Arena – a Run-Time Operating System for Parallel Applications

K.R. Mayes and J. Bridgland
Centre for Novel Computing,
Department of Computer Science,
University of Manchester,
Oxford Road, Manchester, UK
ken@cs.man.ac.uk
bridglaj@cs.man.ac.uk

Abstract

This paper presents the case for a run-time operating system *to provide predictable and controllable resource management for parallel applications. Developments in compilers for parallel applications, which depend on analysis of program behaviour, require corresponding developments in predictable and controllable operating system-level resource management. A customisable system, Arena, is described which provides operating system-level resource management at user-level, where it is accessible to application and run-time system developers. This resource management is accessed via libraries, and effectively becomes part of the application run-time system. In such a system the application gets only the resource management that it needs. A low-policy hardware-dependent executive provides a low-level interface presenting an abstraction of processor hardware to the hardware-independent resource managers. The use of Arena on a Sparc-based distributed store multicomputer is described, and three parallel application areas being investigated on Arena are briefly discussed.*

1. Introduction

This paper discusses using a *run-time operating system* to achieve optimal performance for parallel applications.

A variety of run-time systems exist which provide, for example, a message-passing interface or a virtual shared memory interface to parallel applications. This paper is not about the debate between advocates of different abstract machine paradigms, but is about how these machines should be implemented on high performance architectures. The point is that these run-time systems rely on the native operating system to provide resources such as servers and communication primitives. Indeed, the purpose of such run-time systems is to map the parallel machine required by the application on to the machine actually provided by the operating system. Inefficiencies arise where there are incompatibilities between these two levels of machine, and where compromises have to be made in the run-time system implementation. That is, policy embedded by the designer of the operating system interface may have to be avoided by reimplementation at the run-time library level. Furthermore, general-purpose operating systems may introduce unpredictability into the execution of an application. The system may determine when housekeeping daemons run, for example, or when deferred I/O processing occurs, without reference to the requirements of the application.

Developments in compilers for parallel applications, which depend on analysis of program behaviour, require corresponding developments in predictable and controllable operating system-level resource management.

One way around these problems is to link the application with only that resource management it requires, load this object onto the *bare hardware*, and let it run. This is the situation found in many real-time applications. Mukherjee and co-workers described such real-time systems using the term ‘operating software’ in order to emphasise the intimate connection between real-time application and real-time operating system [20]. However, it is unreasonable to expect parallel application developers to write their own low-level, hardware-specific code.

This paper describes a solution to this problem – to provide a generic ‘hardware’ interface which can reside on the nodes of the parallel machine and which can host the run-time operating system. Arena is a system being developed which takes this approach [16]. It emphasises individual application performance (though it is designed to allow multiple applications to run ‘concurrently’ if required).

2. Background

It has been argued (e.g. [12]) that operating systems for multicomputers should present an abstraction, a 'single system image', to hide the physical machine architecture from users. However, not all users of parallel architectures require such abstractions.

For example, Bryant and co-workers described their experience with Fortran users of the IBM RP3 system [5]. These users regarded the operating system "as an adversary bent on denying them direct access to the hardware". Similarly, developers of data management and persistent object stores need sophisticated use of caching, and allowing the application access to file-caching policies has been found to improve performance [6]. Many existing parallel language implementations multiplex their own process structure onto a smaller set of operating system-provided processes, where the operating system-provided process may run a user-level scheduler. There are many problems associated with building a language on a conventional operating system platform due to mismatches between operating system process model and the requirements of the language [22].

The problem then, for high performance systems, is how to empower the application writer and language developer so that they can safely access the hardware mechanisms, and control the policy of resource management which is traditionally locked into the operating system kernel.

The key here is to separate operating system policy from mechanism. The desirability of separating policy from mechanism has been recognised for some time, and was a motivating force in the design of the Hydra operating system [26]. A trend in operating system work has been to move policy out of the supervisor-mode kernel and into the realm of the user. That is, to put policy at *user-level*. The implication is that these policy routines can be provided by the *users* of the computer system themselves, rather than by the systems programmers who created the operating system kernel. These routines, perhaps generated by language system implementors, will become encapsulated into libraries and accessed by applications in association with the language run-time libraries.

3. Arena

Moving all policy into user-level resource manager libraries effectively extends the run-time system into the operating system. Figure 1 shows schematic diagrams comparing a conventional monolithic operating system with the approach taken in Arena. All the hardware-dependent code for manipulating the hardware resides in a 'nanokernel', or run-time executive, which is accessed by a generic low-level interface. Having a general low-level interface facilitates system portability. Arena has been implemented on the EDS [25] Sparc and PC-AT Intel 3/486 architectures.

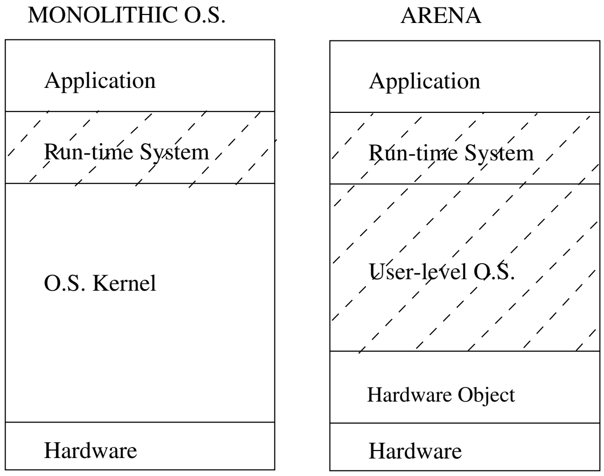

Figure 1. A schematic comparison of approaches to operating systems. Library-based components are shaded.

3.1. Arena structure

The basic structure of Arena is shown in Figure 2. There are basically two components. One component consists of *hardware-independent* resource manager libraries for linking to application. These represent a run-time operating system. The other component is the *hardware-dependent* code. The run-time operating system gains access to hardware-dependent code via an abstract interface. This interface is called the Hardware Object (HWO) interface. The main principle in this division is to distinguish between policy and mechanism. The HWO is intended to give access to the mechanisms provided by the hardware. The resource manager libraries determine the policy in the use of these mechanisms.

3.2. Hardware Object

The HWO is essentially a run-time executive which provides a primitive machine interface to the resource managers. This interface presents a generalised view of processor hardware, so that porting of the system to various hardware platforms is facilitated [23]. The HWO interface provides a set of *downcalls* which is used by the hardware-independent, user-level, manager code. There are, at present, 34 primitives in the HWO interface (Figure 2), not including any device-specific interface primitives. This structure represents an instance of the system that would reside on each node of a distributed-store machine. Note that the trap interface is internal to the HWO.

This HWO interface is accessed via a library. The HWO code is implemented at user-level whenever possible. However, for protection, and access to supervisor-mode instructions/registers, some library routines consist of a stub which traps to supervisor and accesses the code conventionally via the trap table.

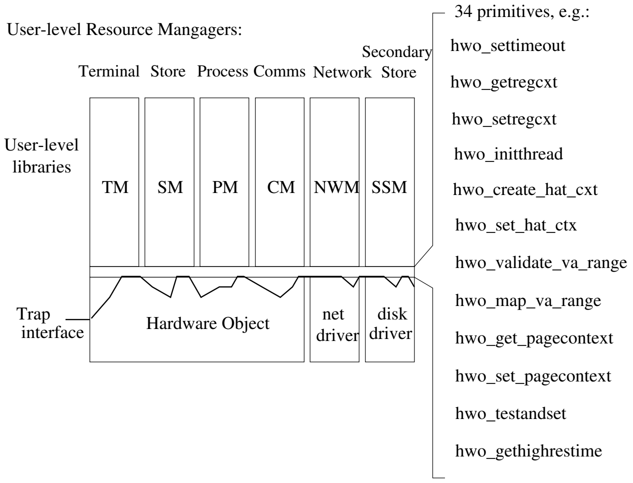

Figure 2. Structure of the Arena system.

The HWO interface gives an abstraction of processor and store which can be implemented on any processor with memory management hardware. The HWO deals mainly with low-level concerns such as general *register contexts* and *address translation*. By providing interface primitives for manipulating register contexts separately from address translation, the HWO interface allows the policy of associating threads of control with address spaces to be determined at user-level. The HWO also provides the concept of a *va_range*, a range of virtual addresses which can be validly accessed in a particular hardware address translation context. The HWO also supports the concept of a *page context* for user-level handling of page faults, for example in the implementation of virtual shared memory (VSM). The *page context* is analogous to the *register context*. Both are hardware-dependent data types enabling the user-level to save and restore aspects of the hardware state, in a hardware-independent manner. The HWO internals are protected from user-level access by page access rights. The HWO imposes protection between applications by forbidding access to hardware address translation contexts which are not owned by an application.

Interfaces to devices such as disks and networks are an extension of the HWO interface. The reasoning here is that different systems will have different devices which may, in turn, require different interfaces. Typical disk (secondary store) interfaces are *hwo_ssm_open()* to establish rights to access certain disk blocks and *hwo_ssm_io()* to read or write to a disk block. A message is sent to a network device via a call to *hwo_cio_netsend()*.

3.3. User-level managers

Arena resource management components are decoupled, interacting via well-defined interfaces, and implemented in C++. Resource manager base classes are defined for Process Manager (PM), Store Manager (SM), Secondary Store Manager (SSM), Communication Manager (CM), Network Manager (NWM) and Terminal Manager (TM). Only those managers required by an application need be used. For example, a single-node application needs no NWM, whereas only applications doing disk I/O will need an SSM.

The interface provided by each of these managers has three components:

1. A set of operations used by application code. These operations are at the level of a typical microkernel interface. They serve to map conventional concepts on to the low-level HWO interface. The PM provides operations for manipulating and switching *threads* and *address spaces*. Threads in Arena are entirely user-level constructs. The HWO has no concept of a *thread*. The HWO only provides the concept of a *register context*. The SM operations allow *regions* to be created and mapped. The CM provides message passing via the familiar *port* abstraction.

2. A set of operations used by other managers. These operations allow managers to interact. For example, the PM provides interfaces for the SSM to stop and resume threads which are involved disk I/O. The SM operations allow managers to allocate regions: the PM requires regions for thread stacks; the SSM requires regions for buffer caches. The PM uses an SM operation to obtain a new address space. The NWM provides operations to allow other managers to exchange messages, upwards and downwards, with the network device interface of the HWO.

3. A set of operations used by the HWO. Such operations allow the manager to do deferred handling of events at user-level. These routines are executed by distinct *event threads* (see section 3.4).

Arena is a customisable system. Resource manager subclasses allow instantiation of a resource manager for particular policies. An application links to the managers providing the required resource management policies. For example, different PMs exist for cooperative, priority-based and round-robin scheduling of threads. Different SSMs can be selected for particular caching policies. Relinking to the application and other manager libraries is facilitated by using base-class pointers.

3.4. Arena event handling

Hardware events, such as disk and network interrupts, and page faults, can be handled at user-level. The mechanism is shown in Figure 3. Events can be handled by application-specific event threads. These threads are scheduled by the user-level PM scheduler code. Thus the application can determine both when and how events are handled.

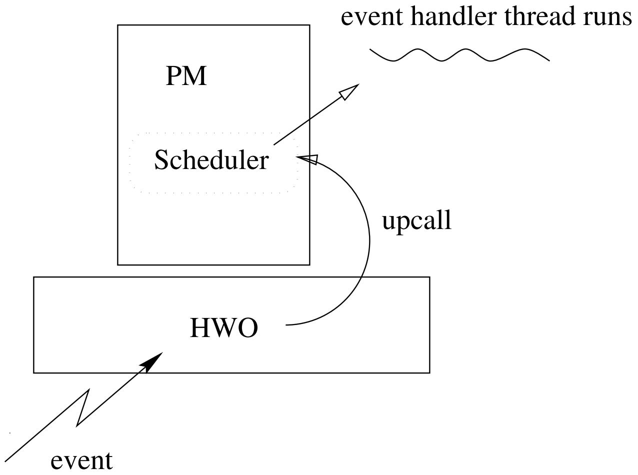

Figure 3. User-level event handling. Deferred handling of events can occur under control of the application.

As will be discussed in section 5.3, the use of user-level code to handle page faults allows the implementation of application-specific VSM policies on parallel architectures.

4. Parallel Arena

Arena is specifically intended to run in parallel on both shared memory and distributed memory architectures (i.e both multiprocessors and multicomputers). The conventional approach to operating system architecture is different on these two types of hardware platform. With multiprocessors, there is generally a single kernel instance running on all processors, and shared data structures are protected by locks. Arena has been implemented for execution on multiprocessor platforms, although this aspect is not emphasised, and has not yet been tested.

Arena does however already run on a distributed store multicomputer. The EDS machine [25] has nodes connected by an internal deltanet. Each node consists of two Sparc processors sharing 64 Mb of store. Only one of these Sparcs receives the network interrupts, and at present Arena runs only on that processor on each node. When running an application, each multicomputer node has its own local HWO and instances of resource managers. Such replication serves to maximise local activity and minimise global activity. No attempt is made to share HWO data structures across distributed store, so there is no contention for HWO resources between nodes.

Where an application is distributed, a set of the resource manager objects for that application occurs on each node on which the application is running. Communication between nodal components of the application resource management occurs via the NWM object instances, mediated via the HWO and network driver, on each node (Figure 4). Network performance in Arena, for user-level to user-level transfer of data, was measured to be faster than a microkernel running on the EDS machine [15]. Both systems used user-level message buffers in the transfer, with no intermediate copying.

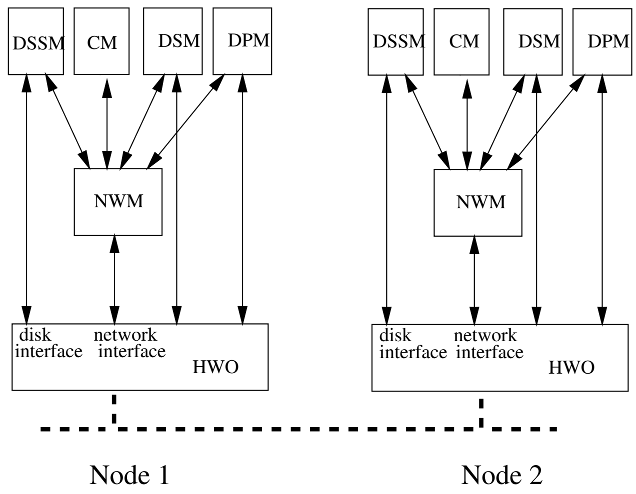

Figure 4. Resource management for a distributed application.

5. Target application areas

In order to demonstrate the usefulness of the Arena approach, several applications are being investigated.

5.1. Requiem database

The Requiem relational database [21] has been ported to Arena. It runs on both EDS and PC platforms. The database is the target application for work on secondary storage management [4]. Requiem runs on a Unix system call interface. This is provided on Arena by using a library of Unix system calls, implemented at user-level on top of the SSM interface. Although Requiem is not a parallel database, it is being used to investigate distributed file system implementation on the EDS machine. The Arena secondary store is implemented on the EDS machine using RAM disks. The EDS machine does not have physical disks attached to its nodes. Thus the effects of caching on secondary store performance with physical disks are being investigated on a PC-compatible Intel 386 machine. This allows comparisons between performances of database transactions on Requiem running on Linux and on Arena. Initial results indicate that performance of the two systems is similar using a flush-back policy. However with a more realistic write-through policy, Arena out-performs Linux by a factor of 1.6.

5.2. UFO language

Arena is intended to support parallel languages. Investigations into running the parallel UFO language [24] on Arena are in progress. Current work is using a distributed PM and CM to investigate a multi-nodal implementation on the EDS machine. The distributed PM allows remote creation of threads. Arena permits the run-time system implementor to rewrite the existing, more general-purpose, CM to achieve performance enhancements, where the developer has knowledge of language run-time system behaviour. More 'kernelised' general-purpose systems would not facilitate this.

5.3. Virtual shared memory for Fortran

Work has been done on parallel Fortran using user-level VSM on a microkernel-based system running on an EDS machine [9]. It is intended to use the accessibility of the mechanisms provided by Arena to implement the primitives of a VSM framework [10] which will be incorporated into a Fortran compiler.

As mentioned in Section 3.4, the Arena page fault handling thread can implement VSM. VSM implementation is based is the transferring of a virtual page between nodes, over a network, in response to a page fault in a virtually-shared region. This mechanism has been implemented and early results indicate that Arena is appreciably faster than a microkernel on EDS (by a factor of 6). However, this initial Arena implementation does not include the directory processing carried out by the microkernel.

6. Related work

There are two ways in which user-level policy can be implemented: by providing a user-level *server process* or by providing a *library* of routines. Existing microkernels such as Mach and Chorus support user-provision of store, file and network management by user-level servers (e.g. [7] [11]). Microkernel systems can locate code in servers, and use an IPC mechanism to bind applications to that code. Although this allows some redefinition of management code outside the kernel, it requires process management and IPC support in the kernel. It has been argued that such systems provide insufficient flexibility for library and language implementors [18] [13].

Work on Psyche [14] and on scheduler activations [1] take thread scheduling out of the kernel by enabling the kernel to make 'upcalls' to the user level. Threads on Arena exist only at user-level. There are no kernel threads, and no permanent kernel state for user-level threads [17]. This means that all scheduling policy decisions are in the user-level PM, in the realm of the application.

More recent systems, which are similar to Arena in emphasising user-level application-specific resource management, are PANDA [2] and Exokernel [8]. This latter paper coined the term 'library operating system' to describe such systems.

An alternative approach to application-orientated resource management is to allow the operating system kernel to be tailored [19] [3].

7. Discussion

Users of high performance parallel computers may wish to develop and link their applications directly on some node of the target machine (though this would seem to be unnecessary with a cross-compiler). For this application development, the machine may be required to provide a familiar general-purpose operating system environment. However, it seems unreasonable to force the application to *execute* using general-purpose resource management facilities. It is suggested here that high performance applications should be loaded onto parallel machines and execute with appropriate resource management policies. That is, that the operating system should conform to the application.

8. Acknowledgements

The authors would like to thank Professor Brian Warboys, Rupert Ford, Andy Nisbet and other members of the CNC. Stuart Quick was responsible for the hardware object implementations. Daniel Pollard of the UFO group is investigating UFO on Arena. This work is supported by EPSRC grants grants GR/J 84045, 93315512 and 91309499.

References

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *tocs*, 10(1):53–79, 1992.

[2] H. Assenmacher, T. Breitbach, P. Buhler, V. Hubsch, H. Peine, and R. Schwarz. Meeting the application in user space. In *Proceeding of the 6th ACM SIGOPS European Workshop*, pages 82–87, September 1994.

[3] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance int the spin operating system. In *Proceeding of 15th ACM Symposium on Operating System Principles*, pages 267–284, 1995.

[4] J. Bridgland. Secondary store management in a customisable operating system. 1996. In Preparation.

[5] R. Bryant, H. Chang, and B. Rosenburg. Experience developing the rp3 operating system. In *Proceedings of Usenix Association Distributed and Multiprocessor Systems*, pages 1–18, Summer 1995.

[6] P. Cao, E. Felten, and K. Li. Application-controlled file caching policies. In *Proceedings of Usenix Association Conference*, pages 171–182, June 1994.

[7] R. Draves. A revised ipc interface. In *Proceedings of Usenix Mach Symposium*, pages 101–121, 1990.

[8] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceeding of 15th ACM Symposium on Operating System Principles*, pages 251–266, 1995.

[9] R. Ford, A. Nisbet, and J. Bull. User-level vsm optimization and its application. *Proceedings of 2nd International Workshop on Applied Parallel Computing. Lecture Notes in Computer Science*, 1041:223–232, 1995.

[10] R. Ford, A. Nisbet, and M. O'Boyle. A new hybrid coherence mechanism: compilation and application. 1996. In Preparation.

[11] M. Guillemont, J. Lipkis, D. Orr, and M. Rozzier. A second-generation micro-kernel based unix; lessons in performance and compatibility. In *Proceedings of Usenix Association Conference*, pages 13–21, Winter 1991.

[12] B. Herrmann, M. Ortega, and L. Philippe. Unix on a multicomputer: The benefits of the chorus architecture. Technical Report CS/TR-91-46, Chorus Systems, 1991.

[13] E. Lazowska. System support for high performance multiprocessing. In *Proceeding of the Usenix Association Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 1–11, March 1992.

[14] B. Marsh, M. Scott, T. LeBlanc, and E. Markatos. First-class user-level threads. In *Proceeding of 13th ACM Symposium on Operating System Principles*, pages 110–121, 1991.

[15] K. Mayes, J. Bridgland, S. Quick, and A. Nisbet. Network performance in arena. *Proceedings of High Performance Computing and Networking, Europe. Lecture Notes in Computer Science*, 1067:1007–1008, 1996.

[16] K. Mayes, S. Quick, J. Bridgland, and A. Nisbet. Language- and application-oriented resource management for parallel architectures. In *Proceeding of the 6th ACM SIGOPS European Workshop*, pages 172–177, September 1994.

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

[18] T. L. M.L. Scott and and B. Marsh. Multi-model parallel programming in psyche. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 70–78, March 1990.

[19] B. Mukherjee and K. Schwan. Experimentation with a reconfigurable microkernel. In *Proceedings of the Usenix Association Symposium on Microkernels and other Kernel Architectures*, pages 45–60, September 1993.

[20] B. Mukherjee, K. Schwan, and P. Gopinath. A survey of multiprocessor operating system kernels. Technical report, Georgia Institute of Technology, College of Computing, 1993.

[21] M. Papazoglou and W. Valder. *Relational Database Management – A Systems Programming Approach*. Prentice Hall, 1989.

[22] D. Pierson. Integrating parallel lisp with modern unix-based operating systems. *Lecture Notes in Computer Science*, 441:312–315, 1989.

[23] S. Quick. *A generalised hardware interface for operating systems*. PhD thesis, Department of Computer Science, 1996.

[24] J. Sargeant. Uniting functional and object-oriented programming. *International Symposium on Object Technologies for Advanced Software. Lecture Notes in Computer Science*, 742:1–26, 1993.

[25] M. Ward and P. Townsend. Eds hardware architecture. *Lecture Notes in Computer Science*, 457:816–827, 1990.

[26] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *cacm*, 17(6):337–345, 1974.