Optimizing Multi-Processor Operating Systems Software Research Review

Jonathan Appavoo

In this paper we review general purpose multiprocessor operating systems research that is relevant to maximizing performance, focusing on the exploitation of locality in large scale systems. We cover three areas in particular:

1. A historical review of multiprocessor operating systems research.
2. A discussion of disciplined approaches to applying distributed data structures to systems construction.
3. A review of modern operating systems research that addresses the unique characteristics of scalable hardware platforms.

Utilizing concurrency to improve performance is not new and certainly not restricted to the domain of operating systems. Many of the features introduced into microprocessor architectures attempt to extract parallelism at the instruction level in order to hide latencies and improve the number of instructions executed per cycle. Increasingly scientific users split their calculations into independent components which can be executed in parallel in order to decrease absolute runtime. Operating systems are unique, from a software perspective, in their requirement to support and enable parallelism rather than exploiting it to improve their own performance. An operating system (OS) must ensure good overall system utilization and high degrees of parallelism for those applications which demand it. To do this, operating systems must: 1) utilize the characteristics of the hardware to exploit concurrency in general purpose workloads and 2) facilitate concurrent applications, which include providing models and facilities for applications to exploit the concurrency available in the hardware.

It is critical that an OS reflect the parallelism of the workloads and individual applications to ensure that the OS facilities do not hinder overall system or individual application performance. This is often overlooked. Smith[117] alludes to the individual application requirements, noting that the tension between protection and performance is particularly salient and difficult in a parallel system and that the parallelism in one protection domain must be reflected in another. In other words, to ensure that a parallel application within one protection domain can realize its potential performance, all services in the system domains that the concurrent application depends on, must be provided in an equally parallel fashion. It is worth noting that this is true not only for individual applications but also for all applications forming the current workload on a parallel system: the demands of all concurrently executing applications must be satisfied with equal parallelism in order to ensure good overall system performance.

Much of the research into multiprocessor operating systems has been concerned with how to support new, different or changing requirements in OS services, specifically focusing on user level models of parallelism, resource management and hardware configuration. We will generically refer to this as support for flexibility. In contrast, the research done at the University of Toronto has pursued a performance oriented approach. The group has suggested two primary goals for the success of multiprocessor operating systems:

1. Provide a structure which allows for good performance to be achieved with standard tools, programming models and workloads without impacting the user or programmer. Therefore the OS must support standards while efficiently mapping any available concurrency and independence to the hardware without impacting the user level view of the system.

2. Enable high performance applications to side step standards to utilize advanced models and facilities in order to reap maximum benefits without being encumbered by traditional interfaces or policies which do not scale.

Surveying the group's work over the last several years two general requirements for high performance systems software have been identified:

1 Reflect the Hardware:

- Match scalability of the hardware in the systems software. Ensure software representations, management and access methods do not limit system scale, for example, software service centers required to manage additional hardware resources should increase with the scale of the system.
- Reflect unique performance characteristics of MP hardware to maximize performance. Mirror locality attributes of the hardware in the software structures and algorithms: avoid contention on global busses and memory modules, avoid shared cache-line access and efficiently utilize replicated hardware resources.

2 Reflect the Workload:

- Map independence between applications in systems structure.
- Match the concurrency within applications in the systems structures.

This paper examines the relevant research work which lead to these requirements and the approaches advocated for their satisfaction by the group at the University of Toronto. We begin with a review of the history of MP OS research, then look at systems work directly related to the use of distributed data structures and finally conclude with a look at modern research into MP OSes.

1 Multiprocessor Operating Systems Review

One can find many definitions of operating systems in the literature[105, 115, 123]. On closer inspection, rather than a precise definition, one typically is given a list of roles, responsibilities and services that operating systems provide. There is really no one definition of what an operating system is or even what it does. Over time we have seen the roles and services provided by operating systems grow as the expected software environment accompanying a general purpose computer has increased both in size and complexity.

Despite the lack of a clear definition there appear to be two general requirements for an operating system: 1) Provide useful abstractions and mechanisms that facilitate and simplify the use of a computer system and 2) Provide functionality in a high performance manner, having minimal impact on applications utilizing the system. Unfortunately, these requirements are often at odds. A set of rich abstractions often leads to complex implementations which result in increased overheads and greater performance penalties. To this end operating system designs have tended to advocate the use of a small set of simple abstractions and mechanisms which can be implemented efficiently and re-applied across the system. For example, Plan 9[99] uses the file system and file abstraction to represent all resources including CPU's. Unfortunately, such religious application of a single abstraction often leads to difficulties when the abstraction does not map naturally.

Another approach that systems designers have taken in order to try and find a balance between the requirement for rich abstractions and high performance is to separate the common from uncommon paths in the system. Effort is then taken to ensure that the common paths are kept short and efficient while the uncommon ones are permitted greater leeway in order to support a richer abstraction.

A complicating factor in the balancing act of designing and building an operating system is the nature of the hardware system and how users expect to leverage it. The more complex the hardware, the greater the challenge it is to develop useful abstractions which simplify the use of the system while still allowing all aspects of the system to be exploited. Additionally, achieving high performance on complex hardware tends to require more subtle implementations which account for the nuances of the system. As such, the hardware of a computer system has significant impact on operating systems. Hardware features may require new abstractions and demand more complex and intricate systems software support.

Arguably the most complex computer systems are those with multiple processing units. The advent of Multiprocessor computer systems presented operating systems designers with four intertwined issues:

1. True parallelism,
2. new and more complex hardware features, such as multiple caches, multi-staged interconnects and complex memory and interrupt controllers,
3. subtle and sensitive performance characteristics,
4. and the demand to facilitate user exploitation of the system's parallelism while providing standard environments and tools.

Based on the success of early multiprogramming and time sharing systems and what was viewed as fundamental limits of uniprocessor performance, early systems researchers proposed multiprocessors as the obvious approach to meeting the increasing demands for general purpose computers. The designers of multics, in 1965, went so far as to say:

> ...it is clear that systems with multiple processors and multiple memory units are needed to provide greater capacity. This is not to say that fast processor units are undesirable, but that extreme system complexity to enhance this single parameter among many appears neither wise nor economic.

Perhaps the modern obsession with uniprocessor performance for commodity systems is the strongest evidence of our inability to have successfully leveraged large scale multiprocessors for general purpose computing. Large multiprocessors are predominately now considered as platforms for specialized super-computing applications[1].

Real world experimentation with general purpose multiprocessors began as simple dual processor extensions of general purpose commercial uniprocessor hardware[7]. An attendant side effect was to extend the uniprocessor operating system to function correctly on the evolved hardware. The

[1]This is slowly changing, albeit at small scale. Over 10% of Sun's workstations being sold are now multiprocessor workstations. Increasingly, Intel-based servers with 2-4 processors are being used. The new PlayStation-3 expected for release in 2005 will have 4 PowerPC processors. Nevertheless, larger MP's with 8-30 processors are still rare and considered special purpose.

primary focus was to achieve correctness in the presence of true parallelism. This precipitated the major trend in industrial operating systems. They start with a standard uniprocessor system, whose programming models and environments are accepted and understood, and extend it to operate on multiprocessor hardware. Various techniques for coping with the challenges of true parallelism have been explored, starting with simple techniques which ensured correctness but yielded little or no parallelism in the operating system itself. As hardware and workloads evolved, the demand to achieve greater parallelism in the operating systems forced OS implementors to pursue techniques which would ensure correctness but also achieve higher performance in the face of concurrent workload demands.

The fundamental approach taken was to apply synchronization primitives to the uniprocessor code base in order to ensure correctness. Predominately the primitive adopted was a shared memory lock, implemented on top of the atomic primitives offered by the hardware platform. The demand for higher performance lead to successively finer grain application of locks to the data structures of the operating systems. Doing so increased concurrency in the operating system at the expense of considerable complexity and loss of platform generality. The degree of locking resulted in systems whose performance was best matched to systems of a particular scale and workload demands. Despite having potentially subtle and sensitive performance profiles, the industrial systems preserved the defacto standard computing environments and achieved reasonable performance for small scale systems, which have now become widely available. It is unclear if the lack of acceptance of large scale systems is due to the lack of demand or the inability to extend the standard computing environments to achieve good performance on such systems for general purpose workloads.

In general, the industrial experience can be summarized as a study into how to evolve standard uniprocessor operating systems with the introduction of synchronization primitives. Firstly, this ensures correctness and secondly, permits higher degrees of concurrency in the basic OS primitives. This is in contrast to the majority of the research work, which has focused on flexibility and improved synchronization. In the remainder of this section we will look at early systems research which focused on flexibility as well as later work into: Lock-Free/Non-Blocking Kernels, Distributed Operating Systems and Hybrid Distributed Systems which influenced modern research.

1.1 Flexibility

The Hydra[34, 78, 133, 134] research system done at Carnegie Mellon University (CMU) in the 1970's set the trend for many of the multiprocessor research operating systems that would follow. Given that there was little experience with the newly emerging general purpose multiprocessors and the field of parallel computing, the Hydra team concluded that an MP OS capable of exploiting and exploring the potential inherent in multiprocessor systems would have to be uniquely focused on flexibility. Their key focus was on a radical system structure which permitted degrees of flexibility not available in contemporary systems. The approach taken was to apply, in concert, two emerging trends of the time: 1) a kernel based system structure[59] which enables flexibility by encouraging a separation of mechanism from policy and 2) object oriented design[93] which permits flexibility through modularization. They proposed one of the first designs for a multiprocessor operating system utilizing a per-resource based decomposition and vertical structure in the context of what today would be called a micro-kernel.

Despite being targeted at exploring the potential for multiprocessors, Hydra's aggressive goal to utilize a radical new design for flexibility resulted in a novel philosophical study of micro-kernels and

object oriented design. Some of the observations they made and features they proposed included[2]:

- conceptualizing resources, both physical and abstract, as central system abstractions representing individual resource instances as independent object instances,
- vertical structuring on a per-user basis, where each user views the system as a set of objects,
- protection on a per-object basis via capabilities,
- resources are the fundamental unit of sharing for the system,
- a small kernel which only implements the essential mechanisms enabling the construction of the capability based object oriented structure,
- the majority of system functionality should be implemented by objects outside of the kernel,
- the use of specialization to permit multiple implementations of a service to coexist,
- leverage specialization to permit implementations to be optimized for particular scenarios,
- other object oriented features: type system, abstraction, polymorphism, opaque types and garbage collection.

Although very unique and forward-looking, the Hydra work does not appear to have explored or even justified the decomposition or flexibility with respect to multiprocessor system issues. It is unclear if the project proceeded past the implementation of the micro-kernel. Nevertheless, the focus on flexibility and basic system structure of Hydra carried on to the majority of research systems which followed.

Two projects closely related to Hydra were StarOS[34] and Medusa[92], also developed at CMU. Both systems were targeted for the CM* multiprocessor architecture which was one of the first systems designed for scalability utilizing clustered distributed memory with a NUMA model. StarOS was primarily concerned with exploring a specific model for structuring parallel computations in order to effectively exploit a large number of processors. Although abstract in nature with few conclusive results or clear suggestions, the StarOS authors appear to have been among the first to raise locality as an issue in exploiting the nature of NUMA architectures. They were also among the first to identify the need for scalable software which matches the scalability of the hardware. The actual kernel structure advocated in StarOS was heavily influenced by Hydra although the authors more explicitly adopted object oriented design principles from FAMOS[56][3].

Medusa's main focus was to address scalability and robustness by partitioning the system into disjoint utilities which execute on specific processors. The system was built around the notion of function shipping. The system services are decomposed into modules which are distributed to specific locations in the system and accessed via a cross processor asynchronous function call facility. By distributing the modules of the system, the authors intended to minimize contention and increase robustness. It is unclear that either was achieved but it is interesting to note that

[2]The Toronto group ultimately came to develop similar structures in the Tornado and K42 operating systems, based on the experiences gained from the Hurricane operating system

[3]One of the first attempts to leverage a modular structure in order to produce a system which can be specialized to meet the needs of a particular workload in the context of a general purpose operating system.

Medusa utilized a per-processor indirection table to locate the system modules. A similar approach was used many years after by Tornado and K42 to facilitate per-processor distribution of data-structures and services.

In 1985 the Tunis[44] operating system, done by another group at the University of Toronto under the direction of Ric Holt, was one of the first systems to focus on the importance of locality rather than flexibility. One of the aims of the project was to explore the potential for cheap multiprocessor systems, constructed from commodity single board microprocessors interconnected via a standard backplane bus. Each microprocessor board contained local memory and an additional bus connected memory board provided shared global memory. Given the limited global resources, the designers focused on structuring the system more like independent local operating system instances. This would prove to be a precursor of much later work like Hurricane which attempted to apply distributed system principles to the problem of constructing a shared memory multiprocessor operating system. Although limited in nature, the system is one of the first to provide uniprocessor UNIX compatibility while employing a novel internal structure.

It is worth noting that the early 1980's not only saw the emergence of tightly coupled shared memory multiprocessor systems such as the CM* but also loosely coupled distributed systems composed of commodity workstations interconnected via local area networking. Projects such as Thoth[29] and V[30] attempted to provide a unified environment for constructing software and managing the resources of a loosely coupled distributed system. Unlike the operating systems for the emerging shared memory multiprocessors, operating systems for distributed systems could not rely on hardware support for sharing. As such, they typically were constructed as a set of autonomous light-weight independent uniprocessor OS's which cooperated via network messages to provide a loosely coupled unified environment. Although dealing with very different performance tradeoffs, the distributed systems work influenced and intertwined with SMP operating systems research over the years. For example one of the key contributions of Thoth[29] and V[30] were micro-kernel support of light-weight user-level threads that were first-class and kernel visible.

In the mid 1980's the Mach operating system was developed at CMU based on the distributed systems of Rig and Accent[15, 68, 101, 102, 137]. One of the key features that was a major factor in Mach's success was the early commitment to UNIX compatibility while supporting user-level parallelism. In spirit, the basic structure of Rig, Accent and Mach is similar to Hydra and StarOS. All systems are built around a fundamental IPC model and in the case of Mach the basic IPC primitives are ports and messages. Processes provide services via ports to which messages are sent using capabilities and access rights. Mach advocates an object oriented-like model of services which are provided/located in servers. Rashid[102] states that Mach was, "designed to better accommodate the kind of general purpose shared-memory multiprocessors which appear to be on their way to becoming the successors of traditional general purpose uniprocessor workstations and timesharing systems". Mach's main contribution with respect to multiprocessor issues was its user-level model for fine grain parallelism via threads and shared memory within the context of a UNIX process. This model became the standard model for user level parallelism in most UNIX systems. Otherwise, Mach takes the traditional approach of fine grain locking of centralized data structures to improve concurrency on multiprocessors[4]. The later Mach work does provide a good discussion of the difficulties associated with fine grain locking, discussing issues of existence, mutual exclusion, lock hierarchies and locking protocols[15].

[4]Contemporary industrial systems such as Sequent's Dynix[13, 49, 67] were employing fine grain locking and exploring its challenges.

Many of the operating systems developed in the 1980's state the importance of multiprocessors and generally appeal to issues of flexibility which they argue is addressed by one form of modularity or another. Some systems such as Peace[106] were more abstract in nature, discussing general architectures for system modularity, while others tended to focus on the applications of a specific model or set of techniques for leveraging some form of object orientation. The following is a brief description of some of the systems:

Choices[21, 20, 70] focused on leveraging modern object oriented software engineering principles and techniques when designing and implementing an operating system. Although the work claims multiprocessor support, no insights or novel multiprocessor observations are made. The later work[70] does however describe a monitoring infrastructure which leverages distributed per-processor buffers in order to improve performance, however the infrastructure seems to have been built outside of their object oriented framework.

Elmwood[75] By the late 1980's the BBN Butterfly, a large scale commercial multiprocessor, had been accessible to researchers at the University of Rochester for experimentation. LeBlanc et al. [76] identified locality, Amdahl's law, and flexibility as key influences on software for large scale machines. The Rochester group would go on to explore the aspects of flexibility in great detail but to a large extent leave locality unexplored. Elmwood was a small prototype operating system for the BBN Butterfly. The key focus was on protection and user defined abstractions. Elmwood also adopted an object based design but with a focus on the flexible support for multiple parallel computations models. The design would heavily influence the later work done on Psyche.

Presto[14] perhaps best captures the intent of many of the parallel systems of the time which adopted an object oriented design. Object orientation is viewed as a tool for coping with the complexity of supporting new and unknown parallel computation models. Specifically, the authors argue that a single fixed model of parallel computation is insufficient and that a general parallel system must allow the application programmers to construct the parallel model which is appropriate. However, since computation models require system support and can be very complex to implement, a compositional approach is advocated. In the case of Presto an object oriented user level library is offered as the solution. Here the basic complexity is captured by the components of the library and flexibility is achieved by allowing the application programmer to construct her desired computational model via composition and specialization.

Psyche[109, 110, 111, 112] unlike Presto, aimed to provide a complete operating system environment for supporting multiple parallel programming models while providing traditional OS enforced protection and isolation. Similar to Mach, Psyche advocated a micro-kernel design but pursued user-level functionality of traditional OS models and services more aggressively. The key focus was to provide a minimal kernel which supported the construction of the system by isolated object-like modules which could be protected via a combination of capabilities and virtual memory protection. The goal again was to allow user defined execution environments by enabling the majority of system policies to be implemented at user level. The authors noted that locality is critical for good performance but decided it was premature to enable automatic support for locality management. They advocated for explicit programmer management via careful software construction but did not provide support or guidance for locality

optimizations beyond cross-processor function shipping.

Clouds[36, 37], unlike many of the distributed systems of the time that utilized a message passing model and as a side effect an object oriented like service model, adopted an object oriented approach as a first class way of building a distributed system. Each system service was provided by a course grain object contained in its own protection domain. Threads executed methods of an object and synchronization primitives where used to control concurrency internal to an object. Unlike many systems, an object in Clouds was globally accessible on all nodes of the distributed system. Software support for distributed shared memory was built into the base system which allowed an object's memory to be accessible in a coherent fashion on all nodes of the system. Clouds used an object oriented decomposition as a uniform model for distributed systems construction which enforced encapsulation and protection. Objects in Clouds were passive and did not presume an execution model; rather any number of threads could access an object in parallel, with concurrency control internal to the object. Many later systems, including Tornado, K42, Spring [57, 58], Mach-US [121] and others [38, 41, 136], adopted a similar object oriented structure albeit with variations in object grain, communication paradigms and protection.

In the late 1980's and early 1990's, projects arose which focused on aspects of building parallel systems beyond flexibility and modularity. The remainder of this section categorizes and discusses these efforts.

1.2 Lock-Free/Non-Blocking Kernels

Gottlieb et al. [52] present operating system-specific synchronization techniques based on replace-add hardware support. The techniques attempt to increase concurrency by avoiding the use of traditional lock or semphore-based critical sections. The techniques they propose are generalized in later work by Herlihy[61, 62]. Edler et al. [42] in the Symunix II system, claim to have used the techniques of Gottlieb et al. as part of their design for supporting large scale shared memory multiprocessors. Unfortunately, it is not clear to what extent the implementation of Symunix II was completed or to what extent the non-blocking techniques were applied. The main focus of Edler's work [42] was on Symunix II's support for parallelism in a UNIX framework, and specifically issues of parallel virtual memory management.

The later work of Massalin[83, 84, 85] explicitly studies the elimination of all locks in system software via the use of lock-free techniques. Massalin motivates the application of lock-free techniques for operating systems, pointing out some of the problems associated with locking:

Overheads: spin locks waste cycles and blocking locks have costs associated with managing the queue of waiters,

Contention: potential for poor performance due to lock contention on global data structures,

Deadlocks: care must be taken to avoid dead locks which adds considerable complexity to the system,

Priority Inversion: scheduling anomalies associated with locking, especially for real-time systems, introduce additional complexity.

One of the key methods used for applying lock free techniques was the construction of system objects which encapsulated a standard data structure and associated synchronization implemented with lock free techniques. Massalin argues that such an encapsulation enables the construction of a system in which the benefits of lock free techniques can be leveraged while minimizing the impact of its complexity. Despite showing that the lock-free data structures have better performance compared to versions implemented with locks, scalability is not established. Massalin's work was done in the context of the Synthesis operating system on a dual processor hardware platform, so the degree of parallelism studied was very limited. Furthermore, although the Synthesis work advocates for first reducing the need for synchronization, there is little guidance given or emphasis placed on this aspect[5]. The lock-free structures developed do not in themselves lead to less sharing or improved locality and hence although having better performance than lock based versions, the large scale benefits are likely limited.

Although the scalability of lock-free techniques is not obvious, the work does add evidence to the feasibility of constructing an operating system around objects which encapsulate standard data structures along with synchronization semantics. Such an approach enables the separation of concerns with respect to concurrency, system structure and reuse of complex parallel optimizations in the Synthesis operating system.

1.3 Distributed Operating Systems

Distributed Operating systems attempt to provide a single system image out of a distributed collection of network connect computers. The general structure and techniques are typified by such systems as: V[30] Accent[102], Sprite[60] and Amoeba[124]. These systems attempt to manage and coordinate the distributed resource in order to present the user with a unified environment in which the distribution is hidden. The nature of the computing environment (eg. local area networks versus wide area networks, homogeneous versus heterogeneous) and the techniques used (eg. explicit message passing versus remote procedure calls, custom versus standard/UNIX compatible environments) vary from system to system. These systems often deal with issues of robustness, flexibility and latency of network operations, however even early systems address issues common to multiprocessors with respect to concurrency and scalability[107].

In general, a client-server model is employed for service construction, where a common microkernel is run on each machine on top of which servers and clients execute. Multiple clients access a centralized server which is responsible for ensuring correctness by arbitrating the multiple requests while providing adequate concurrency for some number of clients. To ensure scalability, techniques guaranteeing that a given service does not become a bottleneck must be applied. The two primary approaches have been to utilize a multi-threading architecture for servers and the distribution of a service via replication and partitioning. Multi-threading allows multiple requests to be in-flight in a server thus increasing concurrency by overlapping computation with IO. Various different techniques have been used to distribute a service to increase concurrency and availability. These include partitioning of a service across multiple servers and client side caching as in the case of file

[5]Primarily, two approaches are advocated; 1) Code Isolation and 2) Procedure Chaining. Massalin argues for the specialization of code paths which operate on independent data in a single threaded fashion thus avoiding the need for synchronization. This approach however, is explored in a limited fashion in Synthesis and relies on runtime code generation. Tornado's object oriented structure explores the parallel advantages independent data in a more generalized and structured manner. Procedure Chaining, the enqueueing of parallel work, does not improve concurrency or reduce sharing but simply enforces serialization via a scheduling discipline.

systems[64, 91], or the use of distributed data structures in the case of Globe[127] and SOS[114] which enable distributed implementation of services (this approach is discussed in more detail in section 2.1. To a large extent the techniques of distributed systems underlie web technologies albeit in a less structured manner.

1.3.1 MOSIX[10, 11]

MOSIX focuses on the scalability issues associated with scaling a single UNIX system image to a large number of distributed nodes. Barak et al. strive to ensure that the design of the internal management and control algorithms impose a fixed amount of overhead on each processor, regardless of the number of nodes in the system. Probabilistic algorithms are employed to ensure that all kernel interactions involve a limited number of processors and that the network activity is bounded at each node.

Unlike many of the other distributed systems, MOSIX is designed around a symmetric architecture where each node is capable of operating as an independent system, making its own control decisions independently. Randomized algorithms are used to disseminate information such as load without requiring global communication that would inhibit scaling. This allows each node to base its decisions on partial knowledge about the state of the other nodes without global consensus.

Although MOSIX is targeted at a distributed environment with limited sharing and coarse grain resource management, its focus on limiting communication and use of partial information to ensure scalability is worth noting. Any system which is going to scale in the large must avoid algorithms which require global communications leveraging partial or approximate information where possible.

1.3.2 Sprite[60]

Like Mach, the Sprite kernel[60], although designed for distributed systems, was designed to execute on shared memory multiprocessor nodes of a network. It employed both course and fine grain locking. The researchers at Berkeley conducted a number of macro and micro benchmarks to evaluate the scalability of Sprite on a 5 processor system. They made the following observations:

- The system was able to demonstrate acceptable scalability for the macro benchmarks up to the five processors tested. Considerable contention was experienced in the micro benchmarks however, indicating that macro benchmark results will not extend past 7 processors.
- Even a small number of coarse grain locks can severely impact performance. A running Sprite kernel contains 500 to 1000 locks but consistently two coarse grain locks suffered the most contention and were primary limiting factors to scalability. At 5 processors a coarse grain lock was suffering 70% contention.
- Coarse grain locks are a natural result of incremental development and locking. Developers, when first implementing a service, will acquire the coarsest grain lock in order to avoid synchronization bugs and simplify debugging, only subsequently do they split the locks to obtain better scalability.
- Lock performance is difficult to predict even for knowledgeable kernel developers who designed the system. They found that locks which they expected to be problematic were not

and unexpected locks were. Further, performance was brittle with performance cliffs occurring at unpredictable thresholds, for example, good performance on 5 processors and poor performance on 7.

- The authors advocate for better tools to help developers understand and modify locking structures.

1.4 Hybrid Distributed Systems

In the 1990's a number of research groups, motivated by distributed systems research, proposed the use of multiple autonomous instances of micro-kernels in the construction of a multiprocessor operating system environment. Such hybrid systems attempt to leverage the distributed systems techniques to bound and cope with the complexity of constructing a single scalable kernel, while trying to exploit the tight coupling available on a multiprocessor in order to enable scalable application performance. Many variations were explored including applying distributed systems architectures to message passing architectures[6, 90] and attempts to build distributed shared memory abstractions on top of a set of kernels in order to provide a unified environment for both shared memory and message passing systems[132]. The Hurricane[126] and Hive[24] projects both focused on the construction of operating systems for large scale shared memory multiprocessors, and the results of both would directly influence the later research and are discussed separately in Section 3.2.1. The remainder of this subsection will discuss Paradigm, a pre-cursor to the Hurricane and Hive work.

In 1991 Cheriton et al. proposed an aggressive distributed shared memory parallel hardware architecture called Paradigm and also described OS support for it based on multiple co-operating instances of the V micro-kernel, a simple hand-tuned kernel designed for distributed systems construction, with the majority of OS function implemented as user-level system servers [28]. The primary approach to supporting sharing and application coordination on top of the multiple micro-kernel instances was through the use of a distributed file system. The authors state abstractly that kernel data structures such as dispatch queues are to be partitioned across the system to minimize interprocessor interference and to exploit efficient sharing, using the system's caches. Furthermore, they state that cache behavior is to be taken into account by using cache-friendly locking and data structures designed to minimize misses with cache alignment taken into consideration. Finally they also assert cheap UNIX emulation. It is unclear to what extent the system was completed. The authors identified key attributes and observed the importance of accounting for cache performance in the design of a modern multiprocessor operating system.

In 1994, as part of the Paradigm project, an alternative OS structure dubbed the Cache Kernel[27] was explored by Cheriton et al. At the heart of the Cache Kernel model was the desire to provide a finer grain layering of the system, where user-level application kernels are built on top of a thin cache kernel which only supports basic memory mapping and trap reflection facilities via an object model. From a multiprocessor point of view however, its architecture remained the same as the previous work. Each cluster of processors of the Paradigm system ran a separate instance of a Cache Kernel. The authors explicitly point out that such an architecture simplifies kernel design by limiting the degree of parallelism that the kernel needs to support. Assuming a standard design, the approach results in reduced lock contention and eliminates the need for pursuing aggressive locking strategies. The authors also allude to the fact that such an approach has the potential for improving robustness via limiting the impact of a fault to a single kernel instance and only the applications which depend on it. Although insightful, the Cache Kernel work did not explore or

validate its claims. Hurricane[126] and Hive[24], contemporaries of the cache kernel did explore these issues in greater detail adopting the high-level architecture proposed by the Paradigm group.

1.5 Summary

To summarize, it is worthwhile reviewing the experiences of the RP3 researchers[17] attempting to use the Mach micro kernel to enable multiprocessor research. After all, the real test for operating systems research is its ability to be used by others. The RP3 authors state that a familiar programming environment was a factor in choosing Mach as it was BSD Unix compatible while it still promised flexibility and multiprocessor support. The RP3 hardware was designed to try and minimize contention in the system by providing hardware support for distributing addresses of a page across physical memory modules, under control of the OS. There was no hardware cache coherency, but RP3 did have support for specifying uncached access on a page basis and user mode control of caches, including the ability to mark data for later manual eviction. The OS strategy was to start with the Mach micro-kernel, which supported a standard UNIX personality on top of it, and progressively extend it to provide: gang scheduling, processor allocation facilities and the ability to exploit the machine specific memory management features.

The authors found that they needed to restructure the OS to utilize the hardware more efficiently in order to improve performance, specifically needing to reduced memory contention. Some interesting points made by the authors include:

- Initially throughput of page faults to independent pages degraded when more that three processors touched new pages because of contention due to the spin lock algorithm. Spin locks create contention in the memory system especially in the memory modules in which the lock is located and performance worsens with the number of spinners. "Essentially, lock contention results in memory contention that in turn exacerbates the lock contention." By utilizing memory interleaving it was possible to distribute data and hence reducing the likelihood of co-locating lock and data and hence improve the performance.

- Contention induced by slave processors spinning on a single shared word degraded boot performance to 2.5 hours. Eliminating this reduced boot time to 1/2 hour.

- The initial use of a global free list did not scale, so the authors had to introduce distributed per processor free lists to yield efficient allocation performance.

- The authors found bottlenecks in both UNIX and Mach code with congestion in memory modules being the major source of slowdown. To reduce contention, the authors used hardware specific memory interleaving, low contention locks and localized free lists. We contend that the same benefits could have been achieved if locality had been explicitly exploited in the basic design.

Critical to the RP3 team were UNIX compatibility and performance. In the end the flexibility provided by Mach did not seem to be salient to the RP3 researchers. Mach's internal traditional shared structure limited performance and its flexibility did not help to address these problems. To a large extent, this experience summarizes the fact that the early multiprocessor work was too focused on flexibility which had not proven to be justified. High performance support for a standard software environment appears to be more important.

Based on the work covered in this section we note that high performance multiprocessor operating systems should:

1. enable high performance not only for large scale applications but also for standard UNIX workloads which can stress traditional implementations and
2. avoid contention and promote locality to ensure scalability.

2 Distributed Data Structures and Adaptation

In this section we focus on the research related to the use of distributed data structures and associated work into adaptation.

2.1 Distributed Data Structures

A number of systems proposed the use of distributed data structures albeit for varied motivations. In this section we review some of the more prominent systems related work.

2.1.1 Distributed Systems: FOs and DSOs

Fragmented Objects(FOs)[16, 80, 114] and Distributed Shared Objects(DSOs)[8, 63, 127] both explore the use of a partitioned object model as a programming abstraction for coping with the latencies in a distributed network environment. Fragmented Objects represent an object as a set of fragments which exists in address spaces distributed across the machines of a local area network but appears to the client as a single object. When a client invokes a method of an object it does so by invoking a method of a fragment local to its address space. The fragments, transparently to the client, communicate amongst each other, to ensure a consistent global view of the object. Local access to an object is synchronized via a local lock which guards all accesses. The Fragmented Objects work focuses on how to codify flexible consistency protocols within a general framework. This allows developers to implement objects composed of distributed replicas which hide their distributed nature while co-ordinating as necessary via network messages. In the case of Distributed Shared Objects, distributed processes communicate by accessing a distributed shared object instance. Each instance has a unique id and one or more interfaces. In order to improve performance, an instance can be physically distributed with its state partitioned and/or replicated across multiple machines at the same time. All protocols for communication, replication, distribution and migration are internal to the object and hidden from clients. This work focuses on the nature of wide area network applications and protocols such as that of the world wide web. For example, global uniform naming and binding is addressed while a course grain model of communication is assumed.

2.1.2 Language Support: CAs and pSather

Chien et al. introduced Concurrent Aggregates(CAs)[31] as a language abstraction for expressing parallel data structures in a modular fashion. The work is concerned with the language issues of supporting a distributed parallel object model for efficient construction of parallel applications in a message passing environment. As is typical of many concurrent object oriented programming systems, an Actor model[1] is adopted. In this model objects are self-contained, independent

components of a computing system that communicate by asynchronous message passing. Such models typically impose a serialization of message processing by the use of message queues, thus simplifying the programmer's task by eliminating concurrency issues internal to an object. Chien et al. studied a language extension called an Aggregate that permits object invocation to occur in parallel. An instance of an Aggregate has a single external name and interface, however each invocation is translated by a runtime environment to an invocation on an arbitrary representative of the aggregate. The number of representatives for an aggregate is declared by the programmer as a constant. Each representative contains local instances of the Aggregate fields. The language supports the ability for one representative of an Aggregate to name/locate and invoke methods of the other representatives in order to permit scatter and gather operations via function shipping and more complex cooperation.

Rather than focusing on an Actor based object oriented model, pSather[79] explores language and associated runtime extensions for data distribution on NUMA multiprocessors to Sather, an Eiffel-like research language. Specifically, it adds threads, synchronization and data distribution to Sather. Unlike the previous work discussed, pSather advocates orthogonality between object orientation and parallelism; it introduces new language constructs independent of the object model for data distribution. Unlike Chien's Concurrent Aggregates, it does not impose a specific processing/synchronization model nor does it assume the use of system-provided consistency models/protocols like Distributed Shared Objects or Fragmented Objects. In his thesis Chu-Cheo proposes two primitives for replicating reference variables and data structures such that a replica is located in each cluster of a NUMA multiprocessor. Since the data distribution primitives are not integrated into the object model, there is no native support for hiding the distribution behind an interface. There is also no support for dynamic instantiation or initialization of replicas, nor facilities for distributed reclamation. Note that this work assumes that every replicated data element will have a fixed mapping of one local representative for every cluster with respect to the hardware organization and that initialization, in general, will be done for all replicas at once prior to the data element's use.

The author of pSather does explore the advantages of several distributed data structures built on top of the primitives introduced for a set of data parallel applications on a NUMA multiprocessor. Given the regular and static parallelism of data parallel applications, the semantics of the data structures explored are limited and restricted. The data structures considered at include: a workbag (a distributed work queue), a distributed hash table, a distributed matrix and a distributed quad tree. The primary focus is to validate pSather implementations of various applications; this includes the construction of the distributed data structures and the applications themselves. Although a primary focus is on evaluating the overall programmer experience using the parallel primitives and the resulting library of data structures, Choew does evaluate the performance of the resulting applications with respect to scale. The author highlights some of the tradeoffs in performance with respect to remote accesses given variations in the implementation of the data structures and algorithms of the applications. He points out that minimizing remote accesses, thus enhancing locality, is key to good performance for the applications studied. Unfortunately it appears that the author does not compare the performance of the distributed data structures to centralized implementations so it is difficult to get a feeling for the exact benefits of distribution.

Finally it is not clear that the distributed data structures that are explored in the pSather work are appropriate for systems software which is dynamic and event-driven in nature, as opposed to the regular and static parallelism of scientific applications. For example consider the semantics of

the distributed hash table studied by Chu-Choe:

- state stored in the hash table is monotonically increasing; once inserted an item will never be removed
- each local hash table employs coarse grain locking
- there are no facilities for modifying stored state
- there are no facilities for hashing distributed state

The restrictions are not necessarily problematic for data parallel applications which would utilize such a hash table to publish progressive results of a large distributed calculation. However, it would be difficult to use such a hash table as the cache of page descriptors for the resident memory pages, when implementing an operating system. Systems code, in general, cannot predict the parallel demand on any given data structure instance and hence the data structure creation, initialization, sizing and concurrency must be dynamic in nature. Additionally, systems software must be free to exploit aggressive optimizations which simple semantics may not permit. For example, once it is determined that a distributed hash table is going to be used, there may be other optimizations that present themselves by distributing the data fields of the elements that are being stored. In the case of page descriptors, rather than just storing a reference to a page descriptor in the local replicas of the hash table, one might want to allow each replica to store local versions of the access bits of the page descriptor in order to avoid global memory access and synchronization on the performance critical resident page fault path.

2.1.3 Topologies and DSAs

In the early 1990's there was considerable interest in message passing architectures which typically leveraged a point to point interconnection network and the promise of unlimited scalability. Such machines, however, did not provide a shared memory abstraction. They were typically used for custom applications, organized as a collection of threads which communicate via messages, aware and tuned for the underlying interconnection geometry of the hardware platform. In an attempt to ease the burden and generalize the use of such machines, Bo et al. proposed OS support for a distributed primitive called a Topology[108]. It attempts to isolate and encapsulate the communication protocol and structure among a number of identified communicating processes via a shared object oriented abstraction. A topology's structure is described as a set of vertices and edges where the vertices are mapped to physical nodes of the hardware and edges capture the communication structure between vertices. An example would be an inverse broadcast which encapsulates the necessary communication protocol to enable the aggregation of data from a set of distributed processes. The Topology is implemented to minimize the number of nonlocal communications for the given architecture being used. They are presented as heavy-weight OS abstractions requiring considerable OS support for their management and scheduling. Applications make requests to the operating system to instantiate, configure and use a Topology. Each type encapsulates a fixed communication protocol predefined by the OS implementors. Applications utilize a Topology by creating, customizing and invoking an instance; binding the application processes to the its vertices, specifying application functions for message processing, specifying the state associated with each vertex and invoking the specified functions by sending messages to the instance.

Motivated by significant performance improvements obtained with distributed data structures and algorithms on a NUMA multiprocessor for Traveling Sales Person (TSP) programs[89], Clemencon et al.[32, 33] proposed a distributed object model called Distributed Shared Abstractions (DSAs). This work is targeted at increasing the scalability and portability of parallel programs via a reusable user level library which supports the construction of objects which encapsulate a DSA. Each object is composed of a set of distributed fragments similar to Fragmented Objects and Distributed Shared Objects discussed earlier. The runtime implementation and model are, however, built on the author's previous work on Topologies.

Akin to the work in pSather, Clemencon et al. specifically cited distribution as a means for improving performance by improving locality and reducing remote access on a multiprocessor. The authors asserted the following benefits:

- potential reductions in contention of access to an object, since many operations on the object will access only locally stored copies of its distributed state
- decreases in invocation latencies, since local accesses are faster than remote accesses, and
- the ability to implement objects such that they may be used on both distributed and shared memory platforms, therefore increasing the portability of applications using them.

As implied by the last two points, influenced by their earlier work and motivated by portability to distributed systems, the authors assumed a message passing communication model between fragments. Despite the message passing focus, the performance results and analysis presented are very relevant to the Clustered Object work done at the University of Toronto[3, 4, 5, 47, 48].

The authors observed two factors which affect performance of shared data structures on a NUMA multiprocessor:

1. contention due to concurrent access (synchronization overhead) and
2. remote memory access costs (communication overhead).

They observed that distribution of state is key to reducing contention and improving locality. When comparing multiple parallel versions of the TSP program they found that using a centralized work queue protected by a single spin lock was limited to a 4 times speedup whereas a 10 times speed up was possible with a distributed work queue. Further, they found that by leveraging application specific knowledge they were able to specialize the distributed data structure implementation to further improve performance. This demonstrated that despite additional complexities, distributed implementations with customized semantics can significantly improve application performance. Note that in the results presented, the authors do not isolate the influence of synchronization overhead versus remote access in their results[6].

Based on performance studies of TSP on two different NUMA systems, the authors state that, "Any large scale parallel machine exhibiting NUMA memory properties must be used in a fashion similar to distributed memory machines, including the explicit distribution of the state and functionality of programs' shared abstractions." They further note that machines with cache coherency such as the KSR do not alleviate the need for distribution. Cache coherency overheads further enforce the need for explicit distribution of state and its application-specific management, in order to

[6]Concurrent centralized implementations which employ fine grain locking or lock free techniques were not considered.

achieve high performance. Based on initial measurements done on small scale SGI multiprocessors, the authors predicted that given the trends in the disparity between processor speeds and memory access times, distributed data structures will be necessary even on small scale multiprocessors. They point out an often over-looked aspect of the use of shared memory multiprocessors:

> Multiprocessors are adopted for high performance and shared memory multiprocessors are claimed as superior to message passing systems as they offer a simple convenient programming model. However, to achieve high performance on shared memory multiprocessors requires the use of complex distributed implementations akin to those used on message passing systems.

This aspect motivates the DSA work and to a partial extent the Clustered Object work, which attempts to encapsulate, help limit the impact and permit reuse of the additional complexity.

The actual detailed model and implementation of the DSA runtime appears to have been strongly influenced by: 1) the previous Topology work, 2) assumptions about application use and 3) the desire to be portable to distributed systems. This resulted in a heavy-weight facility which is only appropriate for coarse grain, long lived, fully distributed application objects whose access pattern is well known. Limiting characteristics include:

- expensive message based interface to objects where object access is 6 times more expensive than procedure invocation,
- a restricted scheduling and thread model,
- support only for inter-fragment communication via remote procedure calls and no support for direct shared memory inter-fragment access,
- expensive binding operations that preclude short lived threads,
- no support for efficient allocation or deallocation of objects, and
- manual initialization that is serial in nature.

The use of the DSA library seems to have been very limited with only the distributed work queue explored in the context of the TSP application. Given limited use and lack of experience in a distributed systems environment, it is unclear if the restrictive fully distributed message passing model is justified.

In summary there are three key results from the DSA work:

1. A shared memory multiprocessor parallel application's performance can greatly benefit from the use of distributed data structures.
2. Better performance can be achieved by exploiting application-specific knowledge to tailor distributed implementations, as opposed to generic object implementations or general distributed methodologies which impose fixed models for the sake of generality.
3. Locality optimization can be effectively employed at the level of abstract data types.

2.1.4 Clustered Objects

In 1995, the University of Toronto group proposed the use of Clustered Objects[94] to encapsulate distributed data structures for the construction of NUMA multiprocessor systems software. Like the Concurrent Aggregates, Fragmented Objects, Distributed Shared Objects and DSAs; Clustered Objects enable distribution within an object oriented model with the motivation to enable NUMA locality optimizations, like those advocated by pSather and DSAs but accounting for the unique requirements of systems software. Unlike the previous approaches, Clustered Objects were designed for systems level software and uniquely targeted at ubiquitous use, where every object in the system is a Clustered Object with its own potentially unique structure. They are designed to support both centralized and distributed implementations, fully utilizing the hardware support for shared memory where appropriate and do not impose any constraints on the access model or computational model either externally or internally to an object. The supporting mechanisms are light-weight and employ lazy semantics to ensure high performance for short lived system threads. Clustered Objects are discussed in more detail in section 3.3.1.

2.1.5 Distributed Hash Tables

Finally, there has been a body of work which has looked at the use of distributed hash tables in the context of specific distributed applications, including distributed databases[43], cluster based internet services[54], peer-to-peer systems[35] and general distributed data storage and look up services[22]. In general, the work done on distributed data structures for distributed systems is primarily concerned with the exploration of the distributed data structure as a convenient abstraction for constructing network based applications, increasing robustness via replication. Like the Fragmented Object and Distributed Shared Object work, the use of distributed hash tables seeks a systematic way of reducing and hiding network latencies. The coarse grain nature and network focus to this type of work results in few insights for the construction of performance critical and latency sensitive shared memory multiprocessor systems software.

2.2 Adaptation for Parallelism

Motivated by the sensitive and dynamic demands of parallel software, researchers have proposed leveraging the strict modularity imposed by object orientation in order to facilitate adaptability and autonomic systems/computing. Unlike the previous work discussed, which focused on object orientation for the sake of flexibility in system composition and configuration, the work on adaptation focuses on using the encapsulation enforced by object boundaries to isolate the computations which can have sensitive parallel performance profiles. Adaptation is enabled by introducing mechanisms which allow reconfiguration of the isolated components either by adjusting parameters of the component or complete replacement. The most appropriate configuration can then be chosen in order to maximize the performance based on the current demands. There has been considerable work in the area of adaptable or autonomic computing. We restrict our discussion to the work on adaptation for the sake of parallel performance.

In CHAOSarc[51], the authors explore a parallel real-time system for robot control using an object oriented decomposition. The work studies predictability within the context of a highly configurable parallel programming environment. The focus on predictability leads to a fine grain classification of computation and locking semantics in order to be able to match application demand to performance characteristics of systems software. This resulted in a large configuration space in

which the components used to implement a computation must match the runtime constraints and aspects of the computation. This lead the authors to consider reconfiguration of objects and the use of adaptive objects[89].

Mukherjee et al. explore the costs, factors and tradeoffs with building adaptive systems in the context of implementing parallel solutions to the Traveling Sales Person problem. They attempt to construct and formalize a theory of adaptation, proposing a categorization of objects as: non-configurable, reconfigurable, and adaptable. Reconfigurable objects permit external changes to mutable properties which affect their operation. Adaptable objects encapsulate a reconfigurable object, monitoring facility, adaption policy and reconfiguration mechanism. The utility of the formalization proposed is unclear and no concrete implementation or mechanisms for supporting it is given. The authors do provide strong motivation with respect to parallel performance of the TSP applications. They illustrate the benefits of an adaptive lock object which modifies its spin and blocking behavior based on the demands it experiences versus using a static lock object. A 17% performance improvement was observed when studying a centralized algorithm of TSP which uses shared data structures and a 6.5% improvement when considering a distributed implementation of TSP. The authors do point out that the distributed implementation has better base performance and has less dependency on lock implementation but they only consider adaptation of the lock object itself.

Motivated by the results above, the authors attempted to extend the single application benefits observed to the entire system by enabling adaptation in the system layers. They proposed an architecture for a reconfigurable parallel micro-kernel called KTK[50]. They assert that:

- Run-time behavior differs across multiple applications and across multiple phases of a single application.
- Operating system kernel configurations can provide high-performance applications with the policies and mechanisms best suited to their characteristics and to their target hardware. The authors appeal to the standard flexibility arguments of previous object oriented systems.
- Dynamic kernel reconfiguration can improve performance by satisfying each application's behavior.
- Efficient application state monitoring can detect changes in application requirements which are not known prior to program execution time.

The KTK architecture proposes a number of core kernel components which support configuration, reconfiguration and adaptation. Each object identifies a set of attributes which are mutable and support some form of arbitration with respect to the attribute change. The micro-kernel should provide monitoring of kernel components in order to facilitate adaptation policies and also provide support for application defined adaptation policies. Although the case for system reconfiguration is compelling it is unclear to what extent the KTK prototype achieved the goals set out or to what extent application performance was improved or facilitated.

Motivated by KTK, Silva et al. attempt to factor the support for reconfiguration of a parallel object-based software into a library called CTK[116] in order to facilitate a programming model that enables the expression and implementation of program configuration and the runtime support for performance improvements by changing configuration. Building on the KTK model, CTK adopts a model which incorporates specification of mutable attributes and policies for attribute

change. It also supports efficient on-line capture of performance data in order to enable dynamic configuration. Utilizing the group's previous work, CTK explores a distributed work queue DSA(see 2.1.3) with respect to reconfiguration in the solution of Traveling Sales Person solutions. The work proposes the use of a custom language that incorporates the expression of object attributes and other facilities. CTK focuses on expressive and general support for the specification of dynamic policies and attribute change. Evaluation was limited to a single user-level application and the suitability of CTK in an operating system is unclear.

Based on similar motivations, the K42 group has explored the integration of mechanisms for enabling runtime adaptation of Clustered Objects [4, 5, 65, 66, 118]. Unlike the KTK and CTK work, the K42 work focuses on the mechanism for enabling efficient replacement of a distributed object rather than general issues of adaptation. The work is uniquely focused on ensuring low overheads so that the use of object replacement can be used pervasively in the systems software. Furthermore the K42 work explores the use of adaptation with respect to system objects which are in critical operating system paths and are subject to unpredictable dynamic concurrent access.

2.3 Summary

Previous work on distributed data structures and adaptation suggests that a high performance operating system should:

- Enable data distribution, as highly concurrent shared memory multiprocessor software requires distributed data structures to ensure high concurrency and low latency. Since operating systems must reflect the concurrency of the workload they, by definition, need to enable the highest possible degree of concurrency in order not to limit application performance.
- Utilize object orientation to help cope with the complexity introduced by data distribution but ensure low overhead.
- Support adaptation in order to support variability in parallel workloads and OS demands.

3 Modern Multiprocessor Operating Systems Research

Many of the research systems discussed in section 1 attempted to explore meta ideas on structure and mechanisms for OS's to achieve scalability but few of these systems have had significant impact on main stream multiprocessor computing. This is in part due to the nonstandard programming environments advocated, resulting in an inability to evaluate the systems within the context of standard workloads. The research that has had an impact has either addressed isolated synchronization problems whose solutions can be directly applied to current industrial systems or, as in the case of the Mach project, have had commercial compatibility as a primary goal. Given the large effort required to produce a compatible system, Mach has primarily been the only research MP OS to have been explored by the research community. Mach's main goal was to explore the use of a micro-kernel architecture for flexibility while supporting a UNIX compatible interface. A number of groups opted to use Mach as a platform for multiprocessor research, hoping to leverage its portability and UNIX compatibility, while being able to explore novel ideas given its flexible structure. Unfortunately, the lack of a multiprocessor performance focus in Mach's infrastructure resulted in systems whose performance for standard workloads were inferior to commercial systems and hence any benefits to the new systems where dismissed.

There have been a number of papers published on performance issues in shared-memory multiprocessor operating systems, but mostly in the context of resolving specific problems in a specific system [19, 23, 27, 87, 100, 122]. These systems were mostly uniprocessor or small-scale multiprocessor systems trying to scale up to larger systems. Other work on locality issues in operating system structure was mostly either done in the context of earlier non-cache-coherent NUMA systems [25], or, as in the case of Plan 9, was not published [98]. Two projects that were aimed explicitly at large-scale multiprocessors were Hive [24], and Hurricane [126]/Tornado [48]. Both independently chose a clustered approach by connecting multiple small-scale systems to form either, in the case of Hive, a more fault tolerant system, or, in the case of Hurricane, a more scalable system. However, both groups ran into complexity problems with this approach and both have moved on to other approaches; namely Disco [18] and Tornado, respectively.

3.1 Characteristics of Scalable Machines

SMP architectures present the programmer with the familiar notion of a single address space within which multiple processes exist, possibly running on different processors. Unlike a message-passing architecture, an SMP does not require the programmer to use explicit primitives for the sharing of data. Hardware-supported shared memory is used to share data between processes, even if running on different processors. Many modern SMP systems provide hardware cache coherence to ensure that the multiple copies of data in the caches of different processors (which arise from sharing) are kept consistent.

Physical limits, cost efficiency and desire for scalability have lead to SMP architectures that are formed by inter-connecting clusters of processors. Each cluster typically contains a set of processors and one or more memory modules. The total physical memory of the system is distributed as individual modules across the clusters, but each processor in the system is capable of accessing any of these memory modules in a transparent way, although it may suffer increased latencies when accessing memory located on remote clusters. SMPs with this type of physical memory organization are called Non-Uniform Memory Access (NUMA) SMPs. Examples of such NUMA SMP architectures include Stanford's Dash[77] and Flash [72] architectures, University of Toronto's Hector [131] and NUMAchine [130] architectures, Sequent's NUMA-Q [113] architecture and SGI's Cray Origin2000[74]. NUMA SMPs that implement cache coherency in hardware are called CC-NUMA SMPs. In contrast, multiprocessors based on a single bus have Uniform Memory Access times and are called UMA SMPs.

It can be difficult to realize the performance potential of a CC-NUMA SMP. The programmer must not only develop algorithms that are parallel in nature, but must also be aware of the subtle effects of sharing both in terms of correctness and in terms of performance. These effects include:

- Increased communication latencies due to the coherence protocols and distribution of physical memory
- The use of explicit synchronization, needed to ensure correctness of shared data, can induce additional computation and communication overheads
- False sharing reduces the effectiveness of the hardware caches and results in the same high cache coherence overhead as true sharing. False sharing happens when independently accessed data is co-located in the same cache line are requires careful data layout in memory to avoid.

Memory latencies and cache consistency overheads can often be reduced substantially by designing software that maximizes the locality of data accesses. Replication and partitioning of data are primary techniques used to improve locality. Both techniques allow processes to access localized instances of data in the common case. They decrease the need for remote memory accesses and lead to local synchronization points that are less contended.

Other more course-grain approaches for improving locality in general SMP software include automated support for memory page placement, replication and migration [73, 81, 129] and cache affinity aware process scheduling [40, 55, 82, 119, 128].

The two key factors affecting multiprocessor software and in particular OS performance, besides the policies it provides and the algorithms it uses, are memory system and locking behaviors. The key to maximizing memory system performance on a multiprocessor is to minimize the amount of (true and false) sharing, particularly for read-write data structures. Not paying careful attention to sharing patterns can cause excessive cache coherence traffic, resulting in potentially terrible performance due to the direct effect of the extra cache misses and to the secondary effect of contention in the processor-memory interconnection network and at the memory itself. For example, in a study of IRIX on a 4-processor system, Torrellas found that misses due to sharing dominated all other types of misses, accounting for up to 50 percent of all data cache misses [125]. Similarly, Rosenblum noted in a study of an 8 processor system that 18 percent of all coherence misses were caused by the false sharing of a single cache line containing a highly shared lock in the IRIX operating system [104].

In larger systems the secondary effects become more significant. Moreover, in large NUMA systems, it is also necessary to take memory access latencies into account, considering that accesses to remote memory modules can cost several times as much as accesses to local memory modules. The significance of this was observed by Unrau et al., where, due to the lack of physical locality in the data structures used, the uncontended cost of a page fault increased by 25 percent when the system was scaled from 1 to 16 processors [126].

The sharing of cache lines can often be reduced by applying various replication and partitioning strategies, whereby each processor (or set of processors) is given a private copy or portion of the data structure. The same strategy also helps increase locality, aiding larger NUMA systems. However, replication and partitioning requires greater work in managing and coordinating the multiple data structures.

Despite disputes about the details, it is widely accepted that scalable large multiprocessor hardware is realizable, given a hardware supported distributed shared memory architecture. But such hardware will have properties which require special attention on the part of systems software if general purpose workloads are to be supported.

3.2 Operating Systems Performance

Poor performance of the operating system can have considerable impact on application performance. For example, for parallel workloads studied by Torrellas et al., the operating system accounted for as much as 32-47% of the non-idle execution time[125]. Similarly Xia and Torrellas showed that for a different set of workloads, 42-54% of time was spent in the operating system [135], while Chapin et al. found that 24% of total execution time was spent in the operating system[23] for their workload.

To avoid the operating system from limiting application performance, it must be highly concurrent. The traditional approach to developing SMP operating systems has been to start with a

uniprocessor operating system and to then successively tune it for concurrency. This is achieved by adding locks to protect critical resources. Performance measurements are then used to identify points of contention. As bottlenecks are identified, additional locks are introduced to increase concurrency, leading to finer-grained locking. Several commercial SMP operating systems have been developed as successive refinements of a uniprocessor code base. Denham et al. provides an excellent account of one such development effort [39]. This approach is ad hoc in nature, however, and leads to complex systems, while providing little flexibility. Adding more processors to the system, or changing access patterns, may require significant re-tuning.

The continual addition of locks can also lead to excessive locking overheads. In such cases, it is often necessary to design new algorithms and data structures that do not depend so heavily on synchronization. Examples include: Software Set Associative Cache architecture developed by Peacock et al.[95, 96], kernel memory allocation facilities developed by McKenny et al.[88], fair fast scalable reader-writer locks developed by Krieger et al.[71], performance measurement kernel device driver developed by Anderson et al.[2] and the intra-node data structures used by Stets et al.[120].

The traditional approach of adding locks and selectively redesigning also does not explicitly lead to increased locality. Chapin et al. studied the memory system performance of a commercial Unix system, parallelized to run efficiently on the 64 processor-large Stanford DASH multiprocessor[23]. They found that the time spent servicing operating system data misses was three times higher than time spent executing operating system code. Of the time spent servicing operating system data misses, 92% was due to remote misses. Kaeli et al. showed that careful tuning of their operating system to improve locality allowed them to obtain linear speedups on their prototype CC-NUMA system, running OLTP benchmarks[69].

In the early to mid 1990's, researchers identified memory performance as critical to system performance [23, 26, 86, 104, 125]. They noted that cache performance and coherency are critical aspects of SMMP hardware which must be taken into account by software and that focusing on concurrency and synchronization, and its performance, is not enough.

Rosenblum et al. [104] explicitly advocated that operating systems must be optimized to meet the demands of users for high performance. However, they point out that operating systems are large and complex and the optimization task is difficult and, without care, tuning can result in increased complexity with little impact to the end-user performance. The key is to focus optimization by identifying performance problems. They studied three important workloads:

1. Program Development Workload
2. Database Workload
3. Large simulations which stress the memory subsystem

They predicted that even for small scale SMMP's, coherency overheads induced by communications and synchronization overheads would result in MP OS services consuming 30% - 70% more resources than uniprocessor counterparts. They also observed that larger caches do not help alleviate coherency, so the performance gap between MP OSs and UP OSs will grow unless there is focus on kernel restructuring to reduce unnecessary communication. They pointed out that as the relative cost of coherency misses goes up, programmers must focus on data layout to avoid false sharing and that preserving locality in scheduling is critical to ensuring effectiveness of caches. Rescheduling

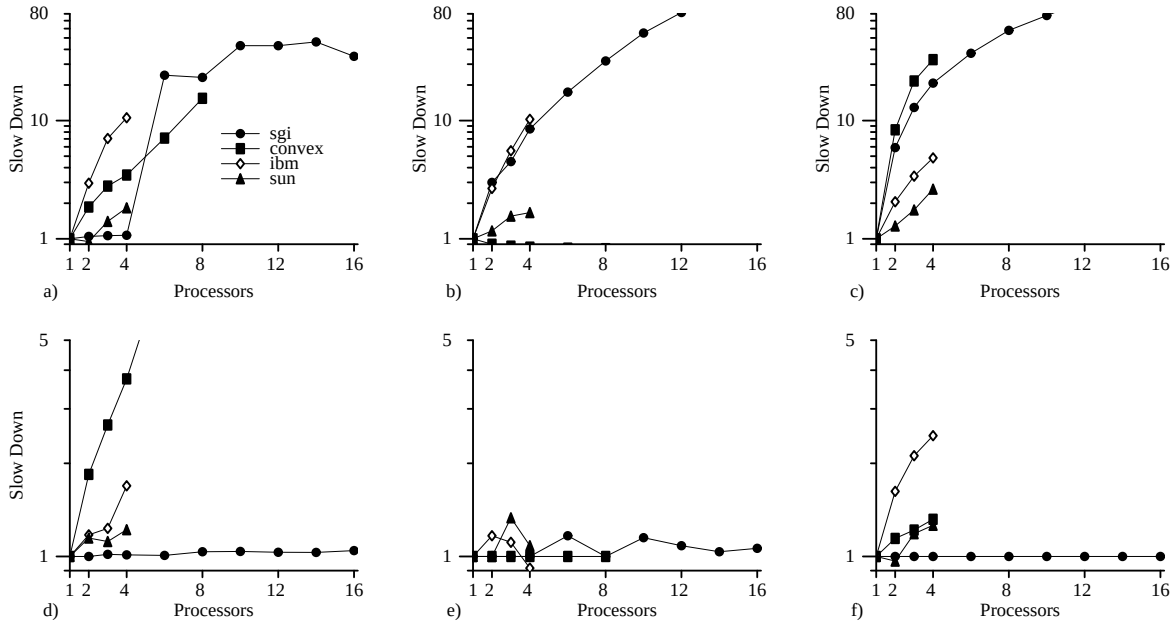

Figure 1: Microbenchmarks across all tests and systems. The top row (a–c) depicts the multi-threaded tests with n threads in one process. The bottom row (d–f) depicts the multiprogrammed tests with n processes, each with one thread. The leftmost set (a,d) depicts the slowdown for in-core page fault handling, the middle set (b,e) depicts the slowdown for file stat, and the rightmost set depicts the slowdown for thread creation/destruction. The systems on which the tests were run are: SGI Origin 2000 running IRIX 6.4, Convex SPP-1600 running SPP-UX 4.2, IBM 7012-G30 PowerPC 604 running AIX 4.2.0.0, Sun 450 UltraSparc II running Solaris 2.5.1.

processes on different processors can result in coherency traffic on kernel data structures. The research at the University of Toronto has been targeted at addressing these issues.

Figure 1 presents results gathered by Gamsa et al.[46] of simple micro-benchmarks run on a number of commercial SMP operating systems. The micro-benchmarks are of three separate tests: in-core page faults, file stat and thread creation, each with n worker threads performing the operation being tested:

Page Fault Each worker thread accessed a set of in-core unmapped pages in independent (separate mmap) memory regions.

File Stat Each worker thread repeatedly fstated an independent file.

Thread Creation Each worker successively created and then joined with a child thread (the child does nothing but exit).

Each test was run in two different ways; multi-threaded and multi-programmed. In the multi-threaded case the test was run as described above. In the multi-programmed tests, n instances of the test were started with one worker thread per instance. Although the commercial systems do reasonably well on the multiprogrammed tests in general, they suffer considerable slow downs on the multithreaded tests. This evidence implies that the existing techniques used by commercial

systems are insufficient in their ability to exploit the concurrency of these simple multi-threaded micro-benchmark applications.

Given that the currently accepted architecture for building large scalable multiprocessors is as an interconnection of clusters of computation nodes, both the groups at UofT and Stanford, wanting to explicitly explore large scale machines, chose to directly reflect the hardware structure in the operating system.

3.2.1 Hurricane

Unlike much of the previous MP OS research efforts, the University of Toronto chose to first focus on multiprocessor performance, thereby uniquely motivating, justifying and evaluating the operating system design and implementation based on the structure and properties of scalable multiprocessor hardware. Motivated by the Hector multiprocessor[131], representative of the architectures for large scale multiprocessors of the time[12, 45, 77, 97], the group choose a simple structuring for the operating system which directly mirrored the architecture of the hardware, hoping to leverage the strengths of the hardware structure while minimizing its weakness.

By focusing on performance rather than flexibility, the Hurricane group was motivated to acknowledge, analyze and identify the unique operating system requirements with respect to scalable performance. Particularly, based on previous literature and queuing theory analysis the following guidelines where identified[126]:

Preserving parallelism The operating system must preserve the parallelism afforded by the applications. If several threads of an executing application (or of independent applications running at the same time) request independent operating system services in parallel, then they must be serviced in parallel; otherwise the operating system becomes a bottleneck, limiting scalability and application speedup. Critically, it was observed that an operating system is demand driven and it's services do not utilize parallelism, thus parallelism can only come from application demand. Therefore, the number of operating system service points must increase with the size of the system and the concurrency available in accessing the data structures must grow with the size of the system to make it possible for the overall throughput to increase proportionally.

Bounded overhead The overhead for each independent operating system service call must be bounded by a constant, independent of the number of processors. If the overhead of each service call increases with the number of processors, the system will ultimately saturate, so the demand on any single resource cannot increase with the number of processors. For this reason, system wide ordered queues cannot be used and objects cannot be located by linear searches if the queue lengths or search lengths increase with the size of the system. Broadcasts cannot be used for the same reason.

Preserve locality The operating system must preserve the locality of the applications. It is important to consider the memory access locality in large-scale systems because, for example, many large-scale shared memory multiprocessors have non-uniform access (NUMA) times, where the costs of accessing memory is a function of the distance between the accessing processor and the target memory, and because cache consistency incurs more overhead in a large system. Specifically it was noted that locality can be increased a) by properly choosing and placing data structures within the operating system, b) by directing requests from the

application to nearby service points, and c) by enacting policies that increase locality in the applications' memory accesses. For example, policies should attempt to run the processes of a single application on processors close to each other, place memory pages in proximity to the processes accessing them, and direct file I/O to devices close by. Within the operating system, descriptors of processes that interact frequently should lie close together, and memory mapping information should lie close to the processors which must access them to handle page faults.

Although some of these guidelines have been identified by other researchers[9, 117] we are not aware of other general purpose shared memory multiprocessor operating systems which pervasively utilize them in their design. Over the years, these guidelines have been refined but have remained a central focus of the body of research work done at the University of Toronto.

Hurricane, in particular, employed a course grain approach to scalability, where a single large scale SMMP was partitioned into clusters of a fixed number of processors. Each cluster ran a separate instance of a small scale SMMP operating system, cooperatively providing a single system image. This approach is now being used in one form or another by several commercial systems, for example in SGI's Cellular IRIX. Hurricane, attempted to directly reflect the hardware structure, utilizing a collection of separate instances of a small-scale SMP operating system, one per-hardware cluster. Implicit use of shared memory is only allowed within a cluster. Any co-ordination/sharing between clusters occurs using a more expensive explicit facility. It was hoped that any given request by an application could in the common case be serviced on the cluster on which the request was made with little or no interaction with other clusters. The fixed clustering approach limits the number of concurrent processes that can contend on any given lock to the number of processors in a cluster. Similarly, it limits the number of per-processor caches that need to be kept coherent. The clustered approach also ensures that each data structure is replicated into the local memory of each cluster.

Despite many of the positive benefits of clustering, it was found that[48]: (*i*) the traditional within-cluster structures exhibit poor locality which severely impacts performance on modern multiprocessors, (*ii*) the rigid clustering results in increased complexity as well as high overhead or poor scalability for some applications, and (*iii*) the traditional structures as well as the clustering strategy make it difficult to support the specialized policy requirements of parallel applications.

Related work at Stanford into the Hive operating system[24] focused on locality, firstly as a means of providing fault containment and secondly as a means for improving scalability.

3.3 Virtual Machine Partitioning

Some groups have pursued strict partitioning as a means for leveraging the resources of a multiprocessor[18, 53, 103]. Rather than trying to construct a kernel which can efficiently support a single system image they pursue the construction of a kernel which can support the execution of multiple virtual machines (VMs). By doing so, the software within the virtual machines is responsible for extracting the degree of parallelism it requires from the resource allocated to the VM it is executing in. Rather than wasting the resources of a large scale machine on a single OS instance, incapable of efficiently utilizing all the resources, the resources are partitioned across multiple OS instances. There are three key advantages to this approach:

1. The underlying systems software which enables the partitioning does not itself require high concurrency.

2. Standard workloads can be run by leveraging the Virtual Machine approach to run standard OS instances.

3. Resources of a large scale machine can be efficiently utilized with standard software albeit without native support for large scale applications and limited sharing between partitions.

To some extent this approach can be viewed as a tradeoff which permits large scale machines to be leveraged using standard systems software.

The Wisconsin Wind Tunnel[103] was one of the first projects to explore the use of Virtual Machines in the construction of a multiprocessor operating system environment. The goal was to create a parallel simulation environment for studying parallel computing. In order to study cache coherent shared memory multiprocessors on top of the Thinking Machines CM-5, a large scale distributed memory message passing system, a virtual machine-like architecture was used. Building on top of the CM-5 architecture and software base, a small kernel, which partitioned the system among multiple virtual machine-like executives, was developed. Each executive provided a virtual machine environment to the software running on top of it. The underlying kernel which partitioned the machine among the executives also provided a shared memory abstraction. Given the goal of simulation, the approach of partitioning an MP system across multiple VM instances was not pursued as a general system architecture. The authors do note that the approach of using a shared kernel, which provides coarse grain scheduling and memory arbitration to provide autonomy of multiple executives, is unique with respect to the approach of starting with multiple autonomous micro-kernels which then co-operate to provide a shared abstraction. They argue that this approach allows each partition the flexibility of matching the necessary scheduling and resource sharing required for the applications running in it.

In the late 1990's the same general approach was used to define a general systems architecture by the Disco[18, 53] project at Stanford based on the Hive experience. Specifically, they attempted to address the problem of extending modern operating systems to run efficiently on large scale shared memory multiprocessors by partitioning the system into a virtual cluster, thus avoiding scalability bottlenecks in standard operating systems. The authors clearly acknowledged the tradeoff of such an approach and the orthogonal need to continue pursuing scalable single system image operating systems research. The authors emphatically asserted that the limiting factor to general use of large scale machines has been poor systems software support. The approach they took side steps the limitations by efficiently partitioning such systems to enable the execution of multiple instances of commodity operating systems. Despite acknowledging the need to pursue scalable operating systems research such as Tornado and K42 they pointed out that considerable investment in development effort and time was required before such systems can reach commercial maturity.

It is worth noting two fundamental points raised by this research:

1. Standard operating systems do not effectively support large scale multiprocessors.

2. Despite point 1, the standard environment offered by commodity systems is compelling enough to justify partitioning of the hardware.

This implies that a new scalable system must support the standard operating environment of a commodity system if it is to be effective.

3.3.1 Tornado

One of the key observations made by the Hurricane group was that fixed cluster sizes were too restrictive, did not perform well for all workloads, and introduced considerable complexity. Although an attempt was made to determine the optimal configuration, it was realized that each service and its data structures required different degrees of clustering. Some data structures (eg. Page Descriptor Index) are best shared across the entire system, while other data structures (eg. Ready Queues) have better performance if they were replicated on a per-processor basis. This implied that greater flexibility with respect to the cluster sizes was required than was offered by the fixed clustering of Hurricane and Hive. It was concluded from the experiences with Hurricane that the locality attributes of data structures need to be expressed and managed on a per-data structure basis. This was a key motivation for Tornado.

The natural outgrowth of the Hurricane experience was to build an operating system in which each data structure could specify its own clustering size. Tornado served as the operating system for the NUMAchine multiprocessor [130]. In Tornado, unlike Hurricane, there is only one operating system instance, but clustering is provided for on a per-object basis. Tornado is implemented in C++ using an object oriented structure, with all operating system components being developed from scratch specifically for multiprocessors. The system components were designed with the primary overriding design principle of mapping any locality and independence that might exist in OS requests from applications to locality and independence in the servicing of these requests in the operating systems and system servers.

The object oriented design was chosen for multiprocessor performance benefits. More specifically, the design of Tornado was based on the observations that: (*i*) operating systems are driven by the request of applications on virtual resources, (*ii*) to achieve good performance on multiprocessors, requests to different resources should be handled independently, that is, without accessing any common data structures and without acquiring any common locks, and (*iii*) the requests should, in the common case, be serviced on the same processor they are issued on. This is achieved in Tornado by adopting an object oriented approach where each virtual and physical resource in the system is represented by an independent object so that accesses on different processors to different objects do not interfere with each other. Details of the Tornado operating system can be found in [46, 48].

Tornado introduced *Clustered Objects*[3, 48, 94], which allow an object to be partitioned into *representative objects*, where independent requests on different processors are handled by different representatives of the object in the common case. Thus, simultaneous requests from a parallel application to a single virtual resource (i.e., page faults to different pages of the same memory region) can be handled efficiently preserving as much locality as possible. However, a CO, despite potentially being distributed, appears to the client as a single object.

A Clustered Object is identified by an address space unique identifier. The identifier locates a per-processor representative object for the Clustered Object. All accesses to a Clustered Object on a processor are directed to a specific representative. To allow for more efficient use of resources, the representatives of a Clustered Object can be instantiated on first use. All the representatives of a Clustered Object are managed via a special per-Clustered Object management object. The management object is responsible for instantiation, deletion and assignment of representatives to processors. A Clustered Object can have a single shared representative that is assigned to all processors, a representative per-processor or any a configuration in between.

An operating system infrastructure is needed to implement Clustered Objects efficiently. In

Tornado this includes:

- an Object Translation Facility
- a Semi-automatic garbage collection scheme
- a Kernel Memory Allocation Facility (KMA)
- a Protected Procedure Call Facility (PPC)

The Object Translation Facility of Tornado is used to locate the processor-specific representative object when a Clustered Object is accessed on a given processor. It is implemented with two sets of tables per address space, a global table of pointers to per-Clustered Object management objects, and per-processor tables of pointers to representatives. The identifier for a Clustered Object is a common offset into the tables. If no representative exists for a given processor the global table is consulted to locate the Clustered Object's management object that manages all the representatives of the Clustered Object.

Tornado uses a semi-automatic garbage collection scheme that facilitates localizing lock accesses and greatly simplifies locking protocols. As a matter of principle, all locks are internal to the objects (or more precisely their representatives) they are protecting, and no global locks are used. In conjunction with Clustered Object structures, the contention on a lock is thus bounded by the clients of the representative being protected by the lock. With the garbage collection scheme, no additional (existence) locks are needed to protect the locks internal to the objects. As a result, Tornado's *locking strategy* results in much lower locking overhead, simpler locking protocols, and can often eliminate the need to worry about lock hierarchies.

The Kernel Memory Allocation facility manages the free pool of global and per-processor memory using a design similar to that of [88] with small block and NUMA extensions. It is capable of allocating memory from pages that are local to a target processor. By overloading the default new operator with a version that calls the localized memory allocation routines of the Kernel Memory Allocation facility, Tornado ensures that default object instantiation occurs with processor local memory. Hence, representatives and the data they allocate, automatically reside on the processors on which they are instantiated. This helps to reduce false sharing across clusters.

The Protected Procedure Call facility of Tornado supports interprocess communication. Protected Procedure calls allow one process within an address space to invoke the methods of an Object in another address space. A Protected Procedure Call is implemented as a light-weight protection domain crossing, executed on the same processor from which it is called. The Protected Procedure Call facility also provides the ability for a process executing on one processor, to invoke a procedure to be executed on another processor within the same address space, although at higher cost. This form of cross-processor Protected Procedure Calls is referred to as *Remote Procedure Calls*. Clustered Objects can use Remote Procedure Calls to implement function shipping as another form of cooperation between representatives.

In Tornado the majority of the system's objects were naive Clustered Objects using just a single representative. The Clustered Object work in Tornado focused on developing the underlying infrastructure and basic mechanisms[48] as well as an initial performance evaluation.[3].

4 Current Work at Toronto

The Hurricane work, and related efforts at characterizing multiprocessor performance, established the need to account for the characteristics of multiprocessor hardware in the OS structure. Tornado established that the parallel demands of the workload must also be reflected in the OS structure, showing that an object oriented decomposition could be utilized to map independence in the workload to a runtime which leverages locality aspects of the hardware to reduce sharing and improve scalability. Given the object oriented structure, Tornado also proposed mechanisms for integrating the use of distributed data structures in the basic object oriented support in order to permit more aggressive multiprocessor tuned implementations.

IBM began an effort to develop K42, a research operating system to explore scalability and novel user-level structure while providing both API and ABI compatibility with a standard OS. In an attempt to account for scalability within the basic design and structure IBM chose to base K42 on Tornado. Our group at the University of Toronto has closely collaborated with IBM to design and implement K42, with Toronto focusing on K42's scalability.

We are currently exploring the use of distributed data structures in K42. Our effort attempts to standardize the use of the Clustered Object mechanisms proposed in Tornado via a set of protocols which provide a distributed object oriented model, permitting incremental development while fully leveraging hardware-supported shared memory. Doing so, this work builds upon the lessons of previous research:

- focus on performance over flexibility
- provide an accepted software environment
- maximize concurrency, focusing on structures and algorithms rather than improved synchronization as done in earlier systems
- maximize hardware locality
- reflect workload independence using an object oriented structure
- utilize distributed data structures
- enable adaptation in order to cope with variability in parallel demands on an instance by instance basis.

References

[1] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandervoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 1–14, New York, October5–8 1997. ACM Press.

[3] J. Appavoo. Clustered objects: Initial design, implementation and evaluation. Master's thesis.

[4] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma da Silva, Orran Krieger, Marc Auslander, David Edelsohn, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[5] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert Wisniewski, Dilma da Silva, Orran Krieger, and Craig Soules. An infrastructure for multiprocessor run-time adaptation. In *WOSS - Workshop on Self-Healing Systems*, 2002.

[6] P. Austin, K. Murray, and A. Wellings. The design of an operating system for a scalable parallel computing engine. *Software, Practice and Experience*, 21(10):989–1014, [10] 1991.

[7] M. J. Bach and S. J. Buroff. Multiprocessor UNIX operating systems. *AT&T Bell Laboratories Technical Journal*, 63(8):1733–1749, October 1984.

[8] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. A distributed implementation of the shared data-object model. In Eugene Spafford, editor, *Proc. First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 1–19, Ft. Lauderdale FL (USA), 1989.

[9] Amnom Barak and Yoram Kornatzky. Design principles of operating systems for large scale multicomputers. Technical report, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, 1987.

[10] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[11] Amnon Barak and Richard Wheeler. MOSIX: An integrated multiprocessor UNIX. In USENIX Association, editor, *Proceedings of the Winter 1989 USENIX Conference: January 30–February 3, 1989, San Diego, California, USA*, pages 101–112, Berkeley, CA, USA, Winter 1989. USENIX.

[12] BBN Advanced Computers, Inc. *Overview of the Butterfly GP1000*, 1988.

[13] Bob Beck and Bob Kasten. VLSI assist in building a multiprocessor UNIX system. In USENIX Association, editor, *Summer conference proceedings, Portland 1985: June 11–14,*

1985, Portland, Oregon USA, pages 255–275, P.O. Box 7, El Cerrito 94530, CA, USA, Summer 1985. USENIX.

[14] Brian N. Bershad, Edward D. Lazowska, Henry M. Levy, and David B. Wagner. An open enviornment for building parallel programming systems. In *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 1–9. ACM Press, 1988.

[15] David L. Black, Jr. Avadis Tevanian, David B. Golub, and Michael W. Young. Locking and reference counting in the Mach kernel. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, Software, pages II–167–II–173, Boca Raton, FL, August 1991. CRC Press.

[16] Georges Brun-Cottan and Mesaac Makpangou. Adaptable replicated objects in distributed environments. Technical Report BROADCAST#TR95-100, ESPRIT Basic Research Project BROADCAST, June 1995.

[17] R. M. Bryant, H.-Y. Chang, and B. S. Rosenburg. Operating system support for parallel programming on RP3. *IBM Journal of Research and Development*, 35(5/6):617–634, September/ November 1991.

[18] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 143–156, New York, October 5–8 1997. ACM Press.

[19] M. Campbell et al. The parallelization of UNIX system V release 4.0. In *Proc. USENIX Technical Conference*, pages 307–324, 1991.

[20] 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.

[21] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of object-oriented operating system design. Technical Report UIUCDCS-R-89-1510,TTR89-14, Department of Computer Science, University of Illinois, Urbana, IL 61801, April 1989. TTR89-14.

[22] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, 2003.

[23] J. Chapin, S. A. Herrod, M. Rosenblum, and A. Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proc. of the 1995 ACM SIGMETRICS Joint Int'l Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'95/PERFORMANCE'95)*, pages 1–13, May 1995.

[24] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 12–25, December 1995.

[25] E. M. Jr. Chaves, P. C. Das, T. J. Leblanc, B. D. Marsh, and M. L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, May 1993.

[26] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proc. Fourteenth SOSP.*, pages 120–133, 1993.

[27] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Operating Systems Design and Implementation*, pages 179–193, 1994.

[28] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Paradigm: A highly scalable shared-memory multicomputer architecture. *Computer*, 24(2):33–46, February 1991.

[29] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager. Thoth, a portable real-time operating system. *Communications of the ACM*, 22(2):105–115, 1979.

[30] David R. Cheriton and Willy Zwaenepoel. The distributed v kernel and its performance for diskless workstations. In *Proceedings of the ninth ACM symposium on Operating systems principles*, pages 129–140. ACM Press, 1983.

[31] Andrew A. Chien and William J. Dally. Concurrent aggregates (CA). In *Proc. Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2nd PPOPP'90), ACM SIGPLAN Notices*, pages 187–196, March 1990. Published as Proc. Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2nd PPOPP'90), ACM SIGPLAN Notices, volume 25, number 3.

[32] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (DSA) on large-scale multiprocessors. In *Proceedings of the Symposium on Experience with Distributed and Multiprocessor Systems*, pages 227–246, San Diego, CA, USA, September 1993. USENIX Association.

[33] Christian Clémençon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (DSA) on multiprocessors. *IEEE Transactions on Software Engineering*, 22(2):132–152, February 1996.

[34] Ellis Cohen and David Jefferson. Protection in the hydra operating system. In *Proceedings of the fifth symposium on Operating systems principles*, pages 141–160, 1975.

[35] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001. IEEE Computer Society.

[36] P. Dasgupta, R. J. LeBlanc, and W. F. Appelbe. The clouds distributed operating SYstem: Functional description, implementation details and related work. In *Proc. 8th Int'l. Conf. on Distr. Computing Sys.*, page 2, 1988.

[37] Partha Dasgupta, Richard J. LeBlanc, Jr., Mustaque Ahamad, and Umakishore Ramachandran. The Clouds Distributed Operating System. *Computer*, 24(11):34–44, November 1991.

[38] Alan Dearle, Rex di Bona, James Farrow, Frans Kenskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. 7(3):289–312, 1994.

[39] Jeffrey M. Denham, Paula Long, and James A. Woodward. DEC OSF/1 version 3.0 symmetric multiprocessing implementation. *Digital Technical Journal of Digital Equipment Corporation*, 6(3):29–43, Summer 1994.

[40] Murthy Devarakonda and Arup Mukherjee. Issues in implementation of cache-affinity scheduling. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 345–358, Berkeley, CA, USA, January 1991. Usenix Association.

[41] P. Druschel. Efficient support for incremental customization of OS services. In *Proc. of the Third International Workshop on Object Orientation in Operating Systems*, pages 186–190, Asheville, NC, December 1993.

[42] Jan Edler, Jim Lipkis, and Edith Schonberg. Memory management in symunix II: A design for large-scale shared memory multiprocessors. In *UNIX and Supercomputers Workshop Proceedings*, pages 151–168, Pittsburgh, PA, September 26-27 1988. USENIX.

[43] Carla Schlatter Ellis. Extensible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 106–116. ACM Press, 1983.

[44] P. Ewens, D. R. Blythe, M. Funkenhauser, and R. C. Holt. Tunis: A distributed multiprocessor operating system. In USENIX Association, editor, *Summer conference proceedings, Portland 1985: June 11–14, 1985, Portland, Oregon USA*, pages 247–254, P.O. Box 7, El Cerrito 94530, CA, USA, Summer 1985. USENIX.

[45] 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.

[46] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. pages 87–100.

[47] B. Gamsa, O. Krieger, E. Parsons, and M. Stumm. Performace issues for multiprocessor operating systems. Unpublished, University of Toronto, 1996.

[48] Ben Gamsa. *Tornado: Maximizing Locality and Concurrency in a Shared-Memory Multiprocessor Operating System*. PhD thesis, University of Toronto, 1999.

[49] Arun Garg. Parallel STREAMS: a multi-processor implementation. In USENIX, editor, *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990, Washington, DC, USA*, pages 163–176, Berkeley, CA, USA, 1990. USENIX.

[50] Ahmed Gheith, Bodhisattwa Mukherjee, Dilma Silva, and Karsten Schwan. KTK: Kernel support for configurable objects and invocations. Technical Report GIT-CC-94-11, Georgia Institute of Technology. College of Computing.

[51] Ahmed Gheith and Karsten Schwan. Chaosarc: kernel support for multiweight objects, invocations, and atomicity in real-time multiprocessor applications. *ACM Transactions on Computer Systems (TOCS)*, 11(1):33–72, 1993.

[52] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):164–189, 1983.

[53] Kingshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th Symposium on Operating Systems Principles (SOSP-99)*, volume 34,5 of *Operating Systems Review*, pages 154–169, New York, December 1999. ACM Press.

[54] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 319–332, Berkeley, CA, October 23–25 2000. The USENIX Association.

[55] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods of the performance of parallel applications. In *Proc. 1991 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, page 120, San Diego, California, USA, May 21-24 1991. Stanford Univ.

[56] A. N. Habermann, Lawrence Flon, and Lee Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, 1976.

[57] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *USENIX Conference Proceedings*, pages 147–59, 1993.

[58] Graham Hamilton, Michael L. Powell, and James G. Mictchell. Subcontract: A flexible base for distributed programming. pages 69–79, Asheville, NC (USA), December 1993.

[59] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[60] John H. Hartman and John K. Ousterhout. Performance measurements of a multiprocessor sprite kernel. In *Proc. Summer 1990 USENIX Conf.*, pages 279–287, Anaheim, CA (USA), June 1990. USENIX.

[61] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[62] Maurice Herlihy. A methodology for implementing highly concurrent objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[63] P. Homburg, L. van Doorn, M. van Steen, A. S. Tanenbaum, and W. de Jonge. An object model for flexible distributed systems. In *First Annual ASCI Conference*, pages 69–78, Heijen, Netherlands, May 1995. http://www.cs.vu.nl/~steen/globe/publications.html.

[64] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. on Computer Sys.*, 6(1):51, February 1988.

[65] K. Hui. Design and implementation of k42's dynamic clustered object switching mechanism. Master's thesis, University of Toronto, 2001.

[66] Kevin Hui, Jonathan Appavoo, Robert Wisniewski, Marc Auslander, David Edelsohn, Ben Gamsa, Orran Krieger, Bryan Rosenburg, and Michael Stumm. Position summary: Supporting hot-swappable components for system software. In *HotOS*, 2001.

[67] Jack Inman. Implementing loosely coupled functions on tightly coupled engines. In USENIX Association, editor, *Summer conference proceedings, Portland 1985: June 11–14, 1985, Portland, Oregon USA*, pages 277–298, P.O. Box 7, El Cerrito 94530, CA, USA, Summer 1985. USENIX.

[68] M. B. Jones and R. F. Rashid. Mach and matchmaker: Kernel and language support for object-oriented distributed systems. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 21, pages 67–77, Portland, OR, November 1986. ACM, IEEE.

[69] D. R. Kaeli, L. L. Fong, R. C. Booth, K. C. Imming, and J. P. Weigel. Performance analysis on a CC-NUMA prototype. *IBM Journal of Research and Development*, 41(3):205, 1997.

[70] D. R. Kohr, Jr., X. Zhang, D. A. Reed, and M. Rahman. A performance study of an object-oriented, parallel operating system. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proceedings of the 27th Annual Hawaii International Conference on System Sciences. Volume 2 : Software Technology*, pages 76–85, Los Alamitos, CA, USA, January 1994. IEEE Computer Society Press.

[71] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II - Software, pages II–201–II–204, Boca Raton, FL, August 1993. CRC Press.

[72] 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 *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[73] R. P. LaRowe and C. Schlatter Ellis. Page placement policies for NUMA multiprocessors. *Journal of Parallel and Distributed Computing*, 11(2):112–129, [2] 1991.

[74] James Laudon and Daniel Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 241–251, New YOrk, June2–4 1997. ACM Press.

[75] T. J. Leblanc, J. M. Mellor-Crummey, N. M. Gafter, L. A. Crowl, and P. C. Dibble. The elmwood multiprocessor operating system. *Software, Practice and Experience*, 19(11):1029–1056, [11] 1989.

[76] T. J. LeBlanc, M. L. Scott, and C. M. Brown. Large-scale parallel programming: Experience with the BBN butterfly parallel processor. *Proceedings of the ACM/SIGPLAN PPEALS 1988*, pages 161–172, July 1988.

[77] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–80, March 1992.

[78] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the fifth symposium on Operating systems principles*, pages 132–140, 1975.

[79] Chu-Cheow Lim. A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions. Technical Report TR-93-063, Berkeley, CA, Oct 93.

[80] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M Shapiro. Fragmented objects for distributed abstractions. In Thoman L. Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, Los Alamitos, California, 1994.

[81] Michael Marchetti, Leonidas Kontothanassis, Ricardo Bianchini, and Michael Scott. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95*, pages 480–485, Los Alamitos, CA, USA, April 1995. IEEE Computer Society Press.

[82] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[83] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[84] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proc. Twelfth ACM Symp. on Operating Sys., Operating Systems Review*, page 191, December 1989. Published as Proc. Twelfth ACM Symp. on Operating Sys., Operating Systems Review, volume 23, number 5.

[85] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Dept. of Comp. Sc., Columbia U., New York, NY USA, April 1991.

[86] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *ACM SIGPLAN Notices*, 29(11):145–156, November 1994.

[87] Drew McCrocklin. Scaling solaris for enterprise computing. In *CUG 1995 Spring Proceedings*, pages 172–181, Denver, CO, March 1995. Cray User Group, Inc.

[88] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessor. In *USENIX Technical Conference Proceedings*, pages 295–305, San Diego, CA, Winter 1993. USENIX.

[89] Bodhisattwa C. Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: experimentation with a configurable multiprocessor thread package. In *Proceedings the 2nd International Symposium on High Performance Distributed Computing*, pages 59–66, Spokane, WA, USA, 1993. IEEE.

[90] Kevin Murray. Wisdom: The foundation of a scalable parallel operating system. Technical Report YCST-90-02, City Unviersity of York, 1990.

[91] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):134–154, 1988.

[92] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: An Experiment in Distributed Operating System Structure. *Communications of the ACM*, 23(2):92–104, February 1980.

[93] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[94] E. Parsons, B. Gamsa, O. Krieger, and M. Stumm. (de)clustering objects for multiprocessor system software. In *Fourth International Workshop on Object Orientation in Operating Systems 95*, pages 72–81, 1995.

[95] J. Kent Peacock. File system multithreading in System V Release 4 MP. In *USENIX Conference Proceedings*, pages 19–30, San Antonio, TX, Summer 1992. USENIX.

[96] J. Kent Peacock, Sunil Saxena, Dean Thomas, Fred Yang, and Wilfred Yu. Experiences from multithreading System V Release 4. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, pages 77–92. USENIX, Newport Beach, CA, March 26-27 1992.

[97] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weise. The IBM research parallel processor prototype (RP3): Introduction. In *Proc. Int. Conf. on Parallel Processing*, August 1985.

[98] R. Pike. Personal communication.

[99] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[100] David Leo Presotto. Multiprocessor streams for Plan 9. In *Proc. Summer UKUUG Conf.*, pages 11–19, London, July 1990.

[101] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. J. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Trans. on Computers*, 37 8:896–908, August 1988.

[102] Richard Rashid. From RIG to accent to mach: The evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society Fall Joint Computer Conference*, pages 1128–1137, November 1986. Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[103] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel support for the wisconsin wind tunnel. In *Proceedings of the Symposium on Microkernels and Other Kernel Architectures*, pages 73–90, San Diego, CA, USA, September 1993. USENIX Association.

[104] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the 15th Symposium on Operating Systems Principles (15th SOSP'95), Operating Systems Review*, pages 285–298, Copper Mountain, CO, December 1995. ACM SIGOPS. Published as Proceedings of the 15th Symposium on Operating Systems Principles (15th SOSP'95), Operating Systems Review, volume 29, number 5.

[105] Curt Schimmel. *Unix Systems for Modern Achitectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley Publishing Company, 1994.

[106] Wolfgang Schröder-Preikschat. Design principles of parallel operating systems —A PEACE case study—. Technical Report TR-93-020, International Computer Science Institute, Berkeley, CA, April 1993.

[107] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems (TOCS)*, 2(1):3–23, 1984.

[108] Karsten Schwan and Win Bo. Topologies: distributed objects on multicomputers. *ACM Transactions on Computer Systems (TOCS)*, 8(2):111–157, 1990.

[109] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Design rationale for psyche, a general-purpose multiprocessor operating system. In *Proc. Intern. Conf. on Parallel Processing*, page 255, St. Charles, IL, August 1988. Penn. State Univ. Press. Also published in the Univ. of Rochester 1988-89 CS and Computer Engineering Research Review.

[110] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Evolution of an operating system for large-scale shared-memory multiprocessors. Technical Report TR 309, URCSD, March 1989.

[111] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Implementation issues for the psyche multiprocessor operating system. *USENIX Workshop on Distributed and Multiprocessor Systems*, pages 227–236, October 1989.

[112] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Multi-model parallel programming in psyche. In *Proc. ACM/SIGPLAN Symp. on Principles and Practice of Parallel Programming*, page 70, Seattle, WA, March 1990. In ACM SIGPLAN Notices 25:3.

[113] *White Paper: Sequent's NUMA-Q Architecture.*

[114] Marc Shapiro, Yvon Gourbant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–337, 1989.

[115] A. Silberschatz, J. Peterson, and P. Galvin. *Operating Systems Concepts*. Addison-Wesley, Reading, MA, 1991.

[116] Dilma Silva, Karsten Schwan, and Greg Eisenhauer. CTK: Configurable object abstractions for multiprocessors. *Software Engineering*, 27(6):531–549, 2001.

[117] Burton Smith. The quest for general-purpose parallel computing, 1994.

[118] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *USENIX*, pages 141–154, San Antonio, TX, June 9-14 2003.

[119] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[120] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and Michael Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[121] J. Mark Stevenson and Daniel P. Julin. Client-server interactions in multi-server operating systems: The Mach-US approach. Technical Report CMU-CS-94-191, Carnegie-Mellon University, September 1994.

[122] J. Talbot. Turning the AIX operating system into an MP-capable OS. In *Proc. USENIX Technical Conference*, 1995.

[123] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[124] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[125] Josep Torrellas, Anoop Gupta, and John L. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174. Boston, Massachusetts, 1992.

[126] 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.

[127] M. van Steen, P. Homburg, and A. S. Tanenbaum. The architectural design of globe: A wide-area distributed sytem. Technical Report IR-442, vrige Universiteit, March 1997.

[128] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 26–40. Association for Computing Machinery SIGOPS, October 1991.

[129] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosemblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[130] Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, and S. Srbljic. The NUMAchine multiprocessor. Technical Report 324, University of Toronto, April 1995.

[131] Zvonko G. Vranesic, Michael Stumm, David M. Lewis, and Ron White. Hector: A hierarchically structured shared-memory multiprocessor. *Computer*, 24(1):72–80, January 1991.

[132] Tim Wilkinson, Tom Stiemerling, Peter Osmon, Ashley Saulsbury, and Paul Kelly. Angel: A proposed multiprocessor operating system kernel. In *European Workshop on Parallel Computing*, March 1992.

[133] 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, June 1974.

[134] W. Wulf, R. Levin, and C. Pierson. Overview of the hydra operating system development. In *Proceedings of the fifth symposium on Operating systems principles*, pages 122–131, 1975.

[135] C. Xia and J. Torrellas. Improving the performance of the data memory hierarchy for multiprocessor operating systems. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, February 1996.

[136] Yasuhiko Yokote. The apertos reflective operating system — the concept and its implementation. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 414–434, Vancouver, BC CD, [10] 1992. ACM Press , New York, NY , USA. Published as SIGPLAN Notices, volume 27, number 10.

[137] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, [11] 1987. Published as Proceedings of the 11th ACM Symposium on Operating Systems Principles, volume 21, number 5.