# Network performance in Arena, a customisable distributed operating system

K.R. Mayes[*], J. Bridgland, S. Quick and A. Nisbet

Centre for Novel Computing, Department of Computer Science,

University of Manchester, Oxford Road, Manchester, UK.

email: ken@cs.man.ac.uk; bridglaj@cs.man.ac.uk; stuck@cs.man.ac.uk; nisbeta@cs.man.ac.uk.

http://www.cs.man.ac.uk/cnc/arena. tel: +44 061 275 6135; fax:+44 061 275 6204

* presenting author

**Abstract**

This paper describes the provision of networking facilities in a customisable operating system called Arena. Customisable operating systems allow applications to determine their own resource management, so that an application can execute optimally on a particular architecture. On distributed architectures there are opportunities for tailoring message passing, virtual shared memory and distributed persistent store abstractions for particular applications. However, in all these cases, the efficiency of the network management is a critical determinant of performance. Arena seeks to balance flexible distributed resource management with efficient network handling. Arena provides an optimised form of network message transfer which avoids unnecessary copying, and allows the application control over the processing of the message. Performance figures are given for Arena network transfer on a distributed store multicomputer, and are compared with results from a microkernel-based operating system on the same hardware. Results show that there is no performance penalty associated with the flexibility of Arena.

## 1 Introduction

General purpose management of distributed resources may not be optimal for individual applications. Customisable systems seek to provide application-specific resource management. There are three basic approaches to building customisable

*

operating systems: customisable kernels (e.g. Mukherjee and Schwan, 1993), user-level servers (e.g. Gien, 1991) and user-level libraries (e.g. LeBlanc and Markatos, 1991). A major consideration in such systems, particularly where customisation is supported at user-level, is the issue of flexibility versus efficiency. This issue is seen particularly in distributed systems, where the performance of the software in handling network traffic is of prime importance. This paper describes the mechanisms used in the Arena customisable operating system (Mayes et al, 1994) to support efficient network handling on a distributed store multicomputer (Ward and Townsend, 1990).

# 2 Basic structure of Arena

The desirability of separating resource management *policy* from *mechanism*, so that applications can obtain optimal performance, has been recognised for many years (e.g. Wulf et al., 1974). Resource management *policy* in Arena is largely implemented at user-level. Resource managers are represented by interacting C++ objects that are *linked* to the application. The execution of this user-level resource manager code occurs via two routes.

1. By an application thread of control making a procedure call to a resource manager object interface routine to obtain a designated service, eg. creating a thread.

2. By an *upcall* from supervisor-level context to resource manager code in response to a hardware event. This supervisor-level context is part of the **hardware object** (HWO) described in the next paragraph. The Arena upcall is an asynchronous mechanism based on scheduling a particular thread from a set of event handler threads defined by the resource managers linked to the application.

Access to the *mechanisms* of the hardware is provided by the HWO – a thin layer of hardware-dependent code which provides a low-level abstraction of processor and store hardware (Quick, 1995). These HWO abstractions form an interface which can be implemented native on many processors, so that user-level resource management is portable across different processor types. At present the HWO has been ported to the Sparc and Intel 386 CPUs. Although the processor and store abstractions are fixed, the HWO interface is 'expandable' to allow system-specific *devices* to be configured. Thus, for example, a network component of the HWO is present only on a networked system. Similarly, a user-level Network Manager (NWM) is used only with an application that requires access to the network.

The initial target architecture for the distributed implementation of Arena is the European Declarative System prototype machine (EDSP). This is a distributed store multicomputer with an internal network connecting the processing

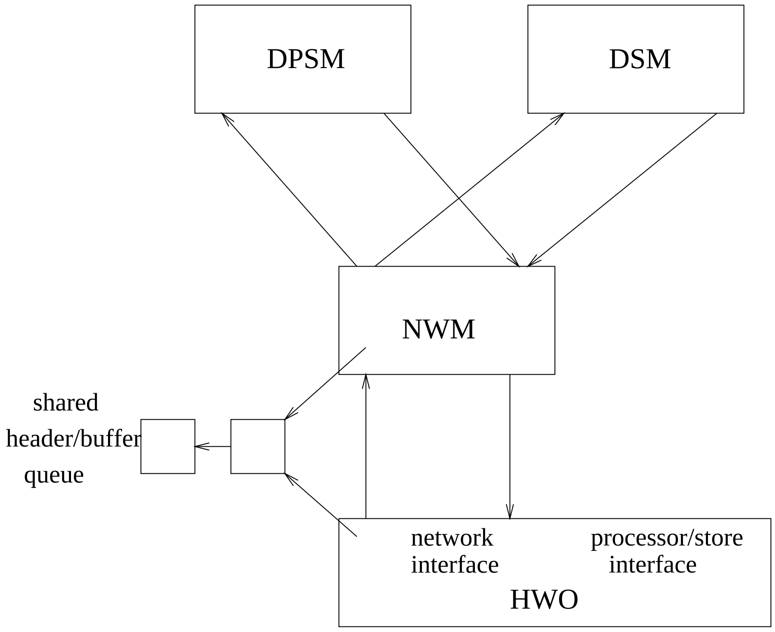


Figure 1: Levels in distributed resource management. On a networked architecture, the Hardware Object presents a network interface to user-level resource managers in addition to its processor and store interface. The user-level Network Manager has a queue of message buffers which is shared with the HWO supervisor-level network driver. Resource management of distributed resources (such as Distributed Persistent Store Management and Distributed Store Management) use the facilities of the Network Manager.

nodes, each of which has two Sparc processors sharing 64 Mb of nodal store. The EDSP networking interface includes a DMA capability for copying data between the network hardware and store. The maximum size of data that can be transmitted in one network packet is 128 bytes.

A single instance of the Arena HWO resides on each node, providing nodal resource management facilities as well as providing networking capabilities for communication between instances of distributed user-level managers. For example, distributed Persistent Store Manager (PSM) instances on different nodes will communicate via the user-level NWM which uses the network interface of the HWO.

# 3 Character I/O in Arena

On a distributed machine such as EDSP, character I/O occurs via the network. The model of character I/O in Arena is one of message buffers passing up and down a stack of modules whose routines can be executed asynchronously if required. These two aspects, message buffering and message processing are dealt with separately.

### 3.1 Buffering

A message buffer is referenced via a block of information (termed a *header*) which accompanies the buffer and carries message information such as application-defined message-type details, source and destination nodes. These headers are accessible in both supervisor and user mode, and are shared between HWO and the user-level NWM. When *sending* a message, headers are passed *downwards* from NWM to HWO as a procedure call parameter. That is, the NWM calls a network device driver-specific HWO interface routine. Conversely, when *receiving*, the HWO passes a buffer and its associated header *upwards* to the NWM by adding the header to a queue, shared with HWO, in NWM state (Fig. 1). The HWO instigates an *upcall* which may result in running the event hander thread defined by the NWM. This thread takes the headers and message buffers from the shared queue for processing at user-level.

Arena provides an optimised transfer, where intermediate copying from message buffer to application buffer is avoided. That is, the application buffer is itself used directly by the network driver and accompanies the header associated with the message. This mode is particularly appropriate for transferring disk blocks or store pages across a network. However, before an optimised transfer can be initiated, certain conditions must be satisfied. The buffer containing the data to be sent must be fixed into physical store, and must obey hardware-dependent alignment and size restrictions.

### 3.2 Processing

Hardware events are conventionally handled synchronously by supervisor-level routines accessed via the trap table. It is, however, useful to defer non-essential processing of an event. In Arena a hardware event may cause an upcall to be made into user-level, to allow deferred processing of the event by an event handler thread. The policy component of *how* to handle the event is determined by the application-specific code executed by the event handler thread. Furthermore, such event handler threads are explicitly scheduled by the normal user-level Process Manger (PM) scheduler code, so that the application can also make policy decisions about *when* to service deferred events. Despite the flexibility inherent in this approach, allowing applications to tailor the responses to events, inefficiency may arise from the need to make a context switch to the handler thread. It is thus important to choose an appropriate point to service the deferred handling.

## 4 Performance results and discussion

The performance figures for transferring data between two nodes of the EDSP machine are given in Table 1. Figures are given for Arena and for a microkernel

(Ward and Townsend, 1990). The value for the microkernel has been obtained using an *RPCSend()/RPCReply()* pair. With the Arena optimised transfer scheme, Arena performance is faster than that of the EDSP microkernel. In both systems, this time represents a user-level to user-level transfer of data. Transfer in Arena involves the scheduling of the network event handler thread at both nodes involved in the transfer. The time also includes a rescheduling of the application thread which originally made the transfer request, and which has received the requested data. The microkernel behaviour similarly involves scheduling threads at user-level. Both systems use user-level message buffers in the transfer with no intermediate copying. Whereas Arena is executed only on one of the processors of each node, the microkernel partitions processing across both processors, running network code exclusively on one processor. However, in the test conditions, little difference in behaviour would be expected. The main point to note is that the cost of running the Arena user-level event handler threads does not represent a significant overhead, compared to the figures for the microkernel. It may thus be argued that the flexibility afforded by the Arena upcall mechanism does not carry too high a cost.

With optimised network transfer in Arena, a user-level buffer is provided by the application and this is used by the device driver to avoid a level of copying. Incoming messages are copied to this buffer until the driver has received the designated size for the transfer, indicated by receipt of an end-of-transfer packet. Upon receiving the last data packet the device driver enqueues the buffer and its associated header on the queue it shares with the NWM. The device driver then initiates an upcall. Thus the handler thread runs once only, and may simply schedule the waiting application without any further copying of transferred data. To support distributed persistent store management, the unit of transfer may be page-sized, in this case 4Kb. Although this size of data unit could support distributed store management, false sharing could result. Virtual shared memory implementations may thus require smaller units of data transfer. The value for 128 bytes is the worst-case figure, where the overhead-per-byte transferred is greatest.

| Operating System | Data Transfer Time | |
|---|---|---|
| | 4 Kbytes | 128 bytes |
| EDSP microkernel | 1425 | 792 |
| Arena | 527 | 281 |

Table 1: Performance figures (microseconds) for transferring 4kb and 128 bytes of data between two nodes of the EDSP machine using two operating systems. Each transfer involves user-level processing.

# 5 Comparisons with other work

The approach described here is roughly based on the Streams approach of AT&T versions of Unix for handling character I/O (Ritchie, 1984). Streams allows deferred servicing of messages. Streams processing is done at supervisor-level, the execution context for the postponed I/O processing is provided by any Unix process that is returning to user-mode. In contrast, in the Arena approach, deferred processing is by specific user-level handler threads which are explicitly scheduled by the application PM, so that deferred servicing occurs in a predictable fashion. Solaris2 has preallocated kernel threads which handle interrupts, with a separate list for each processor (Kleiman and Eykholt, 1995). In Arena, threads are similarly pre-allocated, but are application-specific and reside and execute at user-level.

Several systems provide an upcall mechanism for executing code at user-level in response to some sort of event. The 'software interrupt' of Psyche (Marsh et al., 1991) was executed in the context of a kernel thread, after switching the stack and program counter, and entering user-level. In the thread-oriented upcall mechanism described by Anderson et al. (1992) the kernel created and scheduled a new kernel thread whose entry point was the required upcall code of a user-level threads package. In Raven (Ritchie and Neufeld, 1993) a task registered its interrupt handler code with the kernel. When the corresponding interrupt occurred, the kernel located the user-level task registered to handle the interrupt, possibly involving a task-switch. The upcall consisted of executing the user-level task upcall dispatcher which *called* the appropriate interrupt handler code within the task. Arena has an upcall mechanism based on loading the application-specific user-level PM scheduler code which is then responsible for scheduling dedicated threads. Arena threads are purely user-level threads (Mayes et al., 1995).

It is possible to access network hardware at user level, for example by mapping the device registers into a user address space (Ritchie and Neufeld, 1993). The security problems presented by this can be overcome by partitioning the processors of a machine, with hardware validation of message destination and gang scheduling of applications within a partition (Felten, 1992). Placing handling at user-level allows application-specific communications protocols (Felten, 1992) and removes the need to copy between kernel and user buffers (Ritchie and Neufeld, 1993). However, it also is possible for user-level message buffers to be used in a microkernel system running in supervisor mode (Ward and Townsend, 1990). The Arena approach seeks to combine security, flexibility and efficiency in networking by executing device driver code at supervisor-level, allowing policy to be implemented as user-level handler threads, and providing a means of avoiding copying.

## 6 Acknowledgements

The authors would like to thank Brian Warboys, John Gurd, and all the members of the Centre for Novel Computing. This work is supported by EPSRC grants GR/J 84045, 93315512 and 91309499.

## 7 References

**Anderson, T.E., H.M. Levy, B.N. Bershad and E.D. Lazowska** (1991) The interaction of architecture and operating system design. Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems, 108-120.

**Felten, E.W. (1992)** The case for application-specific communication protocols. Technical Report 92-03-11, Department of Computer Science and Engineering, University of Washington.

**Gien, M. (1991)** Next generation operating systems architecture. LNCS 563, 227-232.

**Kleiman, S. and J. Eykholt (1995)** Interrupts as Threads. Operating Sytems Review, ACM SIGOPS 29 (2), 21-26.

**LeBlanc, T.J. and E. P. Markatos (1991)** Operating system support for adaptable real-time systems. University of Rochester Computer Science and Engineering 1990-1991 Research Review, 14-20.

**Marsh, B.D., M.L. Scott, T.J. LeBlanc and E.P. Markatos** (1991) First class user-level threads. Operating Systems Review 25(5), 110-121. **Mayes, K., S. Quick, J. Bridgland and A. Nisbet (1994)** Language- and application-oriented resource management for parallel architectures. Proceedings of the 6th ACM SIGOPS European Workshop, September, 172-177.

**Mayes, K., S. Quick and B.C. Warboys (1995)** User-level threads on a general hardware interface. ACM SIGOPS Operating Systems Review 29(4), 57-62.

**Mukherjee, B. and K. Schwan (1993)** Experimentation with a reconfigurable microkernel. Usenix Association Proc. Symp. on Microkernels and other Kernel Architectures (Sept), 45-60.

**Quick, S. (1995)** A generalised hardware interface for operating systems. PhD Thesis, University of Manchester (In Preparation).

**Ritchie, D. (1984)** A stream input-output system. AT&T Bell Laboratories Technical Journal 63 (8), 1577-1593.

**Ritchie, D.S. and G.W. Neufeld (1993)** User level IPC and device management in the Raven kernel. Usenix Association Proc. Symp. on Microkernels and other Kernel Architectures (Sept), 111-125.

**Ward, M. and P. Townsend (1990)** EDS hardware architecture. LNCS 457, 816-827.

**Wulf, W., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and**

**F. Pollack (1974)** HYDRA: The kernel of a multiprocessor operating system. Comms ACM 17(6), 337-345.