Language- and application-oriented resource management for parallel architectures

Ken Mayes, Stuart Quick, James Bridgland and Andy Nisbet

Centre for Novel Computing, Department of Computer Science,
University of Manchester, Oxford Road, Manchester, UK.
Contact – email: ken@cs.man.ac.uk; tel: +44 061 275 6135; fax: +44 061 275 6204

1 Introduction

General-purpose operating systems, such as UNIX, which evolved on single processor machines, have made the transition, in one form or another, to parallel architectures (e.g. Rothnie, 1992; Holman, 1992). However, it is not clear that all users of parallel architectures require the virtual machine presented by a general-purpose operating system (e.g. Bryant et al., 1991). It is unfortunate if such users are given the alternatives of either compromising with whatever operating system interface is available, or writing all the low-level routines for themselves. One solution to this problem is to provide customisable systems, so that for high performance, parallel applications acquire a tailored resource management environment (e.g. Mukherjee and Schwan, 1993). This paper first gives some background to applications and operating systems, and then describes a flexible and extensible system currently being developed.

2 Operating systems and applications

Of the major areas in research into operating systems for parallel architectures, one is concerned with providing an abstract, more general, view of distributed resource management, and another is concerned with providing customised, or customis*able*, resource management.

In the first approach, it has been argued (e.g. Schlichtiger, 1991) that operating systems for distributed machines should present an abstraction, a 'single system image', to hide the physical machine architecture from users. This abstraction, like others such as virtual shared memory (or indeed language-based parallel virtual machines) will suit many users. Such virtual machines will enable them, for example, to port existing applications quickly (e.g. Keane and Xu, 1992).

On the other hand, the customisation approach seeks to allow an application to determine precisely its resource management policies. This approach can be implemented by providing resource management which is customisable at some level in the system (e.g. Bershad et al., 1988; Philbin, 1992; Mukherjee et al., 1993). Alternatively, operating systems designers can work in collaboration with users to provide facilities for specific application or language requirements (e.g. Watson and Townsend, 1990; Bryant et al. 1991; Bodin and Priol, 1992).

The general-purpose and customisation approaches are not mutually exclusive: for example, while both the Amoeba and Chorus systems seek to provide a single system image, they allow user-level provision of resource management servers to facilitate customisation (Tanenbaum et al., 1990; Herrmann et al., 1991; Gien, 1991). Furthermore, the two approaches can be complementary: for example, with appropriate low-level monitoring facilities (and both understanding of, and control over, machine behaviour) virtual shared memory was found to provide a good route to efficient parallel numeric implementations (Bull and Riley, 1994). This latter approach is perhaps analogous to compilation of high-level language code which is tunable at assembler-level.

The customisable resource management paradigm can also be contrasted with the more monolithic general-purpose operating systems in that the latter, to varying extents, bundle *mechanism* with *policy*. The requirements on the operating system abstractions vary with the richness of languages (Weiser et al., 1989), so that it is difficult for a single set of abstractions to support all languages. General-purpose operating system interface primitives may therefore not support efficient implementations for all language run-time systems and their applications. The answer may be to 'unbundle' the interface primitives so that higher levels of software can 'rebundle' them as required for specific applications.

Despite their imposition of some resource management policy, general purpose operating systems do provide for code reuse, via the system call interface, relieving the application programmer and language implementor of the burden of writing low-level routines. An alternative mechanism is to access system services via inter-process communication to a server process, as seen in the microkernel architectures. In the proxy objects described by Black et al. (1992) server code is loaded into the client application address space. This converges with the other approach to the reuse of operating system code; that is, linking to specific routines which include code necessary only for the application. This direct linking approach is seen in language systems which provide run-time libraries, and with real-time 'operating software' (Mukherjee et al., 1993) where the application and resource management routines are intimately combined. Recent systems seek to maximise the user-level provision of policy in libraries for language and real-time support (e.g. LeBlanc and Markatos, 1991; Ritchie and Neufeld, 1993). Systems allow application-oriented customisation both at kernel level (e.g. Mukherjee and Schwan, 1993) and at user-level (e.g. Bershad et al., 1988). Placing customisation at user-level allows the possibility of benefiting from the improved performance described (e.g. in Anderson et al., 1992, and Lazowska, 1992) as being associated with user-level mechanisms.

A system which supports such customisation should be flexible and extensible. Policy may be provided at user-level on top of a small, largely supervisor-level, executive. The application run-time system is effectively extended to incorporate operating system resource management library routines. Such an application-oriented system is being designed and developed at the Centre for Novel Computing at the University of Manchester[1] (Mayes, 1993).

3 An application-oriented system

The design of the run-time executive and resource management components of the system being developed is based on the general approach taken by the Flagship project system software (Leunig, 1987; Mayes and Keane, 1993). In the Flagship project, the low-level interface of a graph reduction machine was implemented on conventional hardware as a 'hardware ADT' instance, and resource management was represented as interacting manager ADT instances. These ADT instances were not 'active', and their interface operations were invoked via procedure calls.

Within this basic approach, the primary aim is to 'unbundle' the resource management primitives at each level, so that they can be 'rebundled' at higher levels in the system in order to allow applications to precisely tailor their own execution environments.

At the base level of software, a 'hardware' abstract class provides an abstraction of the low-level features of the processor hardware: register contexts; virtual addressing contexts; physical store. This executive provides abstract operations, or downcalls, used by the hardware-independent resource management objects, to access the hardware. Particular hardware objects are instantiations of the hardware abstract class for specific hardware. Although much of the hardware object runs in supervisor mode, the trap interface is internal to the hardware object; only those operations requiring supervisor-level activity will involve a trap-to-supervisor.

User-level resource manager objects make use of the hardware object interface to control the machine. For example, a Process Manager will call a *getRegisterContext()/setRegisterContext()* pair in order to implement a thread switch. All thread state and scheduling resides at user-level[2]. Whether a thread switch involves a trap-to-supervisor depends solely on the kind of access to thread-of-control state allowed by the processor.

[1]This work is supported by EPSRC grants GR/J 84045, 93315512 and 91309499.

[2]Low-level trap handling may require that some state relating to a thread is held temporarily in the hardware object.

As noted above, the system is application-oriented, and so the unit of work supported by the hardware object is a single application. Here, a single application may, for example, be multi-threaded, use multiple address spaces, and be distributed over several nodes. Such a unit of work is termed a 'computation'. A computation is represented at user-level by a set of resource manager objects. The policies implemented by the resource managers of the computation cooperate with the application code and its run-time system to entirely determine the behaviour of the computation. The interface of each manager has three components: an upper interface component for use by the application or language runtime system; a lateral interface component for use by other managers; and a lower interface component for use by the hardware object. The hardware object maintains state associated with a computation, basically a set of references to the managers of the computation, which logically provide a set of entry points for user-level event handling. Events can be handled as user-level threads. A low-level handler provided by the hardware object can arrange for the appropriate event thread to be 'runnable' before returning to user-mode after an event has occurred. Subsequent scheduling of the event thread is entirely the responsibility of the user-level Process Manager. In this way a Process Manager can implement, for example, preemptive time-slicing using a particular timer interrupt handler thread, and a Store Manger can implement virtual shared memory. In the latter case, where a computation is distributed over several nodes of a distributed-store machine, the distributed computation has a set of inter-communicating manager instances on each node on which it resides. The system is intended to be highly flexible and extensible to allow resource management to conform to application requirements, and to allow an incremental approach to implementation.

3.1 Flexibility of policy

For optimal performance, some applications may require resource management, such as scheduling policy or store coherency schemes, to be varied at run-time. Each manager is represented by a logical class which declares the interface operations. Subclasses of these provide particular definitions of the interfaces. By seeking to maintain the same interfaces, the flexibility of the system is enhanced, since this facilitates the dynamic linking of new manager instances into a computation. Associated with each computation is a run-time linker/loader, whose logical state is the set of manager objects which the computation is permitted to access. In common with other projects (e.g. LeBlanc and Markatos, 1991; Mukherjee and Schwan, 1993; Campbell and Islam, 1993), one of the aims of the project is to investigate how applications may benefit from different resource management policies at different phases of execution.

3.2 Flexibility of policy level

In general it is to be expected that increased flexibility may have to be paid for by decreased performance. For example, it may be that providing policy at user-level will mean that the trap interface will be traversed too often for some purposes. For this reason, for example, a user-level implementation of virtual shared memory could well be too inefficient. It is envisaged that such performance-critical mechanisms may be provided by 'trusted' resource manager objects which, after development at user-level, can execute at supervisor-level. The facility to vary policy level will be a useful investigative tool. On dedicated systems, managers could run as 'trusted'. An analogous mechanism is available in Chorus, where a user-level server can run as a supervisor-level server to improve performance (Gien, 1991).

3.3 Flexibility of use

General purpose operating systems, by emphasising optimal multi-user system resource usage, seek to provide reasonable individual application performance. In contrast, the system being developed emphasises the performance of individual applications, but will also provide support for secure multi-user and general-purpose environments. That is, whilst the system is single application-oriented, it is intended to allow the hardware object to support a number of computations simultaneously. In this case the hardware object must be responsible for allocating processor time to each computation and its associated resource management environment. That is, the hardware object may be instructed to time-slice between computations in order to, if required, run a parallel machine multi-user. To support this situation, the hardware object must provide protection in terms of access to physical resources. In particular, the hardware object must check user-level requests by computations to access virtual address translation contexts and to access disk blocks. In general, protection between computations must be afforded by the hardware object. Protection within a computation must be the responsibility of the managers of the computation.

3.4 Development environment

The usefulness of such a system lies partly in its ability to support multiple resource management *policies*. Its usefulness is thus limited by the ease with which new policies can be implemented. The use of object-oriented techniques provides code-reuse. The implementation language for user-level managers is C++, which has been used for customisable operating system kernels (e.g. Campbell and Islam, 1993). A development environment will be associated with the system which will allow language system implementors to tailor existing resource manager classes to specific requirements. The hardware object is implemented in assembler and C.

3.5 Current status of the project

The system is in the early stages of implementation. The initial target hardware is the EDS machine (Ward and Townsend, 1990). This is a tightly-coupled thirteen-node distributed store multicomputer, where each node has two Sparc (MMU-cache-coherent) processors sharing nodal store (64 Mb per node). This machine thus presents a model for scalable hybrid multicomputer/multiprocessor architectures. All resource management code is being written to allow multithreading in a shared store multiprocessor environment. The major components of the hardware object, that is those dealing with execution contexts, has been implemented to run on the Sparc processors of one node of the EDS machine. Much of the work of this implementation has been concerned with optimising the treatment of the register window file of the Sparc. The current size of the hardware object is 47 Kb of text and 29 Kb of data (mainly trap tables, interrupt vector and interrupt stacks). The code includes trap handling (in particular about a quarter of the text is concerned with register window handling), thread context handling, hardware address translation handling and simple loading of user-level code from an incore disk. There is a small amount of state shared between the hardware object and the user-level Process Manager. This state is used to facilitate user-level event handling, and consists mainly of event thread context blocks, counters and flags. The design decision to access event handling state directly rather than via interface routines was made on the grounds of efficiency (as in e.g. Marsh et al., 1991). The trade-off is that this state must be encapsulated in a data type included by all Process Manager instances. Event handling at user-level is to be performed by threads, flagged as runnable by the hardware object, but scheduled by the user-level Process Manager. This interaction between the hardware object and the Process Manger forms the basis of the upcall mechanism into user-level. At present no Store Manager is implemented; thread stacks are obtained directly from the hardware object by means of downcalls direct from the Process Manager to validate a stack-sized range of virtual addresses.

The approach to design taken by customisable systems is such that an incremental approach to implementation is facilitated. A series of Process Manager instances of increasing complexity have been coded. The version currently being linked with the hardware object is intended to test and obtain measurements of the event handling mechanism only. Customisable backing store management is also being designed and coded. The backing store design is intended to allow user-level determination of caching policy and representation. The EDS machine does not have local disks and so relies on incore disks for accessing user binaries. At present binaries are loaded by a simple boot loader which is part of the hardware object. However, the loader is viewed as being the backbone of the system and will probably be based on the GNU linker, residing at user-level. The system currently uses the GNU assembler, C and C++ compiler and linker tools, and the GNU C library has been modified, where appropriate, to generate downcalls to the hardware object interface.

4 Summary

It is recognised that application-oriented resource management may improve the performance of individual parallel programs. One application-oriented approach is to provide user-level, run-time, libraries of resource management routines. These libraries can be linked to applications and run-time systems. Using object-oriented techniques, new resource managers, with modified policies, can be derived from existing managers. These techniques allow both customisation and code reuse. An overview is given of a system, currently being developed, which has a small hardware-dependent executive, and user-level library-based resource management.

5 Acknowledgements

The authors would like to thank John Keane, Brain Warboys, John Gurd and all the members of the Centre for Novel Computing.

6 References

Anderson, T.E., B.N. Bershad, E.D. Lazowska and H.M. Levy (1992) Scheduler activations: Effective kernel support for the user-level management of parallelism. ACM Transactions on Computer Systems 10(1), 53-79.

Bershad, B., E. Lazowska and H. Levy (1988) Presto: A system for object-oriented parallel programming. Software - Practice and Experience 18 (8), 713-732.

Black, D., D.B. Golub, D.P. Julin, R.F. Rashid, R.P. Draves, R.W. Dean , A. Forin, J. Barrera, H. Tokuda, G. Malan and D. Bohman (1992) Microkernel operating system architecture and Mach. Proc. Usenix Workshop on Microkernels and Other Kernel Architectures (April), 11-30.

Bodin, F. and T. Priol (1992) Overview of the KOAN Programming Environment for the iPSC/2 and performance evaluation of the BECAUSE test program 2.5.1. IRISIA Publication Interne 689 and Proc of BECAUSE European Workshop (October 1992).

Bryant, R., H. Chang and B. Rosenburg (1991) Experience developing the RP3 operating system. Usenix Association Proc. Symp. on Experiences with Distributed and Multiprocessor Systems (Summer), 1-18.

Bull, M. and G. Riley (1994) A method for developing efficient parallel code on virtual shared memory architectures. In Prep.

Campbell, R.H. and N. Islam (1993) CHOICES: A parallel object-oriented operating system. Research Directions in Concurrent Object-Oriented Programming, ed. G. Agha, P. Wegner and A. Yonezawa. MIT Press, 393-451.

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

Herrmann, B., M.I. Ortega, L. Philippe (1991) UNIX on a multicomputer: The benefits of the CHORUS architecture. Chorus Systems Technical Report CS/TR-91-46.

Holman, A. (1992) The Meiko Computing Surface: A parallel and scalable open systems platform for Oracle. LNCS 618, 96-113.

Keane, J.A. and M.Q. Xu (1992) Porting a parallel language onto a virtual shared memory parallel machine. Proc. 1992 DAGS/PC Symp. (June), 234-245.

Lazowska, E.D. (1992) System Support for high performance multiprocessing. Usenix Association Proc. Symp. on Experiences with Distributed and Multiprocessor Systems (March), 1-11.

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

Leunig, S.R. (1987) Abstract Data Types in the Flagship System Software. Flagship Document FLAG/DD/303, ICL.

Marsh, B.D., M.L. Scott, T.J. LeBlanc and E.P. Markatos (1991) First-class user-level threads. ACM operating system Review 25(5) (Proc 13th ACM Symp on operating system Principles (Oct, 1991)), 110-121.

Mayes, K.R. (1993) Trends in operating systems towards dynamic user-level policy provision. University of Manchester Computer Science Technical Report UMCS–93–9–1.

Mayes, K.R. and J.A. Keane (1993) Levels of atomic action in the Flagship parallel system. Concurrency: Practice and Experience 5(3), 193-212.

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

Mukherjee, B., K. Schwan and P. Gopinath (1993) A survey of multiprocessor operating system kernels (draft). Georgia Institute of Technology, College of Computing Technical Report GIT-CC-92/05.

Philbin, J. (1992) Customizable policy management in the Sting operating system. LNCS 748, 380-401.

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.

Rothnie, J. (1992) Kendall Square Research introduction to the KSR1. Proc. 1992 DAGS/PC Symp. (June), 200-210.

Schlichtiger, P. (1991) Closely coupled systems. LNCS 563, 44-47.

Tanenbaum, A.S., R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Janson and G. van Rossum (1990) Experiences with the AMOEBA distributed operating system. Comms ACM 33(12), 46-63.

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

Watson, P. and P. Townsend (1990) The EDS parallel relational database system. LNCS 503, 149-166.

Weiser, M., A. Demers, and C. Hauser (1989) The Portable Common Runtime approach to interoperability. Proc. 12th ACM Symp. on Operating System Principles, 114-122.