The Importance of Performance-Oriented Flexibility in System Software for Large-Scale Shared-Memory Multiprocessors

Orran Krieger, Ben Gamsa, Karen Reid, Paul Lu, Eric Parsons and Michael Stumm
Department of Electrical and Computer Engineering and
Department of Computer Science
University of Toronto

There has been a growing interest in large-scale shared-memory multiprocessors that support a wide range of application workloads. The architectural trend of such systems has been towards a distributed architecture, where processors, memory modules, and I/O modules are distributed throughout the machine, connected by some form of interconnect (rings, meshes, trees). Examples include the Stanford DASH [10] and FLASH [9], Massachusetts Institute of Technology Alewife [2], University of Toronto Hector [15], and Kendall Square Research KSR-1 [3] multiprocessors. In this type of architecture, commonly referred to as a Non-Uniform Memory Access time (NUMA) system, managing the locality between various resources is key to obtaining good performance.

Operating system services and their implementations need to be significantly more flexible on large-scale, shared-memory multiprocessors than on uniprocessors or small-scale multiprocessors. This flexibility is required because of (i) the diverse workload these machines must support, (ii) the sensitivity of large-scale parallel applications to operating system policy, and (iii) because of the need to study various policy and implementation issues due to the immaturity of system software for these systems. We discuss each of these requirements in turn.

First, large-scale multiprocessors must be able to concurrently support the efficient and fair execution of applications that vary from sequential interactive jobs to very large parallel applications, and from scientific applications to transactional data base systems. These applications have very different requirements with respect to (i) the requests they make, (ii) the resources they use, and (iii) their performance objective, such as response time or throughput. This is in sharp contrast to operating systems for workstations, which have been reasonably successful optimizing for a workload consisting of "typical" Unix applications.

Second, the system software for large-scale multiprocessors must be flexible in order to meet the specific requirements of different large-scale parallel programs. These applications often require application-specific policies, and even minor deviations in the expected behavior of a service can severely impact their performance. For example, an I/O intensive application may require that data be distributed across disks to match the way it intends to access the data and that a specific prefetching policy be used; minor deviations may cause unexpected disk seek and rotational delays which can slow an application down by an order of magnitude. Further examples from the domains of I/O, memory, and processor management are given further below.

Third, operating systems for large-scale multiprocessors are still in their infancy, and the operating system demands of parallel applications are still poorly understood. With the extra dimensions of locality and concurrency to consider in multiprocessor systems, most of the traditional operating system policies for managing resources need to be revisited. The only way to realistically compare different policies and implementations is to support them within a single operating system framework. Hence, experimental research in this area requires a flexible operating system that can easily be extended.

It is the exploration of how flexibility can be used to optimize performance, both with respect to individual applications and the system as a whole, that motivates our work. In particular, we are interested in,

1. studying how flexibility can be used to improve the performance of parallel applications,
2. studying how flexibility can be used to improve the performance of system software, and
3. developing techniques to minimize the performance overhead of flexibility.

To keep the problem manageable, we take a pragmatic approach to flexibility, using well-known techniques and readily available support software.

In this position paper, we discuss the need for flexibility and describe how we intend to achieve it in the scalable, shared-memory operating system we are currently designing and implementing, called *Tornado*. The design

of Tornado is mostly driven from our experiences over the last 3–4 years in designing and implementing a first generation such system, called Hurricane [14], for our own locally-developed multiprocessor, Hector [15]. On this system, among other services, we developed a flexible parallel file system [7] that successfully exploits the techniques for flexibility we expect to apply in Tornado.

We first describe the general techniques we use for flexibility, then describe where we apply flexibility in our system, and conclude with some of the problems we have found in designing a flexible operating system.

Techniques used for flexibility

We adopt techniques from several areas to incorporate flexibility into our operating system: object oriented systems design [1, 4], run time code generation [13], and the relocation of kernel and system functionality into user-level libraries [8, 6, 12].

Objects represent virtual and physical resources provided by our operating system, such as files, memory regions, and processes. Encapsulating these abstractions allows us to easily experiment with policies and implementations to suit particular application requirements. In order to achieve operating system scalability across a wide variety of workloads, it is important to allow individual objects to control the degree of replication, migration, and consistency of their internal state, as discussed in the next section.

In many cases, flexibility is needed to select and specialize policies, data structures, and algorithms to meet the requirements of a particular application, but these requirements are stable for the duration of the (long running) program. In these cases, we use run time code generation (as well as specialized object selection) in order to provide an optimized implementation of the policies, data structures, and algorithms chosen.

Much of the operating system functionality is provided by user level libraries. They allow object policies and implementations to be more easily customized for each application, and reduce the need to cross address spaces to handle common requests.

Flexibility requirements

Flexibility is necessary in our environment to obtain good performance for both applications and system software. Characteristics of applications such as memory and file access patterns, processor utilization, and communication patterns can be used to develop application specific policies and implementations that manage system resources more efficiently than common global policies could. For example:

memory management: The page size and the memory management policy used for ejecting, pre-fetching, migrating, and replicating pages should be specific to the particular application memory region in order to match the application's access patterns for the data located in the region. The method (e.g., shared memory, message passing, up-calls on faults) and interface used to support the interaction between the application and memory manager should also conform to the application's specific per-region requirements.

disk I/O: The disk block size, along with the policies used for distributing data across the system disks, storing the data on individual disks, and replicating data should be specific to the file to match the varying file access patterns. The provisions for fault tolerance should vary depending on the level of reliability required for the file. Additionally, it must be possible for an application to override the default policies associated with a given file to meet its specific I/O requirements.

scheduling: Giving an application control over the scheduling of its threads allows the application to use its processors most effectively. The higher-level policies used to schedule multiple applications must interact with the thread schedulers so that the application specific schedulers each have sufficient resources to schedule themselves efficiently, and so that the overall allocation of resources is fair.

The operating system is itself an example of a parallel application with large CPU and I/O requirements. System performance can be substantially improved by selecting the particular implementation of a system object to match a given use, such as trading off flexibility for data structure size, or total code size for critical path length. We have also found that, in order to support a system that can scale from tens of processors to hundreds, each object must be allowed (though not required) to choose its own style of scalable data structures, method invocations, and locking protocols based on the type of service it provides and how widely it provides it.

Flexibility is important to the internal structures of a scalable operating system, in part because of the wide variation in workloads that must be efficiently supported. As mentioned earlier, workloads vary from large numbers of sequential programs to a small number of very large parallel programs, and these make very different demands on the internals of the operating system and how its data structures should be managed. Consider, for example, some resource that may be used at one time by a single sequential application and later by all the processes of a large parallel application (perhaps a shared

region of memory). To support the sequential application efficiently, a simple data structure kept local to the application might be used, whereas to support the parallel application, a replicated or distributed data structure might be more appropriate since concern for concurrency would dominate the concern for sequential speed or space.

A second important consideration is that the operating system should scale down as well as up. When running on a 64 processor machine, an application that uses only 16 processors should see the same type of performance as it would running on a dedicated 16 processor system with an operating system designed for that size (this, of course, requires that the hardware be scalable in both directions as well). By adapting the operating system objects to reflect the current workload and the tradeoffs that apply under that workload, such performance scaling (both up and down) should be possible.

Open issues

A number of issues remain unresolved in our system. First, although flexibility can be used to tune performance in different ways (such as reducing critical path code length, reducing cache footprint, or improving locality), in some cases tuning performance for one dimension may have an unexpected impact on others, either when considering individual objects, or the system as a whole. For example, although run time code generation may improve performance for individual cases, its overuse may result in an overall degradation due to reduced instruction cache hit rates.[1]

A second issue arises with our decision to manage most resources locally, e.g., per-application, per-processor, or per-file. While this allows policies to be object specific, it may complicate system-wide resource management and scheduling, since the global manager must contend with object specific policies that often have conflicting goals.

A third difficulty arises when the abstractions we define in the operating system meet the predefined interfaces exported by the hardware (e.g., protected kernel state, TLB format) and low-level system software (e.g., interrupt routines, exception processing). In these cases, we have found it necessary to constrain the abstractions provided by our system objects to fit the contexts in which the objects are being used. It is not clear how significant these constraints will turn out to be.

Finally, in a number of places we have found it necessary to hard-code a particular implementation to obtain the highest possible performance (for example, interprocess communication). Where these inflexible components meet the flexible ones, we associate public data structures with the flexible objects. For common operations, the hard-coded implementations use the state in the public data structures to complete their operations efficiently. To indicate that the flexible object wants to take special action, the object can mark the public data invalid, causing the hard-coded implementation to request the object (using the normal mechanisms) to complete the operation itself. We are still attempting to determine how much this technique will complicate the system as a whole.

[1] Such a case can be especially difficult to detect, since no degradation of performance occurs until the cache footprint of the system exceeds the processor cache size (which may not occur during simple tests).

References

[1] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, September 1993.

[2] David Chaiken, John Kubiatowics, and Anant Agarwal. "LimitLESS Directories: A Scalable Cache Coherence Scheme". In *ASPLOS-IV Proceedings*, pages 224–234, Santa Clara, California, April 8-11 1991. ACM.

[3] S. Frank, J. Rothnie, and H. Burkhardt. The KSR1: Bridging the gap between shared memory and MPPs. In *IEEE Compcon 1993 Digest of Papers*, pages 285–294, 1993.

[4] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the 1993 Summer Usenix Conference*, 1993.

[5] Graham Hamilton, Michael L. Powell, and James J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 69–79, 1993.

[6] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file systems (ELFS): An object-oriented approach. In *Proceedings of the Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'94)*, 1994. To appear.

[7] Orran Krieger. *HFS: A flexible file system for shared memory multiprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, 1994.

[8] Orran Krieger, Michael Stumm, and Ronald Unrau. The Alloc Stream Facility: A redesign of application-level stream I/O. *IEEE Computer*, 27(3):75–82, March 1994.

[9] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *In Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994. ACM.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Henessy, M. Horowitz, and M. S. Lam.

The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63, March 1992.

[11] Anders Lindstrom, Rex di Bona, J. Matthew Farrow, Frans Henskens, John Rosenberg, Alan Dearle, and Francis Vaughan. A model for user-level memory management in a distributed, persistent environment. In *Proceedings of the Seventeenth Annual Computer Science Conference, ACSC-17*, pages 343–354, 1994.

[12] Chris Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–55, 1993.

[13] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 191–201, Arizona, December 1989.

[14] R. Unrau, M. Stumm, O. Krieger, and B. Gamsa. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*. To appear. Also available as technical report CSRI-268 from ftp.csri.toronto.edu.

[15] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. "The Hector Multiprocessor". *IEEE Computer*, 24(1), January 1991.