scheme on a shared memory machine with a cache hierarchy may induce thrashing on cache pages unless scheduling hierarchies echoing the cache hierarchies are established to keep cache footprints rather small and static (i.e. processes should be scheduled in such a way so that the same set of processes has a greater affinity for the processor most recently used, thus keeping multiple copies of the same process code in different caches to a minimum). [15] discusses the problem in the context of the extremely fine grain parallelism in dataflow architectures. In retrospect, with the introduction of scheduling hierarchies, this becomes reminiscent of the situation where processes are dynamically mapped to processors and migrate periodically, only that the migration process is supported by hardware.

3.2 Processor utilisation and performance

Traditionally, synchronous communication is considered to be inefficient, in the sense that communications latency is left exposed and consequently, the participating processes are suspended. Even worse, the party that arrives first at the synchronisation point has to wait for the other party to catch up, at which point the communication can start. On the other hand, asynchronous communication hides such latency by allowing the participating processes to proceed in the meantime, thus decreasing the amount of time for which the process waits, though only to the extent allowed by the underlying algorithm. However, the behaviour of asynchronous programs can be counterintuitive, in that an obscure but possible scheduling of events can fill up a queue (now necessary for managing communications), resulting in either program failure or a return to synchronous communications until the queue is emptied. The latter can result in obscure and unpredictable deadlock, which cannot be reasoned about from the the language's semantics alone (that is, without knowledge of the particular run time implementation's queue sizes).

Alternatively, when multiple parallel processes coexist at each locality, synchronous communication can hide arbitrary amounts of latency just as easily, without mandating alterations to programs, assuming enough parallelism is available. The processor time during which one process is participating in a synchronous communication can be used to execute another process. Assuming enough processes are available to fill in the gaps introduced by synchronous communication, and that they do not perform communication themselves too frequently, all gaps can be covered. Though *per*-process waiting time (or 'virtual processor' idle time) is not decreased, processor idle time is eliminated, and consequently, processor utilisation is maximised on all components of the parallel machine. Thus multithreading can be seen as an alternative form of latency tolerance to asynchronous communications, extensive caching, or instruction level parallelism.

The number of processes mapped to a single processor measures the degree of *parallel slackness* [62] available. The ratio of communication to computation in a process is known as the *grain size* of that process. A well thought out combination of the program's average grain size (this is also subject to the mapping), the mapping's parallel slackness, the network's latency, and the machine's minimum tolerable grain size can maximise machine utilisation. Keeping latency and machine grain size constant and as low as possible, programs can be written to match or exceed the machine's grain size, and mapped onto a variety of like- or finer-grain architectures with varying latencies by changing the mapping to obtain the required degree of parallel slackness [4, 6].

This still does not solve the mapping problem, since the decision as to *which* processes map best onto a particular processor is still open. Two opposing trends are both popular: either ignore all locality concerns during mapping, map out processes randomly, and guarantee a performance derived from the worst case; or carefully place processes which communicate frequently on adjacent processors. The former is seen as a step in the direction of general purpose parallel computing [42] as algorithmic structure can be separated from network topology. However, for effective results the hardware must qualify as a general purpose parallel computer [63, 7] by guaranteeing certain performance (computation *and* communication) and scalability characteristics. Regarding data locality, the occam programming model clearly specifies the data which should be placed on the same processor as a process.

3.3 Performance prediction of occam programs

It is difficult to predict the performance of unrestricted communicating process programs on real parallel computers. While it is possible to obtain best, worst, and average case measures for occam programs combinatorially through the underlying traces (interleaving or non-interleaving), these do not take into account contention for network links, and other overheads present in real execution on parallel computers. Moreover, even if this were satisfactory, it would be computationally unfeasible for all but tiny programs, unless substantial state space reduction is performed. This approach has been considered in recent work [16, 26, 10], modelling the underlying network operation stochastically using Stochastic Timed Petri Nets, which are then converted to Markov chains.

The method mentioned in the previous section [4] for maximising processor utilisation in communicating process programs assumes an upper limit for values of latency and machine granularity. One can imagine that an extension of this simplified model can be used to loosely predict worst case performance.

Alternatively, restricting the model of parallel computation can simplify the task of performance prediction. This is the thesis underlying Valiant's BSP [62, 44, 43, 12, 45, 46], which presents a simple but realisable model of parallel computation as a generalisation of the idealised PRAM [22]. This work is reminiscent of von Neumann's sequential model, in that it can also act as a bridge between software design, so a variety of parallel architectures can all be considered as BSP machines with differing parameters. A simple equation can predict the performance of a BSP program on a BSP machine. The BSP model of computation stipulates that communicating parallel processes synchronise at barriers, periodically. The computational portion is known as a superstep, during which processes may send or receive data asynchronously or perform local computation. However, the outcomes of communications performed during a particular superstep are only visible at the start of the subsequent superstep. The barrier enforces a 'dead' period during which all outstanding communications pertaining to the previous superstep are completed.

At least two attempts of extending occam with barrier synchronisation constructs for BSP computing have been made [65, 2]. However, it has also been noted [69] that occam performs barrier synchronisation implicitly at the end of a `PAR` construct. Moreover, between subsequent `PAR`s, variables whose scope extend across the whole sequence of `PAR`s are reassigned exclusively to individual components of the `PAR`s every time a PAR ends and another one commences. This instance of 'shared' memory, combined with the implicit barrier synchronisation, induces one to view a SEQ of PARs as a BSP program respecting occam usage rules, with the BSP communications being effected through the exchange of exclusive variable ownership. No alterations to the occam language definition need be made. However, current implementations disallow the distribution across processors of all but PARs at the topmost level, so that the single resultant barrier synchronisation can be ignored without serious consequences. Granted, distributing lower level PARs across processors would imply a barrier synchronisation across a part of the network, during which implicit message passing to exchange the updated values of the 'shared' variables has to be made. This can weaken the real time aspect of occam. On the other hand, one can argue that such techniques can be avoided by an aware programmer in critical sections (though this would stain occam's transparency), and, more convincingly, that an upperbound on the delay can be fixed for a particular architecture. Communications to exchange the values can be inserted by the compiler as early as possible before the actual barrier in the code, by statically analysing processes' dependence graphs.

The benefits to be gained from the use of a SEQ of PARs within the restrictions specified by BSP in the computational kernel of an application are mainly due to the simplicity of performance prediction on a variety of architectures. Program segments outside the computational kernel can be designed with liberal amounts of communication, as required. It would be interesting to investigate to which extent it is possible to have parallel processes communicating liberally within single BSP processes, without breaking the BSP requirements.

It has been argued that satisfactory degrees of portability can be obtained from less restrictive models, such as that described in the previous section, and the LogP model [14, 1]. One wonders whether this would leave BSP with its straightforward performance prediction as its only unique virtue.

4 UNIX and TCP/IP networking

This section contains an overview of lightweight and heavyweight process scheduling, interprocess communication techniques, and networking facilities under the UNIX operating system. While occam process scheduling is beyond our scope, it is instructive to survey the techniques used by various run time systems in existance. Moreover, we have already specified it as our aim to detach our communications process from the occam execution engine to as great an extent as possible, and UNIX may provide a suitable way to implement this, with its various process management and interprocess communication facilities. The communications process will ultimately translate communication requests to UNIX communications calls, through the driver which implements the particular link, be it through TCP/IP, a raw driver for a physical device, another communication package such as PVM, or any other medium.

4.1 UNIX process management

UNIX processes are designed for multiprogramming support with little interprocess communication, at a coarse grain. Process creation and context switches are very expensive, since a very large, generic state has to be maintained for each process. Full protection is provided between process memory spaces. Various interprocess communication facilities are provided, which all carry substantial overhead in suspending processes and communicating. These will be discussed in the next section. Process creation using the `fork` system call which replicates the current process image, or by loading a new executable from the file system using one of the `exec` family of system calls, are both prohibitively expensive. In general, UNIX processes are considered inadequate for application level parallelism, except at a very coarse grain, as in PVM.

Induced by the drive towards application multithreading, which is in part due to the event driven nature of GUI based applications, many UNIX vendors now provide a 'lightweight' process library, usually compliant with the POSIX Threads standard. Typically, these libraries work within the context of a single UNIX process, thus escaping the weight of saving and restoring a UNIX process' state. Consequently, context switch times are in the order of a few tens of microseconds on current architectures (this is still too heavy for occam processes). The memory space belonging to the UNIX process is shared between all lightweight processes (or *threads*), and the lightweight scheduler works in user mode. Alternative means of synchronising lightweight processes have to be provided, since normal interprocess communications mechanisms act on the whole of a UNIX process. Also, a lightweight process making a blocking call will suspend the whole UNIX process, and thus all other lightweight processes within it. More recently, UNIX schedulers have been implemented which integrate normal (heavyweight) process scheduling with lightweight process scheduling (and possibly real time process scheduling) under one scheduling policy, solving the above limitations while maintaining their original motivating properties. The context switch time remains too expensive for occam's fine grain parallelism.

As all the above facilities for multithreading do not satisfy occam's granularity requirements, implementors turned to implementing their own schedulers within the environment of a single UNIX process [13, 51, 17]. occam process state is minimal, bringing down the cost of a context switch down to under one microsecond on current architectures, if the scheduling state is permanently kept in selected machine registers [51]. Obviously, this ties down the code generated by a compiler to comply to a particular run time system in restrictions on register usage, analogous to the hardware providing special support for multithreading. Various other techniques to keep context switch time low within a UNIX process can be exploited, such as switching between multiple stacks within a process, using the `setjmp` system call. Placing the onus of scheduling on the occam processes themselves by inserting switching instructions at prespecified descheduling points in the code, besides being in itself efficient, has the advantage of eliminating the shared resource access synchronisation problem, since processes will never be interrupted in critical sections. In such frameworks, communication between occam processes, and occam process creation both benefit from low cost, in the order of microseconds.

Notwithstanding these efforts to reduce overheads, ulterior measures have to be taken to ensure minimal interruptions to the UNIX process which contains the occam processes, in the event of external communication, both locally (for console input/output), and remotely (through external network links). Unnecessary interruptions to the occam computational engine will eat away from the available scheduled timeslices for that UNIX process, thus it is desirable to avoid performing system calls from within that process. Though nonblocking versions of many calls exist, the overhead for polling for completion, is still taken from that process' timeslice. It may be advantageous to separate all forms of communication into a separate UNIX process from the one performing occam computations, in such a way that communication requests from the occam computational process is relayed to this communication system process without consuming much time from the occam computational process. The communication system process performs all system calls; every single communication request may involve several system calls and several interruptions which are avoided from the occam computational process, but suffered by the communication system.

It can be argued that since both these processes will typically share a single processor between them, the extra delays suffered by the communication process through performing system calls (both blocking and nonblocking) will still be propagated to the occam computational process. Ultimately, it is a unique, shared processor resource which is being held up to perform these costly operations, and consequently will be unavailable to the occam computational process anyway. A counterargument to this counts on the fact that out of the total amount of time assigned by the UNIX scheduler to the occam computational process, a higher percentage is being used to perform useful computation. The communication process absorbs all other work within its own timeslice. Integrating both these tasks into one UNIX process could mean that less overall processor time would be shared between both tasks, though extra switching overheads are avoided. In a tightly coupled multiprocessor UNIX machine, it is expected that this arrangement demonstrates its advantages more convincingly. Another compelling motivation for splitting up the two arises from our desire to decouple the communications system from any particular run time system implementation, and avoid recompilation of the communications system with the occam program and vice versa.

4.2 Interprocess communication mechanisms

In the design of the communications system a low cost means of coordinating with the occam computational engine, which will be hosted in separate UNIX processes, is desirable. This brief survey of interprocess communication methods is included with the intension of identifying a suitable alternative for this purpose. A detailed description of IPC mechanisms may be found in [59]; the Solaris manuals [49] were also referred to.

Modern System V UNIX interprocess communications (IPC) mechanisms fall into three categories. Each created IPC resource has a unique identifier, and is managed through an interface consistent across all three categories. The IPC resource categories are:

Messages. A message resource consists of a message queue to which a process may append messages (ASCIIZ strings) and a message type identifier (which can be used to receive messages selectively). The maximum message size depends on the machine's memory page size. The maximum number of active message queues and the maximum number of messages in a message queue are two limits imposed on the use of message passing; however, the latter can be increased dynamically by the superuser.

Semaphores. Semaphore resources are allocated in arrays of semaphores, each of which has a non-negative integer value, and on which operations to read, increment and decrement are defined through the `semop` call. A decrement which would result in a negative semaphore value suspends the whole UNIX process until the eventual change in situation. Further such processes are queued, to be reenabled in the original order of suspension. The maximum number of semaphores in an array is limited by processor page size and the maximum number of semaphore resources at any one time is a system constant. Semaphores can be used to synchronise access to shared memory by different processes, thus emulating message passing through shared memory.

Shared memory. The `shmget` call accepts a key, which can either identify an existing shared memory segment, in which case the memory is attached to the calling process' address space; or it can be an unused key, in which case a new memory segment is allocated and attached to the process' address space. Thus multiple UNIX processes can read and write a single memory segment in an uncoordinated manner. The shared memory segment can be made to appear anywhere in the process' address space. Various methods may be used to control access to parts of the memory.

For the purposes of this discussion, decidedly coarser grain communication and sychronisation methods such as lock files and pipes will not be considered. One finer grain means of interprocess communication, with a particularly long history, was not originally intended for such applications:

Signals. Every process can issue signals of a variety of types to any other process in the system, provided the user identifier is common, or the sender has a superuser identifier. Amongst the common signals are SIGQUIT, SIGKILL, SIGHUP, SIGTERM (terminate the process with varying degrees of forcefulness), SIGSTOP (suspend the process), SIGCONT (reenable the suspended process), SIGIO (sent by the system to notify application processes of the completion of I/O operations), SIGUSR1 and SIGUSR2 (user definable signals). Each process has a default signal handler for all signals, whose default actions (some of which are mentioned above) depend on the particular signal. Nearly all signal handlers may be replaced by user defined ones; of particular interest are SIGUSR1 and SIGUSR2, to which user handlers may be attached without consequence to process behaviour; this way signals may be used to provide synchronisation between processes. The SIGIO signal will be vital in the communications system drivers which utilise nonblocking sockets.

Note that the signal handler operates asynchronously to the rest of process execution, so care must be taken to ensure that the two threads do not interfere with each other when accessing the process' memory space. Signals are a kernel level feature, and vary significantly in details between UNIX dialects. Originally the reception of signals was an unreliable procedure, though subsequently reliable signals were introduced, albeit with incompatibilities between System V and BSD releases, and requiring a hatful of tricks to use them effectively.

Both semaphores and messages have been measured to incur costs in the order of milliseconds to the participating processes, and must be avoided where possible for our purposes. Reading and writing to shared memory, however, is a machine level operation which lies in the order of microsecond cost. Coordinating shared memory read and writes can turn out to be expensive, especially if encoded using semaphores or messages. Issuing signals is known to be a less expensive operation, and can be used effectively, though it is difficult to guarantee reliability. An alternative would be to avoid descheduling processes which are synchronising for a shared memory read or write, by using a method based on atomic machine code instructions for mutual exclusion:

Test-and-set instructions. A set of inline assembly language functions for performing atomic bit manipulation operations is not supplied with UNIX based C development systems, thus forcing one to resort to architecture dependent methods for ensuring microsecond cost mutual exclusion to shared resources. This facility would be well suited to synchronising shared memory access between UNIX processes where semaphores are too expensive, and between asynchronous 'threads' within a single UNIX process, for instance between a user defined signal handler and the main execution thread. In both cases, the problem of busy waiting must be handled, and may well be avoided. The UNIX scheduler ensures that each full blown UNIX process gets its own timeslice, enabling the use of busy waiting; this does not make it acceptable, since the technique is wasteful of system resources, especially with such coarse grain scheduling. Also, if one of the UNIX processes contains many user level processes (for example, occam processes), only one of which is interested in the synchronisation, the busy waiting would stop *all* the user level processes. Within a UNIX process, the lack of timeslicing between 'threads' prevents the use of busy waiting for internal synchronisation, since a thread attempting to busy wait would proceed forever without being interrupted. It is thus evident that naive use of test-and-set instructions is unsatisfactory; luckily it turns out to be the case that for our intended use, a combination of test-and-set instructions and multiple queues suffice. The algorithm will be described in the design of the communications system.

4.3 TCP/IP communications

The *de facto* standard for UNIX networking is the TCP/IP protocol suite [60, 21]. The corresponding programming interface is the sockets interface [58], which supports not only TCP/IP communications (Internet domain sockets), but also across UNIX domain sockets (within a single host) uniformly. The intended replacement, TLI (Transport Level Interface) never managed to oust sockets; confirming this, the Solaris 2 manuals [48] no more include a statement to the effect that the sockets interface should not be used in new applications, as was originally recommended in Solaris 1 (SunOS 4) manuals [47].

Through the sockets interface, one can send UDP (datagram) packets, wait on a port (as a server) to receive UDP packets from a particular source address, or from any address on a particular UDP port, actively open (as a client) a TCP (stream) connection to a particular port on which a server is waiting, and wait on a port (as a server) to receive TCP connection requests, and accept or refuse on the basis of source address. An established TCP connection exhibits a bidirectional stream abstraction to both ends, across which they may send and receive arbitrarily sized messages, provided a suitable buffer is allocated at the receiving end.

Waiting for a connection at the server end and waiting for a message, are synchronous operations which can block the process in question until they are completed in their entirety. Other operations such as sending messages can also block the process if they cannot be completed immediately. Two approaches to overcoming these problems can be used and combined: nonblocking sockets and asynchronous sockets. Operations on a nonblocking socket do not complete if doing so would suspend the process, but return immediately with an appropriate indication. System calls on an asynchronous socket complete immediately, while the operation proceeds in the background, thus eliminating the blocking problem. Completion of the operation is notified through the SIGIO signal, whose handler can then identify the correct socket using a nonblocking `select` system call. Although some references state that asynchronous sockets must be of stream type, asynchronous datagram sockets seem to function well on the systems which were tested. The general idea is to have many communications over multiple sockets proceeding concurrently with each other and with the UNIX process, notifying their completion through an asynchronous thread (driven by a signal).

TCP/IP carries considerable baggage, and is known to bring about poor utilisation of available bandwidth. In its favour are its popularity, portability across platforms, medium independence, and transparent routing. To improve its performance, knowledge of the particular implementation may be exploited, such as padding messages to the right size to force TCP to flush its buffers, and sizing UDP datagrams to the native packet size of the medium [61]. Using sockets for internal communications is ridiculously expensive, and a direct means is preferred. Though sockets provide a nondeterministic choice (the `select` system call) and a resource channel-like mechanism for queueing messages or connections to a port, these similarities cannot easily be exploited due to the impracticality of integrating internal channels, implemented differently, into the scheme.

4.4 Underlying networking technology

Ethernet-based networks currently dominate as the prevalent local area network technology. Packets approximately 1500 bytes long are sent and received over an ethernet segment. This shared medium introduces contention for the available bandwidth, resolved by collision detection and backoff, which has an substantial impact on latency and bandwidth under heavy loading. At such low speeds, one rarely has to worry about computational overheads such as protocol processing, as the computation speed to communication speed ratio is very high. The problem with using ethernet as an interconnect parallel processing is the fact that bandwidth does not scale with the number of processors, and latency increases suddenly and significantly as the number of processors, and thus packets, increases.

Recently, 100 megabit/second ethernet has become available. Of equal significance, new intelligent switching hubs connecting hosts in a star topology can route packets between any number of pairs of hosts at the full ethernet bandwidth without propagating the packets across all the connected segments, and thus with accordingly reduced contention. The host-network and operating system/protocol processing overheads may increase in significance as ethernet speeds are elevated.

Hailed as the future of telecommunications, ATM's (Asynchronous Transmission Mode) [64, 5] main breakthrough is in the negotiation for, and control over assigned user bandwidth and quality of service. At speeds starting in excess of 100 megabits/second and projected to figures in gigabits/second, the host-network interface and protocol processing overheads dominate. Though negotiating for bandwidth does not actually *create* any of the precious resource, one can guarantee good performance even over non-dedicated, loaded networks. TCP/IP over ATM [52] can change our workstation network scenario significantly, shifting the focus from maximising network usage to reducing operating system and protocol processing overheads while still operating under a heavyweight UNIX-like system.

5 Implementation of occam communications on T9000/C104 networks

The INMOS T9000 microprocessor [41, 40, 37, 36] provides a programming interface for multithreading, and both internal and external communications which fits closely around the occam model. A scheduler interleaves between processes, which correspond closely to occam processes, with sub-microsecond cost. Instructions for communications hide the location (local or remote) of the communications channel. Instructions are included for directly implementing synchronous point-to-point input and output, alternation, and shared channels. With the help of the VCP (Virtual Channel Processor) integrated in the T9000 each of its four links can multiplex a large number of reliable *virtual links*. Each virtual link hosts a pair of *virtual channels*, one in each direction, every one of which corresponds to an external occam channel, and is handled transparently in exactly the same way as an internal channel would be. Used in conjunction with a network of C104 routers [57, 35], (the INMOS C104 implements a 32×32 full crossbar in VLSI with minimal delay) every virtual link emerging from a T9000 can be connected to any other T9000 in the network. The scheduler incorporates two priority levels, and a prioritised choice (alternation) mechanism is provided. The parallelism within the T9000 between the processing unit and the VCP allows communication to proceed without interrupting computation, while the very tight coupling between them enables the VCP to reschedule suspended processes itself on the completion of communication.

The main T9000 facilities for multithreading and communications (internal and external) are summarised below. A direct mapping between occam concurrency and communications primitives and the machine code instructions can easily be made.

5.1 Processes

An occam process can either be active or inactive. An active process can be executing, or waiting for execution, while an inactive process can be ready for input or output, waiting on a timer, or on a semaphore. A process consists of a *workspace* containing information pertinent to the process state, such as a pointer to its executable code, storage for machine registers and the instruction pointer (when not executing), the channel identifier on which the process is waiting (inactive — ready for input or output), and a control word for the alternation mechanism. A single word is shared between several of these. A process is uniquely identified within its host processor by its *workspace pointer*, which also indicates its priority. Processes are descheduled at communication instructions, and at fixed points in the code as determined by the instruction type.

5.2 Channels

A channel consists of a word in memory, uniquely identified by its address. Channels are classified into internal and external by their address (a machine register delimits their ranges):

Internal. The channel word may contain either the value `notprocess`, or the workspace pointer of a ready process. The pointed-to workspace contains further information about the type of communication being effected.

External. External channels can either be virtual channels, or stream channels (only four of which can be used, each of which takes over an external link completely). Virtual channels with consecutive addresses are grouped in pairs, forming virtual links, which have to pass through a single external link. Each virtual channel address is a logical address which is translated by the hardware to the physical address of its virtual link's VLCB (Virtual Link Control Block). VLCBs come in pairs, one on each of the T9000s at both ends of the virtual link. A VLCB contains information about the status of the virtual link, and thus about both its virtual channels (which face opposite directions), such as the current state of communications on both virtual channels, the workspace pointers of (up to two, one in each direction) local processes ready to communicate on the channels, and the external link over which the virtual link passes. VLCBs with pending data packets, and with pending acknowledgement packets are linked into corresponding queues, possibly into both.

5.3 Input and output

The variable length input and output instructions `vin` and `vout` (and their fixed length versions `in` and `out`) send or recieve a message from or to a memory buffer, over a channel specified by its address. Whether the channel is an external one is determined by the hardware from the channel address and a special machine register which delimits external channel addresses. The two cases are handled accordingly:

Internal. The first process to execute a communication instruction on that channel (say P1) finds the value `notprocess` in the channel word. P1's workspace pointer is stored in the channel word, and a pointer to the message buffer (whether it is a send buffer containing data, or an empty receive buffer), together with the length of the message, are stored in P1's workspace. P1's workspace is unlinked from the active process queue. The other process participating in the communication (say P2), eventually executes the communication instruction (the channel address is recognised to be internal) and finds the address of P1's workspace in the channel word. From P1's workspace, the message buffer address and message length are recovered, and a block copy is made to/from the buffer address specified in P2's communication instruction, copying min(*length specified by* P1, *length specified by* P2) bytes across. Finally, P1's workspace is reinserted into the active process queue, and the channel word is cleared to the value `notprocess`.

External. External communication requests are passed on from the T9000's processing unit to the VCP. External channel addresses specified in communication instructions are converted to the address of the corresponding virtual link's VLCB. One of the two processes (say P1) participating in the external communication, executes the communication instruction, and notifies the VCP, which examines the VLCB. The outputing process (say P1) in the external communication executes the communication instruction and notifies the VCP. The VCP in turn sets the send buffer and limiting address, resets the send count, stores P1's workspace pointer, all of which are in the VLCB. Finally it sends the first data packet, and P1's workspace is removed from the active process queue. The receiving side's VCP stores the data packet in a fixed size buffer within it, until a process (say P2) eventually executes an input instruction on the channel. Then the recieve buffer contents are emptied into the specified input address, an acknowledge packet is sent to the original VCP, and the receiving fields of this VLCB are updated accordingly. P2's workspace is removed from the active process queue. On receiving the acknowledge packet, the VCP on P1's side updates the corresponding VLCB and sends the next data packet, and so forth until completion. Finally the VCPs clear the VLCBs and place P1 and P2's workspaces onto the respective processors' active process queues (on the sending side, completion means receiving the last acknowledgement packet; on the receiving side, completion means receiving the last data packet). If P2 had executed the input instruction before receiving the first data packet, it would have been already made inactive, the VLCB would have been filled with its workspace pointer and recieve address and the acknowledge packet would have been sent immediately. Note that although the initiation of input and output is stimulated by the processor (by signalling to the VCP), the remainder of the procedure is handled independently and asynchronously from the processor by the VCP. Completion is also handled by the VCP, which reinserts processes onto active process queues without the processor's intervention.

5.4 Alternative input

A number of instructions are provided which together implement nondeterministic choice of inputting on one of a set of channels. In reality, the transputer implementation of alternative is unfair, favouring channels over others, but is still very usable. A word in the workspace of the process performing the alternative is reserved for storing the operation's current state — this happens to be the same word which contains a pointer to the message buffer during normal communication — which will be referred to as WSp. The procedure is as follows:

1. execute an *alternative start* instruction, which sets WSp to `enabling`;

2. enable every participating channel using the *enable channel* instruction specifying the channel address (whether internal or external; in the latter case, the VCP notifies the enabled state in the corresponding VLCB). If the channel is ready at the time of enabling (that is, the channel word either contains the workspace pointer of another process, or, in the case of a virtual channel, the VLCB's receive buffer contains a packet), WSp is set to the value `ready`;

3. execute the *alternative wait* instruction, which sets WSp to `waiting`, disables the process, and indicates in the process' workspace (call the word WS0) that it is suspended due to an alternative wait, rather than a normal input;

4. when a process eventually outputs on an enabled internal channel one of the values `enabling`, `waiting` or `ready` is found in the process' WSp workspace word: if `enabling`, it is changed to `ready`; if `waiting`, it is changed to `ready`, and the alternating process is reactivated; if `ready`, no further action is taken (at the end of this instruction, it is guaranteed that at least one of the processes on the opposite ends of the channels is ready for output). On an external channel, the output operation proceeds as usual: indeed, the outputting end is not aware of the alternative at the receiving end, though the receiving end's VCP, on receiving the first data packet, notes whether the channel is enabled, and in such an eventuality, directs the scheduler to proceed as for an internal channel;

5. disable every participating channel using the *disable channel* instruction, which checks whether the channel is ready for output, and if so, writes a jump destination specified in the instruction to WS0, provided that a preceding *disable channel* instruction had not previously filled WS0 with another jump destination. This actually makes the whole alternation operation unfair, since channels which are disabled earlier on in the code are at an advantage;

6. execute the *alternative end* instruction, which finds the jump destination pertaining to the selected channel in WS0, and jumps to that location within the same process, which typically contains a normal input instruction hardcoded to the corresponding channel.

5.5 Resource channels

In the T9000 implementation, the contention is moved from the resource channel, to the RDS (Resource Data Structure). In fact, every process 'sharing' the resource channel on the user (client) end is given its own, uncontended-for 'resource' channel, sharing a RDS at the server end instead. The RDS, itself containing a synchronisation word and queue pointers, owns a queue of resource channels, each of which consists of a normal channel word and queue pointers.

- A resource channel may be used as a normal channel (the queue pointers are then ignored), unless it is included as part of a resource (that is, linked into a RDS queue) using the *mark resource channel* instruction. This must be issued on the transputer hosting the server, specifying the RDS, the resource channel address (used to access the server end VLCB for an external channel), and a channel 'identifier', which may be the channel address, or some other word (this is the identification returned when a channel is finally selected). If the resource channel has not been output yet, the value `reschan` is written to the channel word if it is internal and the specified identifier and a pointer to the RDS are written in its other words; a similar annotation is written to the VLCB if it is external. If the resource channel already contains a workspace pointer, or its VLCB's buffer contains a packet, the resource channel is linked into the RDS queue instead, with the specified identifier stored in the resource channel as well.
- The server process issues a *grant* instruction specifying the RDS, and the location where to store the identifier for the selected resource channel. If the queue is nonempty, the top resource channel is unqueued and the identifier it contains is written to the specified location. Otherwise, the server process' workspace pointer is written to the RDS's synchronisation word, and the server process is deactivated. The resource channel is unmarked automatically, and can subsequently be used as a normal channel.
- An output operation on a marked internal resource channel finds the value `reschan` in its normal channel word and subsequently retrieves a pointer to the RDS from one of its other words. If the RDS's synchronisation word points to the server process' workspace, the channel word is set to point to the workspace pointer of the outputting process, the server process is rescheduled, and the channel identifier in the resource channel is written to the location specified in the RDS (originally stored there

by the `grant` instruction which had stalled the server process). Otherwise, the resource channel is queued on the RDS queue.

On a virtual (external) resource channel, the remote end outputs as usual, with no indication that the channel is a resource channel; on the receiving end, the VCP will notice that the receiving VLCB is a marked resource channel upon receiving the first data packet, and notifies the scheduler, which proceeds as for an output on an internal marked resource channel.

5.6 Rationale

It can be seen that the design rationale behind the VCP is to decouple computation and communication to the largest extent possible. Message startup costs suffered by the computational engine are kept at a minimum by delegating such jobs to the VCP, which runs in parallel. Message sending and receiving proceeds independently. Completion of communication involves only reinserting the process in question on the active process queue. This can either be done by the VCP itself, through careful mutual exclusion to access of queue registers, or delegated by the VCP to the main processor, to avoid possible contention. Though no such details are specified in the T9000 documentation, it seems that the former situation is the case. This is only feasible as the VCP will always work with the same scheduler: portability between different run time systems and schedulers is not an issue here, so the VCP can be exposed to the internals of the computational engine.

The resulting situation enables multiple external channels to connect transputers, without any concerns about locality influencing the program designer. In fact, the same program code can be redistributed across transputer networks in an arbitrary manner, changing the status of channels from internal to external and vice versa in the process, without having to alter a line of code. When used in conjunction with a network of C104 routers, all packet routing concerns are delegated to the routers, so proximity of transputers is of no importance. The C104 routers provide hardware support for randomised, or two phase routing (in the form of header deletion), so that all packets are routed (using a deterministic protocol guaranteeing deadlock freedom) to a random intermediate C104, which then strips the outer header from the packet, exposing the final destination, and routes it deterministically to the destination T9000 link. The effect of this is to reduce hot spots on the network. Still, particular links on a transputer can still manifest themselves as hot spots which cannot be removed in this manner. A suggested solution is to randomly distribute memory locations across transputers using a fast good hash function, thus reducing the possibility of contention for a transputer's link. Alternatively, one can resort to careful placement (possibly automated through the use of heuristics) of processes and virtual channels on processors and links to eliminate these problems. In our case, all routing decisions will be taken by the underlying system software (such as IP), and are not a concern of ours. Our focus will be on designing a counterpart to the T9000's VCP for handling external communications.

6 The Design of an occam communications system for UNIX workstation networks

The selected design for the communications system borrows heavily from the T9000 VCP in its decoupling from the occam computational engine. Following the arguments presented in section 4, it was decided to isolate the communications system and the computational engine into two separate UNIX processes. The main difference is that while the T9000 VCP handles the scheduler registers directly, here this is not desirable, to ensure portability of the communication system across different schedulers, and therefore the adapter of the run time system provides a routine to be called from the communication system to carry out the scheduler-dependent work. The overhead for synchronising the communications process and the computational process is kept low through the use of shared memory and the direct embedding of test-and-set instructions. Busy waiting is avoided through the use of a synchronisation algorithm described later in this text.

Another similarity is in the provision of a packet driven communications facility built on the lines of the VCP. This enables the use of a reliable packet driven link to send multiple messages of unlimited size in the same way as the VCP multiplexes its packet driven links. Together with this facility, dedicated stream communications is allowed over reliable continuous stream links. These two methods may be used concurrently over different links, depending on the links' characteristics. Figure 6 gives an overall view of the system structure.

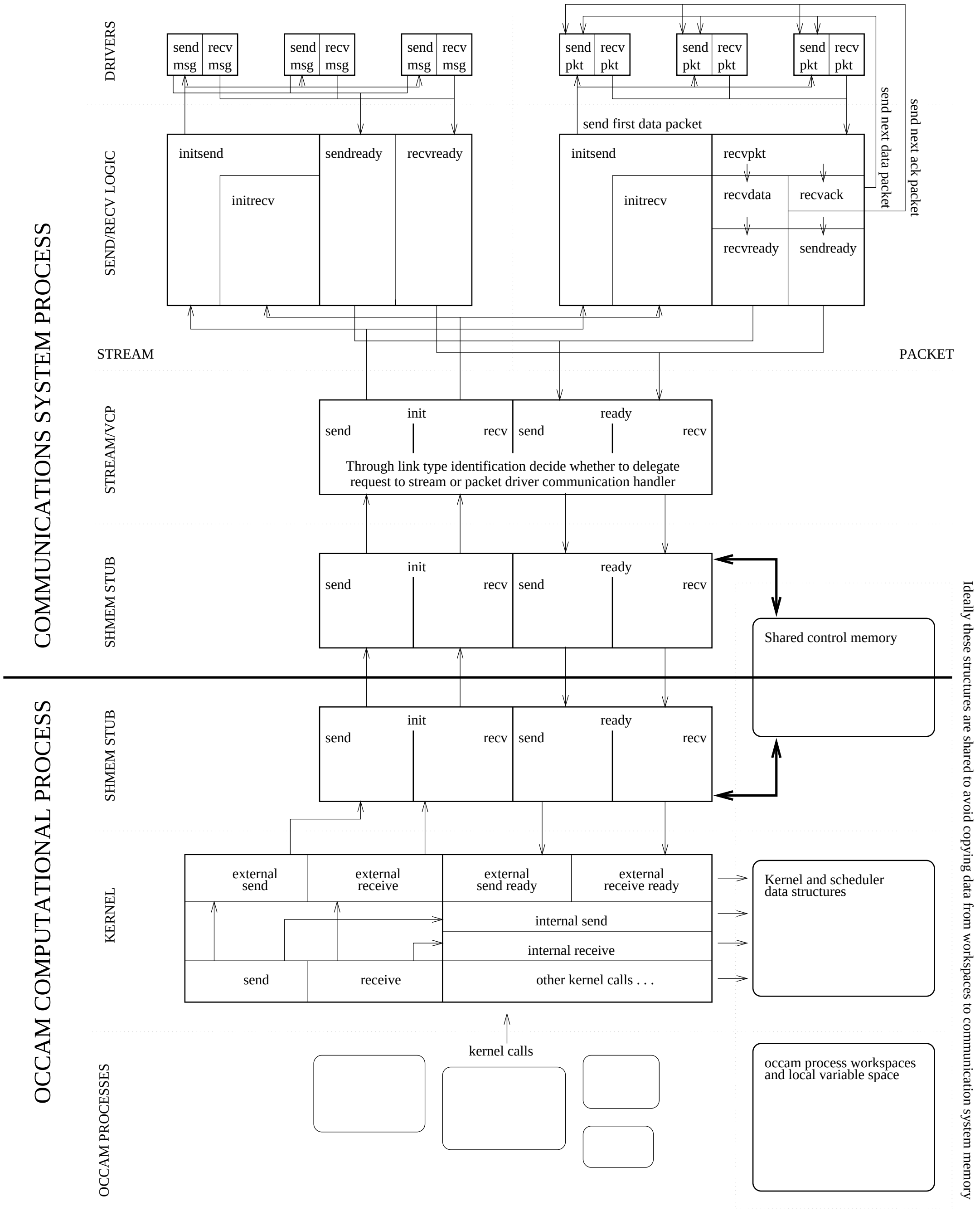

Figure 1: A visual representation of the system's structure.

6.1 The network: links and drivers

Two classes of links are envisaged to be available in a typical operating environment: reliable streams, and (reliable or unreliable) packet driven links. The former can also be multiplexed to provide multiple reliable packet driven links over the single stream. Support for both stream and packet driven communications is included in the communications system design; correspondingly, two classes of driver are defined, one for sending and receiving packets, and another for sending and receiving unsegmented messages. Third party drivers for alternative transport mechanisms may be linked to the communications system through standard interfaces, by recompiling the communications system alone. The everchanging suite of drivers prompts the need for a means of identifying the range of link types which they introduce, in a suitable network description language. A table of currently available link types will have to be maintained for the program linker or loader to refer to.

Underlying routing mechanisms are hidden from the communications system. Usually, the drivers themselves will utilise the routing mechanism provided by their transport protocol. For instance, TCP, UDP and raw IP based drivers will rely on IP routing. Thus the function which is performed by networks of C104 routers in T9000 systems is hidden from our concern.

6.2 Packet driven communications

The packet driven communications facility in the communications system will identify over which link an external channel lies, from the associated VLCB-like structure, identify the link's driver (obviously a packet driver), and send data or acknowledge packets through the driver's standard interface, which should never block. An asynchronous thread flows in the reverse direction, activated by the reception of packets over various packet drivers (this will, in most cases, be a SIGIO signal handler which notifies completion of I/O operations). This thread, immediately dispatched to the appropriate packet driver, will then move to the communications system routine for handling received packets, which usually sends a data or acknowledgement packet in response except at the end of a message (in which case the occam run time system is notified). The VCP logic for sending and receiving messages is drawn on heavily in the design of this facility. This method has the advantage of requiring only a small, fixed size buffer (specified by the particular driver) for storing initial packets of received messages; subsequent packets are only received once the relevant acknowlege packet is sent.

It is inevitable that the speed of unreliable packet driven links such as those provided by UDP, and special access to raw IP will make them attractive. It is risky to use packet drivers directly built on these protocols without provision for guaranteeing reliable and correctly sequenced reception of packets. One solution is to delegate all such concerns to the packet driver; alternatively, one may include a third facility in the communications system for handling unreliable packet driven links.

6.3 Stream communications

The stream communications facility is relatively simple; the link for a particular external channel is identified from an appropriate stream channel structure, and the link's stream driver determined. Messages to be sent are expedited through a single driver call, which should never block. Received messages, and completed sends invoke an asynchronous thread (usually through a signal handler), dispatched to the appropriate stream driver and propagated to the communications system routine for receiving unsegmented messages, which will notify the occam run time system. Limitations on the number of simultaneous connections for the transport protocol on which stream drivers are based may restrict the use of stream links. Since whole messages of any length are received irrespective of whether the receiving process has issued a receive and a destination buffer, intermediate buffers of arbitrary size may have to be allocated to store incoming messages on a stream channel.

6.4 Example drivers and their use

Drivers can be built upon various transport mechanisms, utilising their features in a wide variety of manners. The following list of possible drivers which may be implemented is based on speculation rather than definite results:

TCP Socket (Stream) pre-established connection. TCP connections are established at load time between machines, as specified in the network description by links with the appropriate driver type. A send message causes the data to be sent on the stream connection, as a non blocking call. Completion is notified by a SIGIO signal, and delegated to the driver for appropriate handling (causing reactivation of the occam process). Message reception is similarly notified. A TCP socket is dedicated to a link throughout the running time of the occam program, so this driver type should be used sparingly. Connections can be maintained between executions of programs running on the same network configuration, so the cost of establishing connections is minimised. It is necessary to use knowledge of TCP protocol implementation to overcome its inefficiencies. Results presented in [61] give an idea of the tricks which a driver implementation may have to resort to.

TCP Socket (Stream) connection at run time. There are many variants on this theme, where TCP connections are established at run time, resulting in an overall performance degradation but better utilisation of the limited number of simultaneous TCP connections. Connections can be opened and closed on a per-message basis; alternatively, one can maintain the maximum number of simultaneous open connections possible, and on subsequent requests, discard connections on a least recently used basis. It is expected that not enough information would be available to the driver to open and close connections on the basis of a channel's lifetime in the occam program.

UDP Socket (Packet). The packet driven nature of the UDP protocol lends it to a natural packet driver implementation. Packets associated with a link of this type are immediately output through the UDP socket to the destination end. The overheads associated with TCP are not encountered, though it is possible that other hidden inefficiencies manifest themselves, according to the results presented in [19]. The connection establishment phase is eliminated, though at load time, implicit 'connections' may be made between pairs of client and server UDP sockets on machines between which such a link is specified. An alternative driver may share a single UDP socket between all its links. It must be kept in mind that UDP packet delivery is not guaranteed to be reliable, or in sending order, so a mechanism must be introduced into the driver for guaranteeing these properties. Alternatively, a generic communications facility for driving unreliable packet links may be used, if provided in the communications system. However, on a single ethernet segment operating in a reasonable environment (that is, without physical disconnection, power failures, or interference), packet collision detection is handled by the ethernet card, and packets do arrive in order of transmission; thus a UDP mechanism on this medium can be expected to operate without problems, though it would be very unwise to guarantee faultless operation to users!

Raw IP Socket (Packet). Though the sockets interface does not usually offer raw IP as a freely available service, on a dedicated network of machines where security is not an issue one can create such a service available to non-root processes. Potential benefits may be experienced from bypassing the UDP protocol mechanisms and any associated buffering. Similar arguments to those mentioned in the discussion on reliability of UDP packet drivers operating on an ethernet medium apply.

TCP Socket (Packet). A single TCP socket between two hosts may be multiplexed between many virtual links by defining a packet format to be transmitted on top of the stream abstraction. While this may seem to be terribly inefficient, it overcomes the limit on maximum simultaneous connections; moreover, if the size of packets is tuned to match the TCP implementation's buffer size and the IP packet size, the amount of extra packets generated by this method can be kept to a minimum.

Character device (Stream). Unix character devices, such as serial and parallel ports on a workstation, can be exploited as links to other workstations, or as links to external devices to be controlled. Of the drivers discussed, this is the first which does not have IP routing support, thus exposing physical network structure restrictions in the network description and the mapping of processes and channels onto links. Program loading across such links, and their configuration, has to be delegated to adjacent nodes.

Character device (Packet). A packet passing abstraction may be built on top of a character device stream, similar to the TCP socket packet driver. Thus multiple channels may pass across a single serial line, connecting processes residing on the physically connected machines.

Raw Ethernet (Packet). Provided a UNIX driver provides unrestricted direct user access to the ethernet interface (which is not usually the case in a general purpose workstation cluster) a packet driver may be built to send packets directly through the ethernet card's packet driver. IP's routing capabilities are obviously not available, and such packets are restricted to the current ethernet segment. However, it is expected that this would be the most effective means of utilising an ethernet segment.

PVM messages (Stream). Although all the potential driver implementations mentioned above target operating system services directly for communicating, it is possible to make use of a driver to send messages through higher level message passing software such as PVM. The communications system's scope changes to that of providing a uniform interface to the occam run time system, isolating the emulation of occam primitives such as alternation and resource channels in PVM to the appropriate driver [20], and reducing the impact of PVM's high delays in initiating a communication.

DS-links (Stream). SGS Thomson Microelectronics' DS-links protocol has been made an IEEE standard (IEEE 1355). This is essentially the same communications technology on which the T9000 and C104 are based. Third party products based on DS-links are expected to emerge shortly. In particular, a PCI bus based DS-link interface for PCs is imminent [67]. It can be expected that similar interfaces for various workstations will be available. While providing a fast, low latency and scalable interconnect suitable for parallel processing with workstations, this option provides a clean way of interfacing workstations with existing parallel computers, more so if DS-links are to be used in a wide range of parallel systems. In our system, DS-link virtual links can be exposed through a stream driver.

6.5 Alternation and resource channels on external channels

Though the discussion up to this point caters for straightforward unconditional and uncontended-for message passing between pairs of processes, it is equally important that the implementation of alternation and a resource channel mechanism to handle a mix of internal and external channels is straightforward and inexpensive.

Alternation logic has to be based in the occam run time system, so that purely internal alternation is inexpensive. Introducing an external channel into an alternation requires an addition to the communications system, in particular, for enabling the external channel (*enable channel* instruction). The run time system, on being instructed to enable an external channel notifies the communications system, which marks the enabled status in the VLCB of a virtual channel, and in some suitable way for a stream channel. The communications system, on receiving a packet on the channel, or an entire message in the case of a stream channel, notifies the run time system, which then proceeds as for an internal channel, reenabling the alternating process if it was already `waiting`. The alternating process proceeds by disabling channels as usual, at which point it may turn out to be necessary to notify the communications process of the disabling of any external channels. The vital property is that the performance of alternation is not degraded by the use of external channels.

Similarly, most of the resource channel logic will be handled in the run time system, so as to keep overheads low, especially for wholly internal operations. The RDS (Resource Data Structure) together with its associated resource channel queue is held by the run time system. It is expected that the marking of external resource channels be propagated to the communications system, though the way in which the run time system would handle a resource queue containing external channels, without suffering major alterations, has not yet been decided.

6.6 Interfacing with the occam computational process

The interface between the occam computational process and the communications system process is crucial for maintaining low message startup times, at least within the occam computational process. A particularly low cost interprocess communication mechanism is vital. From the discussion conducted in section 4, conventional means were judged to be too expensive. An alternative which was briefly mentioned based itself on the utilisation of shared memory (a standard UNIX System V IPC facility) for passing data, and test-and-set instructions or signals for synchronisation. The exact technique is still to be decided upon.

It is important that internal communications remains as inexpensive as originally intended by the occam run time system. External channels can be distinguished from internal ones using a method chosen for the particular run time system, at best consisting of a single comparison of the channel address with a bounding address, thus increasing the execution time of an internal channel operation by only a few cycles. On recognising an external channel, the run time system calls a stub, which in turn communicates the request to the communications system process in a manner hidden from the rest of the run time system. Notifying the occam run time system of the completion of an external communication requires the provision of a routine which runs as part of the run time system (asynchronously to the rest of the system if called by a signal handler, or as part of the same thread of control if activated by polling a shared memory flag), to reschedule the relevant process. This cannot be included as part of the communications system process, as the latter would be rendered run time system dependent. Notifying the receipt of a message (or the receipt of the initial packet in the case of packet communications) is handled in the same way. It is thus hoped that as little as possible of the occam computational process' allocated timeslices are spent on handling external communications, and that the process never performs system calls itself (though it is not clear whether this will gain more processor time for the occam computational process — after all, the communications process is stealing time from the same processor resource — except maybe in the case of a multiprocessor UNIX scheduler, where it might also be feasible to introduce more than one occam computational process on a machine).

It is equally important to allow absolute freedom to the occam compiler and run time system over the assignment of external channel addresses. The compiler/run time system pair should aim for a fast way of separating internal and external channel addresses. Besides, the code which a compiler generates and the corresponding run time system supports, may be retargetted from a transputer-like architecture. Thus the communications system must be able to help the run time system to emulate various memory maps, such as the T800's eight external channel address structure, and the T9000's variable number of external channel addresses delimited by a register, and translated from the virtual address they represent to a physical VLCB address (pairs of virtual channel addresses map to the same VLCB address). In the latter case, the mapping from external channel addresses (which the run time system recognises) to the VLCB address has to be performed by the communications system. This will be implemented using a look up table mapping each external channel address to the address of its VLCB, or the corresponding structure representing a stream channel. When setting up external channels at load time, the entries are inserted by making calls to the communications system, specifying the channel address, and optionally, a pointer to the VLCB address. If the latter is not supplied, the communication system fixes up its own memory area for VLCBs. It is planned that in this way, even straightforward translated T800 code may have its eight external channels passing over packet driven links represented by VLCBs.

6.7 Rationale for the proposed design

A well known way of hiding communications delays to obtain high resource utilisation is by overlapping communication and computation. A remaining source of underutilisation is the message startup cost, which is inevitable, but can be pushed downwards. In our case, this corresponds to the notification of the communications system by the occam computational process of the sending of a message over an external channel. This is kept to a minimum through the use of shared memory and lightweight synchronisation mechanisms such as signals and test-and-set instructions. In our system, an analogous price is also paid on completion of external communication.

Overlapping communication with computation still procures considerable gains. The time for protocol processing can be handled in parallel (as in the T9000's VCP), though in our case separating the task in a separate process still steals away time from computation in a uniprocessor system where the single processor resource is shared by all. The system may benefit from the use of a multiprocessor, since the occam computational process may be scheduled in parallel with the communications system and associated system calls. This time often pales into insignificance when compared with the time taken for packets to travel across current networks (luckily unaided by the processor). Hiding *this* delay is where overlapping becomes indispensable. Our system exploits overlapping of computation and communication in two ways: by separating the occam computational engine and communications system into two UNIX processes, and by performing all communications in the communications system as non blocking calls. Though it may be argued that the latter duplicates the function of the former, separating them into two UNIX processes has other advantages (in the multiprocessor case, and for compiling in new drivers in the communications system without recompiling occam executables). One can formulate a simple model to represent the cost of communication:

$$
\begin{aligned}
s &= \text{message startup/completion cost (time spent on notifying communications system)}\\
p &= \text{protocol processing cost (time spent in communications system and system calls)}\\
d &= \text{network communications delay (time spent waiting for packets/message to be delivered)}\\
p + d\ (+s) &= \text{total communications cost without overlapping}\\
p + s &= \text{total communications cost with overlapping, protocol processing using same processor}\\
s &= \text{total communications cost with overlapping, protocol processing using a separate processor}
\end{aligned}
$$

A universal property of distributed memory message passing systems is that d, the network communications delay (or remote memory access time), is much higher than that for local operations. p is incurred by local operations and should be dwarfed by d (although high speed networks can change this, improvements in scheduling might more or less keep the balance; however, this point is debatable). Finally, s is kept as low as possible through the use of shared memory and lightweight synchronisation mechanisms. The use of multiprocessor workstations and suitable scheduling algorithms would place the total communications cost somewhere between s and $p+s$, obscuring d completely as long as sufficient parallel slackness is available in the occam program. In a T9000/C104 system, s is negligible (an onchip synchronisation), and p is met by the VCP.

6.8 Implementation considerations

Various issues have arisen during the implementation of the communications system, which is currently in progress. Many problems are caused by the lack of standardisation in the UNIX programming interface. Other difficulties can be traced back to the inadequacy of the TCP/IP and ethernet implementation for such applications, and the need for detachment from the occam run time system.

Early signal mechanisms were unreliable, which meant that signals were prone to getting lost. Also, system calls interrupted by signals did not regain control. While such implementations are virtually nonexistent today, and reliable signals prevail, there are still significant variations in signal semantics between UNIX dialects. These must be catered for in our implementation, as signals play a vital part in our implementation strategy. In particular, SIGIO signals in conjunction with asynchronous socket I/O are crucial. It is important to establish to what extent datagram sockets can be used if asynchronous support for them cannot be guaranteed across all platforms. A rather frustrating experience with socket libraries in various UNIX implementations is due to their nonreentrancy. Though various new releases make socket calls 'multi thread safe', or reentrant, one must cater for existing nonreentrant libraries in common use. Our application makes it necessary to make socket calls from within signal handlers, which may interrupt the main process flow at any time.

The importance of performance in our application forces us to resort to fine tuning our TCP/IP based driver implementations. Through padding TCP messages to multiples of a particular size, one can force the immediate flushing of a buffer which attempts to combine smaller sends together. This buffer often introduces send delays in the order of hundreds of milliseconds, all for the sake of better bandwidth utilisation, and may be detrimental to our overall performance [61].

The socket interface requires the user to point to the start of the message to be sent. In a packet driven scheme, the message must start with a VCP-like header specifying the destination VLCB address and other necessary information. To construct this message structure, it is necessary to introduce memory to memory copying for each and every packet to be sent. This was found to be inevitable, as even the strict occam usage rules do not permit the construction of packets in place *within* the message (by displacing header-sized segments from the message temporarily) without potential interference. It may be presumed that the same situation occurs within the IP to ethernet interface, where IP, in constructing ethernet packets, may have to perform memory to memory copies again. On the other hand, it can be argued that the cost of a block copy is minimal compared to the amount of processor time spent inside the UNIX system internals.

Certainly, it is desirable to minimise the amount of work the occam computational process spends on work related to external communications. One would like to avoid copying messages to be sent over an external channel from process workspaces to some buffer area in shared memory, also accessible by the communications system process. For this reason, it is preferable to place the occam workspaces and data areas in shared memory as well, so that the communications system can access directly messages to be sent without involving the occam computational process. However, as some run time systems would not fit well within such requirements, this must not be enforced, but be included as an option which can be refuted. As a penalty, noncompliant run time systems will have to perform copying of messages from workspaces to the separate shared memory area.

7 Lightweight asynchronous access control to a shared resource without relying on critical sections, semaphores, or busy waiting

We would like to share a resource between a fixed number of asynchronous processes, while guaranteeing mutually exclusive access. However, the Draconian restrictions which are imposed complicate our task considerably:

- no scheduling support can be relied upon, which implies that a process cannot wait on a semaphore; however, a process may be interrupted by other processes at any time during its execution;
- the only synchronisation mechanism allowed is an atomic test-and-set instruction, together with other typical atomic bit twiddling instructions;
- busy waiting is not allowed for two reasons: we cannot rely on scheduling support to timeslice between processes; thus a busy waiting process may proceed indefinitely; moreover, busy waiting is wasteful of precious processor time;
- critical sections cannot be guaranteed to be uninterruptible, as no lightweight implementation such as disabling interrupts, or using busy waiting and test-and-set instructions can be utilised.

It is evident that these restrictions prevent a solution from being reached. Fortunately, we can relax our requirements to arrive at a less taxing formulation: When an access request is made, the access can be carried out at a later time, not necessarily by the requesting process itself, provided that sufficient information is supplied about the request. No acknowlegement of satisfaction of the request is expected by the requesting process, whose subsequent computations do not use any results from the request. These concessions turn the seemingly impossible problem into a more reasonable version, which we will attempt to solve. A progression of attempts towards a solution will be described, until a seemingly satisfactory algorithm is arrived at. No formal claims are made regarding its correctness, though empirical tests have sustained the guarantee.

7.1 Intended applications

The need for such an algorithm arises at various points during the design of the communications system:

- The SunOS/Solaris sockets library is nonreentrant (at least up to Solaris 2.2) [48]. This may cause problems, in that initial message packets are sent from the main communications system thread of control, while subsequent packets are sent from a signal handler, which may interrupt the main thread in the middle of a socket system call. Socket calls may be considered a resource to be shared using such a mechanism, since no proper scheduling support exists between these two threads, and busy waiting would give rise to an infinite loop. Results from these socket calls may need to be handled centrally, rather than by the invoking thread.
- We expect synchronisation and communication between the communications system process and the occam computational process (for invoking message sending and receiving and notifying completion of these operations) to be least expensive through shared memory, using a similar, possibly altered, technique to synchronise access to the shared memory resource. Though UNIX interprocess communcations facilities and signals are available, our technique could give rise to much smaller delays. Busy waiting can be used in this case, depending on UNIX scheduler timeslicing, but the coarse granularity would result in large amounts of processor time being wasted.
- If the notification of external communications operations is done through signals to the occam computational process instead, the signal thread needs to access occam process queues, and may interfere with the main thread. Our technique may be applied directly towards a solution.

7.2 A series of unsatisfactory solutions

Unsuccessful attempt #1: Single queue

In an environment with adequate scheduling support, processes claiming access to the resource may be queued, and reenabled on being granted access. In our case, a process claiming access cannot be suspended for later reenabling, but has to proceed immediately. Also, busy waiting on an access bit is disallowed, since a process may wait forever. A critical section, during which interruptions by other processes are disabled, cannot be implemented. We are led to a situation where an access request must be got rid of without delay. One way of doing this is by queueing requests instead of processes. A process P_i claiming access queues a request on queue Q (figure 2) and proceeds as though the request has been carried out (we are assuming that results from the access are not needed). The resource handler RH, through which accesses are made, is protected by a locking bit, *lock*-RH, which is tested and set atomically by processes after queueing a request. If unlocked, RH is entered, using processor time allocated to P_i. If already locked, P_i proceeds as usual. In the latter case, queued requests must be serviced by RH as soon as it is ready. Thus, once RH is active, it

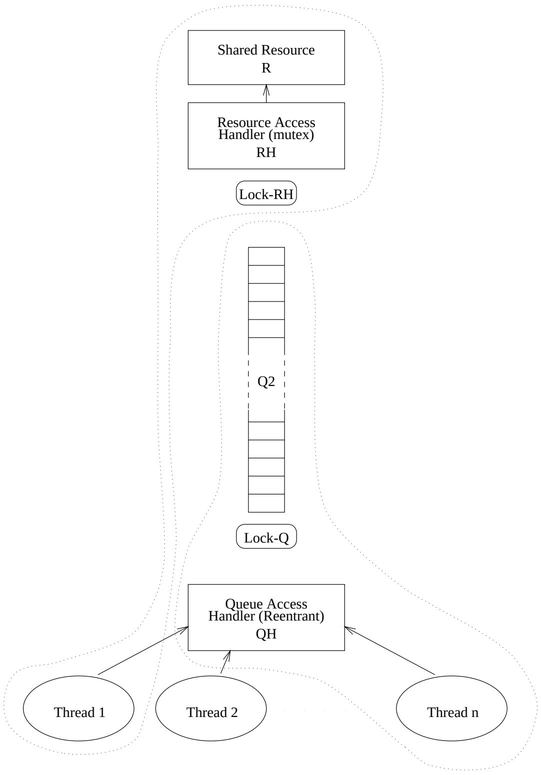

Figure 2: Attempt at mutual exclusion with one queue.

must check Q for requests added while *lock-RH* was on and service them, every time before exiting. Such subsequent requests are serviced as part of the process which itself had originally managed to lock RH, rather than the requesting process.

Unsuccessful attempt #2: Multiple queues
The previous solution is satisfactory so long as queue access is atomic. This is unfortunately not usually the case, unless critical sections are allowed (in which case we could do away with the queue altogether). In locking queue access, a locking bit *lock-Q* would have to be added, on which the same mutual exclusion problem initially tackled arises. To solve the problem of contention for queue access, assuming that a constant number n of processes are contending for access, n queues $Q_1 \ldots Q_n$ can be introduced, each with a locking bit lock-$Q_1 \ldots$lock-Q_n, as shown in figure 3. At any time in which a process needs to queue a request, at least one queue will be unlocked, and the process may scan for that queue. Once found, the request is added to that queue, and the queue is again unlocked, and an attempt is made to lock and enter RH, which may, or may not be successful. RH, once active, will scan for an unlocked, non-empty queue, lock it and service all requests, and repeat this until a scan through the queues results in locked or empty queues exclusively, in which case, RH exits, allowing its host process to continue execution. In doing this, however, we have pushed the atomicity problem to the scan for unlocked queues! Both cases (a process looking for a free queue, and RH looking for a free, non-empty queue) introduce new problems:

- Suppose a process P_i scans through the queues, adopting an arbitrary deterministic strategy (say from left to right in the visual representation), looking for an unset *lock*-Q_i using a test-and-set instruction. It is possible that it scans from Q_1 through to Q_k, $1 < k < n$, finding every one locked. It is guaranteed that at least one is free, so that one must be between Q_k and Q_n, say Q_j, $k < j < n$. However, at that point, a queue is freed between Q_1 and Q_k, and Q_j is locked. The scan proceeds and P_i does not manage to find a free queue. Eventually, one could argue, a free one is found after an arbitrary amount of rescans, but a situation could be constructed (using a particular scheduling pattern) where this iterates forever. A possible solution is the adoption of a nondeterministic scanning algorithm, where the queue to be checked is selected randomly. This will guarantee an eventuality condition, though no upperbound on the number of retries can be fixed.

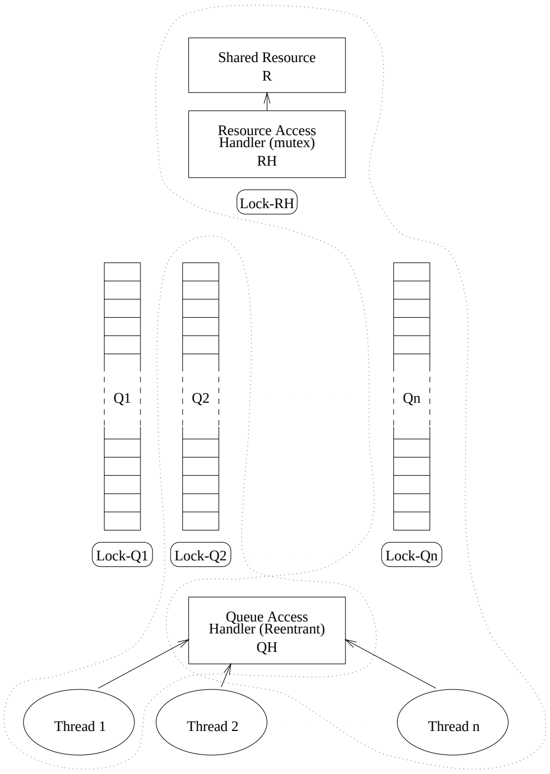

Figure 3: Attempt at mutual exclusion with n queues.

- Upon completing a request, RH scans the queues for any further pending requests. A deterministic scan has to be made here with a fixed upperbound, since it is important that RH exits as early as possible. A deterministic scan introduces the same problem which we have just encountered, only that changing to a nondeterministic scan causes RH to proceed even while no requests are pending, until a request is submitted.

Although the mutual exclusion condition is satisfied, and we can consider this to be a satisfactory solution, requests may be left pending for an indefinite period, left waiting on a queue by an unaware RH which has since exited, until its next invokation. Though we have not set any hard and fast constraints on response time, we choose to view this situation where no upperbound on servicing time can be guaranteed as unsatisfactory.

Unsuccessful attempt #3: Multiple queues with flag
One can state our latest problem as having to detect requests queued without the knowledge of RH. Enclosing the scan routine in a critical section would be fine, but, making use of critical sections, we could have done without the queues in the first place. Alternatively, adding a flag bit f (figure 4) alongside the queue locks, which would be set exactly before, or after, unlocking a queue, could act as a notifier for RH, prompting it to rescan the queue. At the start of a scan, RH would reset f. If at the end of the scan, f is set, this means that a request has been queued in the meantime, and the whole routine is repeated. This seems to work fine in both cases, probably in a very high percentage of executions, but not quite always. Consider the sequence of events in figure 5. By following the sequence, it is evident that P_1's request remains queued, and is not serviced until the next call to RH. This alteration has not solved our problem, though it reduces its occurance frequency substantially. This technique has been described here as it leads the way to our current best solutions.

7.3 Two (hopefully) correct solutions

The unsuccesful attempts just described strongly indicate that it is not possible to solve the problem with the available tools. In fact, the solutions which we shall now supply twist the conditions subtly in our favour, though to an extent which is acceptable for our application. The problem with attempt #3 is that if an

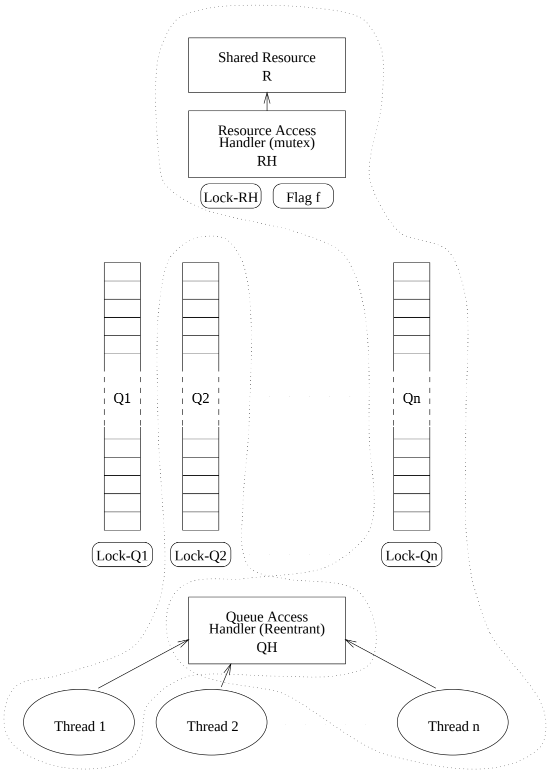

Figure 4: Attempt at mutual exclusion with n queues and a flag.

interruption occurs in between setting the flag f and clearing the queue lock (during which f is cleared), the same process is rescheduled while *lock*-RH is still set. Consequently its latest request is ignored. This calls for uniting the setting of f and the unlocking of the queue into a single, atomic operation. Alternatively, restricting possible schedules to a subset of those originally possible, in which interrupted processes cannot be rescheduled by interrupting another process, will prevent the occurance of such an event. We shall investigate the impact and practicality of these restrictions in turn.

Solution #1: Atomic clear flag and unlock queue combined operation
One way of preventing residue requests accumulating until the next execution of RH is the use of an atomic instruction which clears f and *lock*-Q_i. How practical is this — does it require the provision of specialised hardware, or is it implementable using conventional processors? Provided n, the number of threads, is less than the machine word size, one can store *lock*-Q_1 ... *lock*-Q_n in bits 1 to $n-1$, and reserve bit 0 for f. Test-and-set instructions usually operate on a selected bit of a machine word, so this is an acceptable arrangement. Now, bits from a word can be set and cleared atomically using machine instructions for `AND`, `OR` and `XOR`. An `XOR` with a word whose bits are all zeros except for the relevant queue bit, *lock*-Q_i and the flag bit f fails in the eventuality that f is already set at the time. However, if f is inverted, that is, set to 0 and cleared to 1 instead, an `AND` with a word whose bits are all set to one except for the flag bit f and the relevant queue bit *lock*-Q_i should satisfy our requirements, setting f and clearing *lock*-Q_i atomically. This is achieved using standard hardware, with the restriction that the number of processes must be less than the machine word length.

Solution #2: Restricted scheduling order
The sequence of events in the counterexample given in attempt #3 invalidate our requirements because P_1 (the process whose request will eventually remain unserviced) after being interrupted regains control by in turn interrupting P_2, which is in RH at the time, about to clear *lock*-RH. By enforcing the following condition:

> An interrupted process may not regain control before the process which had caused the interruption terminates

P_1	(scans queues)
P_1	test-and-set *lock*-Q_3 (successful)
P_1	queue a request in Q_3 (now it is the exclusive owner)
P_1	set f
P_2	(scans queues)
P_2	test-and-set *lock*-Q_1 (successful)
P_2	queue a request in Q_1 (now it is the exclusive owner)
P_2	set f
P_2	clear *lock*-Q_1
P_2	test-and-set *lock*-RH (successful)
$P_2(RH)$	(scans queues)
$P_2(RH)$	test-and-set *lock*-Q_1 (successful)
$P_2(RH)$	unqueue a request from Q_1
$P_2(RH)$	service the request
P_3	(scans queues)
P_3	test-and-set *lock*-Q_1 (successful)
P_3	queue a request in Q_1 (now it is the exclusive owner)
P_3	set f
P_3	clear *lock*-Q_1
P_3	test-and-set *lock*-RH (unsuccessful)
P_3	(proceeds with execution)
$P_2(RH)$	clear *lock*-Q_1
$P_2(RH)$	test f (it is set, so rescan)
$P_2(RH)$	clear f
$P_2(RH)$	(scans queues) (unsuccessful)
$P_2(RH)$	test f (it is not set, so prepare to exit)
P_1	clear *lock*-Q_3
P_1	test-and-set *lock*-RH (unsuccessful)
P_1	(proceeds with execution)
$P_2(RH)$	clear *lock*-RH
$P_2(RH)$	(exits, returning to P_2)
P_2	(proceeds with execution)

Figure 5: A scheduling sequence which leaves a residue in queues.

an ordering on the rescheduling of suspended processes is enforced. Though this ordering could possibly be relaxed to a partial order of scheduling constraints, as yet this has not been determined. The crucial point is whether maintaining this ordering is unnatural to the system in question. Luckily, it turns out that this condition is kept by an interrupt driven system, as exemplified by signal interruptions, since return addresses are often stored on a stack, to be popped off, naturally conserving the order. With such a scheduling restriction, P_1 in particular can never barge in when P_2 in RH is about to clear *lock-RH*. Subsequently, RH is made available before P_1 resumes, *lock-RH* is test-and-set successfully.

As yet, no verification of the correctness of these algorithms has been carried out, the only tests performed being empirical in nature. Still, at the time of writing no counterexample has been found which demonstrates the invalidity of these solutions.

8 Future Directions

To date, various parts of the communications system have been implemented, mainly pertaining to packet driven communications. A variant of the shared resource access algorithm described has been implemented as part of a separate project. Diverse issues have been exposed as this project proceeded, offering opportunities for ulterior research. Openings which may be investigated further in future include:

- implementation of the communications system in its completion, and integration with a compiler and run time system such as KROC [51]; initially, TCP/IP-based drivers will be constructed;
- design and execution of tests for the measurement of overall system performance over an ethernet-based workstation network; it is hoped that the results compare favourably with those for existing message passing implementations; isolation of costs pertaining to particular sections of system operation, such as message startup cost;
- implementation of further drivers, performance testing and evaluation;
- investigation of issues in the design of a network description language to model the (possibly virtual) topology of a heterogenous system, and the driver type used for each link; the corresponding software configuration language will be in line with current designs; it would be desirable to automatically generate software configurations, although this may fall outside the project's subject area;
- examination of abstract models such as BSP and LogP which attempt to mould a model of parallel computation that enables straightforward performance prediction, and the extent to which it is possible to obtain similar benefits with occam's freely communicating CSP-like model; to what extent is it possible to automatically determine the optimal amount of occam process distribution across a network (with suitable performance characteristics) to maximise efficiency, using only parameters such as the machine's granularity and latency?
- alongside with the above line of thought, investigate the possibility of enforcing restrictions similar to those imposed by the BSP model into the computational kernels of occam programs, in order to ease performance prediction for computation intensive routines; correspondingly, a novel implementation of distributed occam can be investigated and optimised towards such use, if possible without changing the syntax or semantics of the language; otherwise, investigate relevant language extensions or restrictions, their performance, and ease of performance prediction on real parallel architectures.

References

[1] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheinman. LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. Technical report, University of California, Santa Barbara, 1995.

[2] James Allwright. The WP6 BSP-occam library. Technical report, University of Southampton, March 1991.

[3] S.A. Baker and K.R. Milner. A process migration harness for dynamic load balancing. In Janet Edwards, editor, *WoTUG-14 occam and the Transputer — Current Developments*, pages 52–61, Amsterdam, 1991. IOS Press.

[4] C. Barnaby, M.D. May, and D.A. Nicole. General purpose parallel computers. In M.D. May, P.W. Thompson, and P.H. Welch, editors, *Networks, Routers and Transputers*, chapter 8, pages 119–132. IOS Press, Amsterdam, 1993.

[5] C. Barnaby and N. Richards. A generic architecture for ATM systems. In M.D. May, P.W. Thompson, and P.H. Welch, editors, *Networks, Routers and Transputers*, chapter 10, pages 151–182. IOS Press, Amsterdam, 1993.

[6] G. Barrett. How to write a highly parallel program. In J.M. Kerridge, editor, *Transputer and* occam *Research : New Directions*, pages 209–217, Amsterdam, 1993. IOS Press.

[7] G. Barrett, E. Barton, T. Carden, D. Duval, and D. Nicole. General purpose parallel computers: a standard architecture with a standard programming interface. In A. Allen, editor, *Transputer Systems - ongoing Research*, pages 129–138, Amsterdam, 1992. IOS Press.

[8] G. Barrett, M. Goldsmith, G. Jones, and A. Kay. The meaning and implementation of PRI ALT in occam. In Charlie Askew, editor, occam *and the Transputer- Research and Applications OUG-9*, pages 37–46, Amsterdam, 1988. IOS Press.

[9] Geoff Barrett. occam3 reference manual. Technical report, INMOS Limited, Bristol, BS12 4SQ, England, March 1992.

[10] O. Botti and F. De Cindio. Comparison of occam program placements by a generalized stochastic petri net model. In M. Becker, L. Litzler, , and M. Trehel, editors, *Transputers '92 Advanced research and Industrial applications*, pages 65–85, Amsterdam, 1992. IOS Press.

[11] Juanito Camilleri. An operational semantics for occam. *International Journal of Parallel Programming*, 18(5), October 1989.

[12] Thomas Cheatham, Amr Fahmy, Dan C. Stefanescu, and Leslie G. Valiant. Bulk synchronous parallel computing — a paradigm for transportable software. In *Proceedings of the 28th Annual Hawaii International Conference on System Science*, January 1995.

[13] B.M. Cook. A fast C kernel for portable occam compilers. In *Proceedings of WoTUG-18: Transputer and* occam *Developments*, volume 44 of *Transputer and* occam *Engineering*, pages 47–65, Amsterdam, April 1995. IOS Press. ISBN 90 5199 222 x.

[14] David Culler, Richard Karpt, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Rarnesh Subrarnonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. Technical report, Computer Science Division, University of California, Berkeley, 1993.

[15] David E. Culler, Klaus Erik Schauser, and Thorsten von Eicken. Two fundamental limits on dataflow multiprocessing. In *Proceedings of the IFIP Working Group 10.3 (Concurrent Systems), Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism.* Elsevier Science Publishers, January 1993.

[16] Neil James Davies. *The performance and scalability of parallel systems.* PhD thesis, University of Bristol, December 1994.

[17] M. Debbage, M. Hill, S. Wykes, and D. Nicole. Southampton's portable occam compiler (SPOC). In Roger Miles and Alan Chalmers, editors, *Proceedings of WoTUG-17: Progress in Transputer and* occam *Research*, volume 38 of *Transputer and* occam *Engineering*, pages 40–55, Amsterdam, April 1994. IOS Press.

[18] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical report, Oak Ridge National Laboratory, March 1993.

[19] Craig C. Douglas, Timothy G. Mattson, and Martin H. Schultz. Parallel programming systems for workstation clusters. Technical report, Yale University, August 1993.

[20] C. Elamvazuthi and G.A. Manson. occam, PVM and the alternative construct. In Roger Miles and Alan Chalmers, editors, *Proceedings of WoTUG-17: Progress in Transputer and* occam *Research*, volume 38 of *Transputer and* occam *Engineering*, pages 56–68, Amsterdam, April 1994. IOS Press.

[21] S. Feit. *TCP/IP – Architecture, Protocols and Implementation.* McGraw Hill Series on Computer Communication. McGraw Hill, 1993.

[22] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[23] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, May 1994.

[24] Al Geist, Adam Beguelin, Jack Dongarra, and Weicheng Jiang. PVM3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[25] D. Goodeve. Mapping revisited. In J.M. Kerridge, editor, *Transputer and* occam *Research : New Directions*, pages 76–90, Amsterdam, 1993. IOS Press.

[26] Jane Hillston. *A compositional approach to performance modelling.* PhD thesis, University of Edinburgh, 1994.

[27] Dennis N.M. Ho. Variations of alt implementation on Transputer. In T. L. Kunii and D. May, editors, *Transputer/*occam *Japan 3*, pages 195–208, Amsterdam, 1990. IOS Press.

[28] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[29] I.A. Horton and S.J. Turner. A virtual architecture for investigating dynamic load balancing on Transputer networks. In A.S. Wagner, editor, *Transputer Research and Applications 3*, pages 263–274, Amsterdam, 1990. IOS Press.

[30] Chengchang Huang and Philip K. McKinley. Communication issues in parallel computing across ATM networks. Technical report, Communications Research Group, Michigan State University, June 1994.

[31] Geraint Jones. On guards. In *Parallel Programming of Transputer Based Machines.* IOS Press, September 1987.

[32] S B Jones and Goh Soon Liong. Load balancing evaluation for Transputer based systems. Technical Report TR1/044, SERC, 1988.

[33] S.W. Lau and F.C.M. Lau. An effcient and flexible implementation of ALT. In A.S. Wagner, editor, *Transputer Research and Applications 3*, pages 245–254, Amsterdam, 1990. IOS Press.

[34] INMOS Limited. occam2 *Reference Manual.* Prentice Hall, London, 1988. ISBN 0-13-629312-3.

[35] INMOS Limited. IMS C104 packet routing switch (preliminary data), June 1993.

[36] INMOS Limited. *T9000 Transputer Hardware Reference Manual.* SGS-Thomson Microelectronics, 1993.

[37] INMOS Limited. *T9000 Transputer Instruction Set Manual.* SGS-Thomson Microelectronics, 1993.

[38] INMOS Limited. *T9000 Transputer Development Systems Manuals: Hardware Configuration Manual*, May 1994.

[39] INMOS Limited. *T9000 Transputer Development Systems Manuals: Toolset Reference Manual*, May 1994.

[40] M.D. May, R.M. Shepherd, and P.W. Thompson. The T9000 communications architecture. In M.D. May, P.W. Thompson, and P.H. Welch, editors, *Networks, Routers and Transputers*, chapter 2, pages 15–38. IOS Press, Amsterdam, 1993.

[41] M.D. May and P.W. Thompson. Transputers and routers: Components for concurrent machines. In M.D. May, P.W. Thompson, and P.H. Welch, editors, *Networks, Routers and Transputers*, chapter 1, pages 1–14. IOS Press, Amsterdam, 1993.

[42] W F McColl. General purpose parallel computing. In A M Gibbons and P Spirakis, editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, Cambridge International Series on Parallel Computation, pages 337–391. Cambridge University Press, 1993.

[43] W F McColl. The BSP approach to architecture independent parallel programming. Technical report, Oxford University Computing Laboratory, December 1994.

[44] W F McColl. BSP programming. In G E Blelloch, K M Chandy, and S Jagannathan, editors, *Specification of Parallel Algorithms. Proc. DIMACS Workshop, Princeton, May 9-11, 1994*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 21–35, 1994.

[45] W F McColl. Scalable parallel computing: A grand unified theory and its practical development. In B Pehrson and I Simon, editors, *Proc. 13th IFIP World Computer Congress. Volume I (Invited Paper)*, pages 539–546. Elsevier, 1994.

[46] W. F. McColl. Bulk synchronous parallel computing. In John R. Davy and Peter M. Dew, editors, *Abstract Machine Models for Highly Parallel Parallel Computers.* Oxford Science Publications, Oxford University Press, 1995.

[47] Sun Microsystems. *SunOS 4.1.3 Network Programming Guide*, March 1990.

[48] Sun Microsystems. *Solaris 2.2 Network Interfaces Programmer's Guide*, May 1993.

[49] Sun Microsystems. *Solaris 2.2 System Services Manual*, May 1993.

[50] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[51] occam For All Group. KROC: Kent retargettable occam compiler, 1995. Available at `unix.hensa.ac.uk`.

[52] Michael Perloff and Kurt Reiss. Improvements to TCP performance in high-speed ATM networks. *Communications of the ACM*, 38(2):90–100, February 1995.

[53] D. M. Ritchie and K. Thompson. The UNIX timesharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[54] A W Roscoe. Denotational semantics for occam. In *Proceedings of the NSF/SERC Workshop on Concurrency*, volume 197 of *Springer Notes in Computer Science*, pages 1–25, July 1984.

[55] A W Roscoe and C A R Hoare. The laws of occam programming. Technical Report Technical Monograph PRG-53, Oxford University Computing Laboratory Programming Research, Group, February 1986.

[56] K.M. Shea and F.C.M. Lau. On the performance of ALT in occam. In A.S. Wagner, editor, *Transputer Research and Applications 3*, pages 285–294, Amsterdam, 1990. IOS Press.

[57] M. Simpson and P.W. Thompson. DS-Links and C104 routers. In M.D. May, P.W. Thompson, and P.H. Welch, editors, *Networks, Routers and Transputers*, chapter 3, pages 39–54. IOS Press, Amsterdam, 1993.

[58] W. Richard Stevens. *UNIX Network Programming.* Prentice-Hall, N.J., 1990.

[59] W. Richard Stevens. *Advanced programming in the UNIX environment.* Addison-Wesley Publishing Company, 1992.

[60] W. Richard Stevens. *TCP/IP illustrated, volume 1: the protocols.* Addison-Wesley Publishing Company, 1994.

[61] A. M. Tentner, R. N. Blomquist, T. R. Canfield, P. L. Garner, E. M. Gelbard, K. C. Gross, M. Minkoff, and R. A. Valentin. Advances in parallel computing for reactor analysis and safety. *Communications of the ACM*, 37(4), April 1994.

[62] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), August 1990.

[63] L. G. Valiant. General purpose parallel architectures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 943–971. Elsevier, 1990.

[64] Ronald J. Vetter. ATM concepts, architectures, and protocols. *Communications of the ACM*, 38(2):30–38, February 1995.

[65] Alf Wachsmann and Friedrich Wichmann. occam-light — a multiparadigm programming language for Transputer networks. Technical report, Universitat-GH Paderborn, 1992.

[66] S.W. Waithe and J.M. Kerridge. An appreciation of the subtleties of shared channels in occam3. In J.M. Kerridge, editor, *Transputer and* occam *Research : New Directions*, pages 232–245, Amsterdam, 1993. IOS Press.

[67] C. P. H. Walker. PCI 1355-01/2 PCI bus interface boards to IEEE 1355 DS-DE links product outline, 1995.

[68] P. H. Welch. An occam approach to Transputer engineering. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, Pasadena, California, U.S., January 1988.

[69] P. H. Welch. Shared-memory multi-processors and occam. Technical report, Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent CT2 7NF, U.K., 1992.