

# Operating System Design Principles for Scalable Shared Memory Multiprocessors

Rajat Mukherjee  
IBM T.J. Watson Research Center  
Yorktown Heights, NY 10598  
*rajatm@watson.ibm.com*

John K. Bennett  
ECE, Rice University  
Houston, Texas 77251  
*jkb@ece.rice.edu*

## Abstract

We describe SALSA, an operating system that incorporates techniques for achieving scalability in large-scale shared memory NUMA multiprocessors. We evaluate the effects of cache organization and caching policy on latency hiding via rapid thread switching. With write-back, set-associative caches, we demonstrate significant improvements in program performance (120%) with latency hiding when cache miss latency is high, even with low miss rates (1%). SALSA exploits clustering in a NUMA system, and provides primitives for hierarchical data placement and thread scheduling. We show that proper data placement can double performance in Willow, a hierarchical shared memory architecture. SALSA also provides user-control over memory allocation for fine-tuning a program's memory requirements, which is shown to improve program performance by up to 20%.

## 1 Introduction

Large-scale shared memory multiprocessors are characterized by non-uniform memory access (NUMA) behavior, memory hierarchies and large access latencies. In this paper, we present operating system techniques that can be used to achieve scalability in such systems. We describe SALSA, an operating system for Willow, a hierarchical bus-based shared memory system. SALSA incorporates these techniques and is implemented on a commercially available microprocessor (SPARC) architecture.

As memory hierarchies become deeper, and processors get faster, cache misses cost more in terms of reduced processor utilization. Rapid context switching can be used to hide miss latency [1]. SALSA provides

support for this. We examine the effect of cache design on the performance of latency hiding.

An implication of non-uniform memory hierarchies is that operating systems must exploit locality and be tunable. SALSA supports user-control of memory allocation and provides primitives for hierarchical thread placement and data allocation, allowing the user control over locality and load balancing. SALSA provides support for timeslicing and space sharing. We describe a novel technique, using lockable TLB entries, to provide private pages within an application's shared address space.

The remainder of the paper is organized as follows: In Section 2, we briefly describe the Willow architecture and the SALSA programming model. In Section 3, we present the performance results of latency hiding via rapid thread switching. We discuss issues related to thread and data locality in Sections 4 and 5, and scheduling issues in Section 6. Finally, we point to related work in Section 7 and conclude in Section 8. Further details on this work can be found in [7].

## 2 Architecture & Programming Model

### 2.1 Willow Architecture

Willow is a large-scale shared memory system with a tree-like bus hierarchy that contains two types of modules. At the processor level, the leaves of the tree are clustered processor-cache (P-C) modules. All other levels consist of memory-cache-synchronization (M-C-S) modules. Willow uses SPARC processors. Figure 1 depicts Willow with 4-processor clusters and 3 memory levels. At the most global bus, a high-speed (1 Gb/sec.) global interconnect provides direct communication between the cache and other memory-cache modules. Although all memory is shared, locality is the key to performance in Willow.

The hierarchical bus architecture reduces the potential for bus saturation as bus traffic is filtered by

---

This research was supported by an IBM Graduate Fellowship.

Global Interconnect

caches/memory modules at each level. Since physical memory is distributed, there is no traffic beyond the bus where program and data reside. Using multiple buses also increases the potential level of concurrent I/O activity in the system, as each bus could have resident disks.

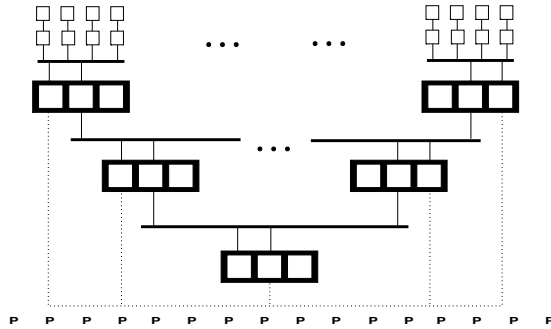


Figure 1: Willow with 3-Level Memory Hierarchy

## 2.2 SALSAs Programming Model

There are three abstractions of control recognized in SALSAs: *processes*, *tasks* and *threads*. A process has its own virtual address space, and executes on a number of processors. Each process consists of one or more kernel-level tasks, all sharing the address space. Each task may be composed of multiple user-level threads. A thread consists of a program counter, a stack pointer, and pointers to state information. Tasks and threads belonging to a process communicate via shared memory.

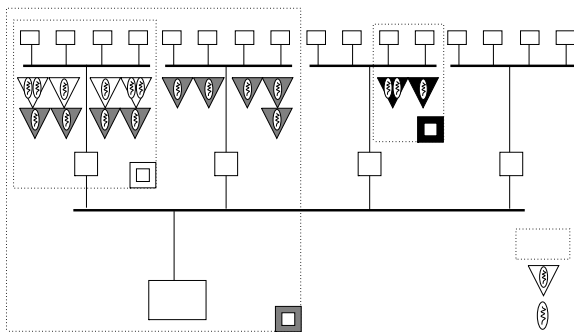


Figure 2: SALSAs Programming Model

Figure 2 depicts 3 application processes running on a hierarchical shared memory multiprocessor. Each consists of a set of tasks (similarly shaded) executing on a set of processors. Each processor maintains a list of tasks, possibly multithreaded, that have been

scheduled on it. While tasks are scheduled by the kernel, threads within a single task are scheduled by a user-level scheduler.

The SALSAs threads package is based on Presto, a lightweight user-level programming environment developed for shared memory systems [3]. While retaining some of the high level synchronization constructs of Presto, we modified the data structures and most of the thread primitives. SALSAs also differs from Presto in the use of register windows [4] for context caching, and support for hierarchical thread pools and pool affinity.

## 2.3 Shared and Private Data

Although program tasks share an address space, it is frequently necessary for each task to maintain private (task/processor-specific) data. Although possible to index global arrays via task identifiers, it is simpler and more efficient to provide a mechanism by which the same virtual address can be used by different tasks to access private data, thus eliminating indexing overhead. Our requirement for private variables stems from the necessity to maintain information about processor state in the run-time threads package. SALSAs's implementation of private pages reduces thread switch time by 4-5 cycles (about 15%).

In SALSAs, tasks in a process share the address space, but *locked TLB entries* provide independent mappings for private addresses. Figure 3 shows the virtual-to-physical mappings for a process on a 4-processor cluster with one private page. Shared addresses map to the same physical addresses, irrespective of requesting processor. Private pages, on the other hand, map to different physical pages on different processors via locked TLB entries. The SPARC chip supports TLB locking, each locked private page accommodating 4096 bytes. Since private variables are not common, a few locked TLB entries suffice. This approach avoids using multiple copies of the page tables, providing significant memory savings ( $\approx .25$  MB for a 16-processor program using 16 MB of address space), while adding only a few cycles to task switching.

## 2.4 Simulation Environment

Our simulation environment used MPSAS, a detailed instruction-level simulator that was developed at Sun Microsystems, Inc. and modified for Willow. The SALSAs kernel, run-time system and programs were compiled using the standard Sun compiler (which issued unnecessary save and restore instructions for

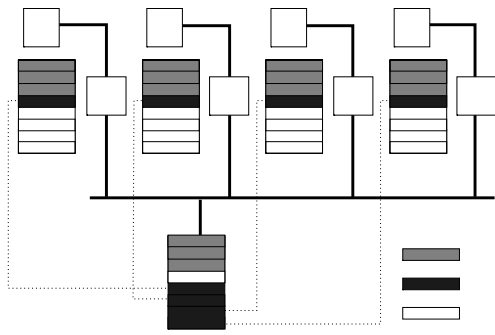


Figure 3: Memory Mapping for Private Data

procedure call (see Section 3.1)). The executable image was loaded into the memory of the simulated system. The kernel, run-time system, and program are all simulated in detail, including register, cache, and memory transfers as well as virtual memory translations, thus accurately accounting for resource contention and delays.

We used a mix of symbolic and integer programs. For our work, we consider the program characteristics to be more important than the algorithms used. The programs using SALSA threads are **LIFE** (Game of Life), **MULT**(Matrix Multiply), **ISO** (Subgraph Isomorphism), **TSP** (Traveling Salesman), **SOR** (Successive Over-Relaxation), **SIEVE** (Finding Primes), **STRING** (Pattern Matching), **MST** (Minimum Spanning Tree), **QSORT** (Quicksort) and **GAUSS** (Gaussian Elimination). Latency hiding is effective only for programs using threads. The programs using SALSA kernel tasks for parallelism are **FFT** (Fast Fourier Transform) and **MERGE** (Merge-sort).

### 3 Hiding Memory Latency

Multiple levels of cache memory are used to reduce average access latency in large shared memory systems. However, large cache miss latency, as in DASH(>135 cycles) and KSR-1(570 cycles), causes significant drops in processor utilization.

Memory access latency can be masked via rapid context switching [1]. When a thread misses in the cache, it is possible to switch to an alternate thread while the access is being satisfied, allowing higher utilization. SALSA provides hardware and software traps that can be invoked to switch between user-level threads belonging to an application. In this section, we present an overview of the latency hiding technique used and the performance improvements possible.

### 3.1 Fast Context Switching Using Register Windows

SALSA<sup>TRAP</sup> uses register windows [4] to cache multiple thread contexts, allowing a context switch to a cached thread to be accomplished in 30 processor cycles. The use of register windows for context caching, as opposed to procedure call, has been previously proposed for Alewife by Agarwal et al. [1]. All switches in Alewife are triggered by hardware traps. SALSA supports switches initiated by both hardware and software, requires no hardware modifications to the SPARC processor architecture, and is thus usable with commercially available processors.

Thread contexts are cached in the processor, using a window per context. Figure 4 depicts a register file of 8 register windows divided into four banks, each with its own trap window. Up to four threads (1 active) can be *resident* on the processor. A program that sleeps on a synchronization event does so by means of an explicit call to the run-time system. A high latency access switch is triggered by a hardware trap (30 cycles), enabled by *save* and *restore* instructions, traditionally used for procedure call [4].

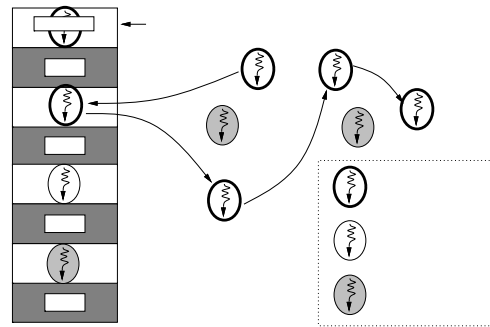


Figure 4: Context Cache Using Register Windows

### 3.2 The Interaction of Latency Hiding with Cache Architecture

We simulated a set of programs on a uniprocessor architecture with a 64 KB cache that generates hardware traps on cache misses. In order to emulate large-scale systems, we varied the bus latencies from 20-150 cycles. Latency traps were taken only on read misses to program data. Other techniques, such as write-buffering, are able to hide write latency [6].

Figure 5 shows the percentage degradation in program execution time with latency hiding enabled (traps vs. no traps) for a miss latency of 20 cycles with direct-mapped, 2-way and 4-way associative caches.

The degradation in the case of direct-mapped caches is about 1% for MST and QSORT, but much larger for ISO (9%) and LIFE (300%). With associative caches, the degradation is significantly smaller (0.1% (ISO) and 3% (LIFE) for 4-way associativity).

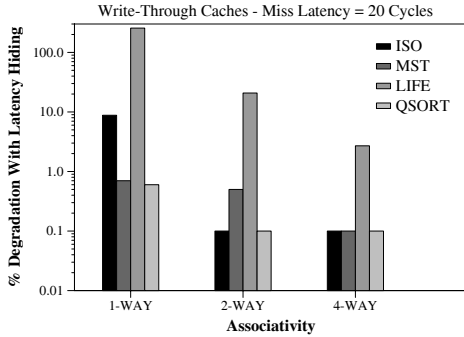


Figure 5: Degradation With Latency Hiding

Bad data or code mappings that exist in the program can be exposed from changes in thread ordering due to context switching, causing severe victimization in a direct-mapped cache. Cache contention between the application and the latency trap code can result in poor cache hit-ratios, increasing effective switch time in direct-mapped caches.

While performance improves with set-associativity, latency hiding did not present speedups with write-through caches. Figure 6 depicts the overall improvements due to latency traps with 4-way associative, write-through caches for 5 programs for a range of miss latencies. The overheads are due to the filling of write-buffers due to increased utilization, causing the processor to stall even on write hits. This increases average switch time, reducing the fraction of the latency that is hidden. Figure 7 shows the improvements due to latency hiding with write-back caches. With write-back, 4-way set-associative caches, the programs showed improvements from 3 to 125% for a miss latency of 150 cycles.

Table 1 shows the total miss rates, user-level read miss rates (with traps enabled) and improvements for the programs for a 150-cycle miss latency. Figure 8 depicts program instruction and data miss rates with and without context switching. MULT had the highest overall miss rate of all 5 programs, and the majority of misses in MULT were read misses that resulted in latency traps; the other programs had many write misses that were ignored. MULT also exhibits a large decrease in miss rate due to positive interference.

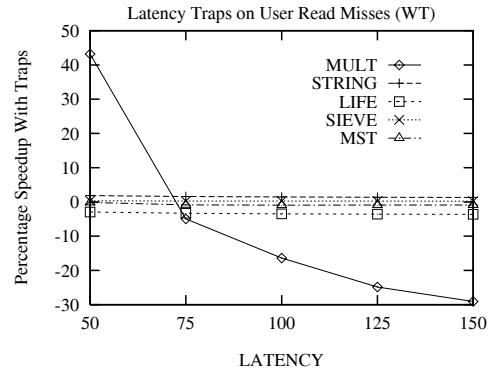


Figure 6: Performance with Write-Through Caches

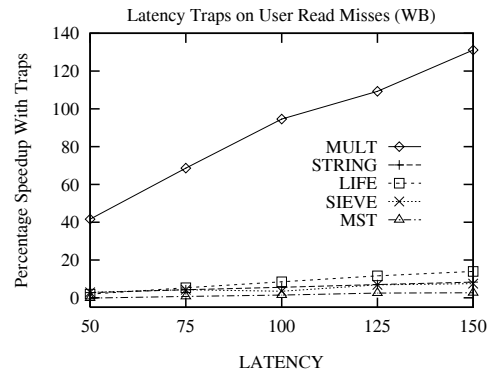


Figure 7: Performance with Write-Back Caches

Prog.	Total Miss Rate		Read Miss Rate(Trap)	% Faster
	No Traps	Traps		
STRING	0.15	0.10	0.012	8.32
MST	0.16	0.18	0.099	2.6
SIEVE	0.86	0.77	0.032	7.47
LIFE	0.41	0.40	0.179	13.97
MULT	1.43	0.65	0.602	131.13

Table 1: Miss Rates & Performance (Latency 150 cyc.)

### 3.3 Multiprocessor Issues

Since the performance improvements from latency hiding accrue due to increased utilization at the individual processors, the uniprocessor results are pertinent to the issue of scalability in a multiprocessor environment, where coherence-based invalidations are likely to increase misses. However, the capacity of the network to support the higher processor utilization determines multiprocessor latency hiding performance.

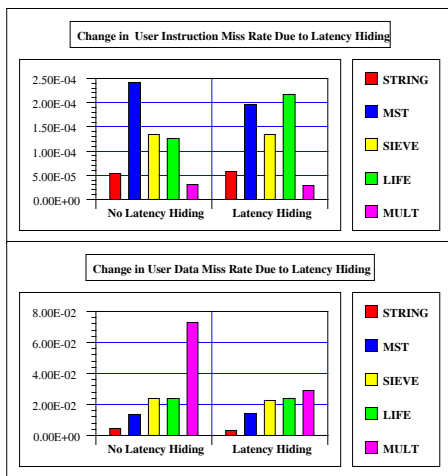


Figure 8: Change in User Cache Miss Rates

We studied the performance on a single-bus, 4-processor cluster, varying miss penalties from 100 to 200 cycles to emulate high network latency. The improvements were similar to the uniprocessor case.

An implicit assumption in attempting to hide latencies and increasing processor utilization is that the network can handle the increased bandwidth requirements. With latency hiding enabled, we observed 17-100% larger read wait times and 33-125% larger write wait times due to increased bus contention, demonstrating the importance of high-bandwidth networks to the success of this technique. Increased processor utilization with latency hiding will therefore limit the number of processors that can be placed in a cluster of a shared memory multiprocessor.

## 4 Hierarchical Thread Allocation

In SALSA, it is necessary to create at least one thread queue per processor, for two reasons. First, as the system scales, there will be contention for a single global ready queue. Second, since blocked threads may be resident, with state saved in a register window, they must be restarted on processors on which they are resident.

SALSA provides more than per-processor thread pools; a thread pool exists at each level in the hierarchy. Threads can have affinity for a cluster/sub-tree in the hierarchy, as opposed to a single processor, allowing the user to control thread locality. Idle processors in the sub-tree can find available work from the pool at the root of that sub-tree. Load balancing across clusters can be achieved by scheduling equal numbers

of threads in all cluster pools. Processor affinity can also be forced if required.

This approach can exploit the hierarchical memory organization efficiently. In Willow, there is no bus-traffic on buses outside a sub-tree that processes a program. Figure 9 shows the organization of the ready queues on a 3-level bus hierarchy. Processors can only access the global ready queue and the queues of sub-trees that they belong to. If any resumed thread is resident on a processor, it is restarted on that processor's queue. If not resident, it is placed in the pool for which it has affinity. If no affinity exists, threads are placed in the global pool. Since thread scheduling is handled at user-level, it is possible to use different scheduling algorithms and structure the queues to take advantage of the underlying architecture. On systems like Dash and Alewife, memory is either local or remote, and three thread pool levels (per-processor, cluster and global) would be the logical choice.

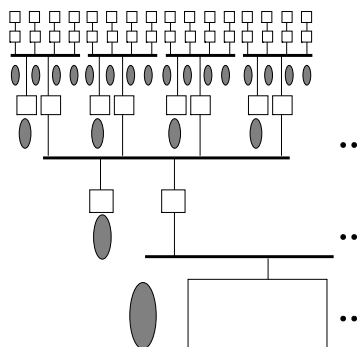


Figure 9: Hierarchical Ready Queue Organization

## 5 Hierarchical Memory Allocation

### 5.1 User Control of Memory Allocation

The SALSA allocator maintains separate free-lists for blocks of different sizes. On a request for a block of certain size, the allocator examines the appropriate list for a free block. If found, a system call can be avoided. If no free block is found, a system call is made to allocate more memory. There is no coalescing of blocks because each list contains blocks of one size.

When the application requests a small block, a larger block is requested from the kernel, divided into smaller chunks of the required size, and appended to the free-list. When a large block is requested, only a single block is requested from the kernel. We observed that large allocation requests usually occur once at

initialization for the program data set and most allocation requests are for small blocks.

The size of the block requested from the kernel must be chosen carefully to minimize the number of system calls without drastically increasing the overhead of creating too many blocks. SALSAs allows the user to specify the *minimum block size* (MINBLOCKSIZE) to request from the kernel in the case of small block requests. SALSAs provides instrumentation that can be used to profile a program’s allocation requirements for tuning the choice of MINBLOCKSIZE, which can be set to different values at different program stages.

We ran the programs with MINBLOCKSIZE equal to 128, 256, 512, 1024 and 2048 bytes. In all programs, performance improves with the increase in the minimum block size until a certain point. This correlates with the fact that small block requests are common and prefetching small blocks avoids system call overhead. The actual improvement is highly application dependent, ranging from 0.01% (TSP) to as high as 21% (SOR).

## 5.2 Hierarchical Memory Allocation

Most NUMA systems either use intermediate levels of memory as cache, or use extant operating systems that do not support hierarchical allocation, e.g., KSR-1, the Data Diffusion Machine, Wisconsin Multicube. SALSAs provides the user explicit control of data placement based on level in the memory hierarchy.

At each level in the hierarchy, there can be multiple memory modules. The number of modules at a level changes with the level, as in Figures 1 and 9, which depict a 3-level hierarchical architecture (64 processors). Memory at Level 3 is most global, and in the absence of the global interconnect, the only memory shared by all processors.

For hierarchical allocation, the user specifies the level and the size of the block desired. The index of the memory module at the given level is determined automatically, based on the requesting processor. The level and index are saved in the block header for identification during deallocation; memory allocated from a free-list at one level cannot be deallocated at another level or index.

In order to demonstrate the performance benefits of hierarchical allocation, we executed 4 programs - MERGE, LIFE, FFT and GAUSS - using two different allocation strategies (local memory vs. global memory) on a 3-tier Willow hierarchy. These are the two extreme placement alternatives and demonstrate the performance improvements possible with improved data locality. Each program had a data set size larger

than the size of the second-level cache (64 KB) so that the results would not be skewed by whole data-set caching. Figure 10 shows the total execution times and the second-level cache hit ratios.

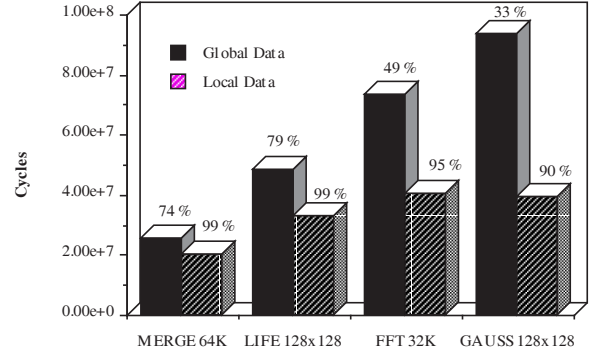


Figure 10: Effect of Data Locality

Local allocation resulted in overall performance improvements of between 20% (MERGE) and 58% (GAUSS). By allocating data in local memory, the number of accesses (and cache misses) to the cache at the same level can be reduced significantly (the FFT program saw a 99% decrease in the total number of second-level cache accesses, i.e., most second-level cache accesses are eliminated). Global allocation pays for increased second-level cache misses and the higher latency of these misses. Hierarchical allocation thus provides significant performance benefits.

## 5.3 Hierarchical Lock Allocation

The performance of a shared memory system is heavily dependent on the efficiency of its synchronization primitives. SALSAs provides synchronization constructs such as locks, barriers, monitors and condition variables. The processor is relinquished upon failed synchronization to prevent deadlock when context caching is used to hide latency [7].

In order to provide hardware synchronization support, e.g., reserved cache lines for locks or locks with interrupt capability, SALSAs supports locality in lock allocation. Hierarchical lock allocation can also be used to improve the performance of distributed synchronization schemes such as tree barriers, tournament barriers, etc., which are superior to centralized barriers.

SALSAs provides system calls for lock allocation distinct from the calls for memory allocation. Lock memory is allocated in cache lines to minimize cache interference and false sharing. Lock allocation primitives, like the hierarchical memory allocation primitives, use level specification for locality.

## 6 Processor Allocation

SALSA uses the clustering factor to allocate processors; processors are allocated from a single cluster unless there are no available clusters with the requested number of processors. This improves locality and potentially reduces bus traffic by containing communication within sub-trees.

The goal in SALSA is to be able to support various scheduling policies. SALSA supports both *timeslicing* and *space sharing* [8]. With timeslicing, in addition to the task switching overhead, performance can degrade for two reasons. First, applications may busy-wait for locks held by preempted tasks; thus, it is imperative that preemption occurs at safe points, i.e., outside critical sections. Second, switching address spaces affects cache behavior; even if a task is rescheduled on the same processor, intervening processes may have replaced useful cache data.

One approach to solving these problems is to partition the available processors among concurrently executing jobs (space sharing). With space sharing, a processor is allocated only to one process unless mandatory, i.e., a new process has no processors. Allocated processors may be reallocated to other applications on demand. This can be accomplished by the kernel vectoring events to the user-level thread schedulers, which manage the threads and can decide upon safe points to relinquish processors for reallocation. This approach is used in *process control* [8] and *scheduler activations* [2]. This support is in-built in SALSA's work-queue model, where threads are constantly taken off a ready queue by a scheduler.

To compare the performance of space sharing and timeslicing, we ran 3 sets of programs (2 per set) on four processors, first with both programs on all processors (timesliced) and then with the two programs on separate sets of two processors each (static approach to space sharing). The programs were (ISO and GAUSS), (TSP and MST) and (SOR and LIFE).

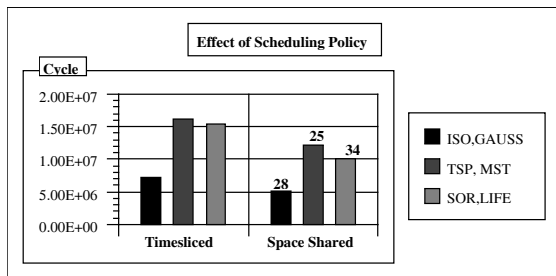


Figure 11: Timeslicing Vs. Space Sharing

Figure 11 shows the total completion time for both applications (in cycles) in the two cases. We observed that space sharing had better performance than timeslicing (25-34%) for all 3 program sets. Table 2 shows the average cache hit ratio (of the 4 processor caches) in both schemes. Note that we did not use gang scheduling with timeslicing. Without gang scheduling, tasks can easily lose synchronization in cases where there are phases of initialization that involve a single processor.

Application Group	Average Cache Hit Ratio	
	TimeSlicing	SpaceSharing
ISO, GAUSS	98.84 %	99.20 %
TSP, MST	98.95 %	99.85 %
SOR, LIFE	99.06 %	99.38 %

Table 2: Effect of Timeslicing on Hit Ratio

## 7 Related Work

Several techniques for achieving scalability in shared memory multiprocessors have been proposed. Schemes such as adaptive caching, where the coherence policy adopted is based on expected data usage patterns, and prefetching [6], where data and instructions are fetched from memory before they are used, can be used to improve cache hit rates, but cannot eliminate cache misses entirely. To further improve performance when large latencies cannot be avoided, it is essential to be able to hide these latencies.

Previous proposals for fast context switching [1] mandate hardware modifications to existing processor architectures, which precludes the use of commercially available processors. SALSA's software-based technique can be implemented on the SPARC architecture without hardware modifications. Traps need to be disabled when in supervisor mode because additional synchronous traps when executing in supervisor mode cause the processor to reset. The overhead for this is ignored in the switch times reported for April [1]. SPARC Version 9 allows stacked traps, which would reduce SALSA thread switch overhead by 3-4 cycles.

Relaxed memory consistency models allow the overlap of memory accesses with other computation and accesses [5]. The performance of weak consistency models can be enhanced by dynamic scheduling, which can hide a portion of read latency. Context switching can be used in conjunction with these schemes.

The assumptions used in some previous studies of latency hiding are not realistic, and have the potential of skewing results. For example, the small caches used in [6] inflate miss rates and reduce inter-miss distances. We do not make any simplifying assumptions

about context switch times, cache sizes, etc., but use complete context switch code, and realistic cache sizes that are based on commercially available caches.

Sharing between address spaces has been described in Multics, UNIX (*fork*) and SunOS. Mach provides sharing via mapping portions of different address spaces to the same physical pages. However, the provision of private portions within shared address spaces via TLB locking has not been presented before.

## 8 Conclusions

In this paper, we have evaluated operating system design techniques that can be used to provide scalability in large-scale shared memory multiprocessors with non-uniform memory access behavior. These include:

- ▷ Simple programming model with user threads
- ▷ Masking memory latency via thread switching
- ▷ User control of memory allocation
- ▷ Support for hierarchical thread pools
- ▷ Support for hierarchical data placement
- ▷ Support for different scheduling policies

We address the problem of latency in large-scale shared memory multiprocessors by switching thread contexts on cache misses. The novelty of our approach is the combination of software and hardware synchronization models that makes it applicable across different programming environments. We investigated the interaction of latency hiding with cache design. We achieved significant improvements with write-back, set-associative caches (120% with MULT for a miss latency of 150 cycles with a read miss rate of about 1%). We used realistic cache sizes (64 KB), and made no assumptions about switch overheads or thread interactions. We showed that the success of latency hiding via context switching is dependent on the ability of the communication network to support the increased processor utilization. Average cluster bus access times of 17% to 125% worse than uniprocessor averages motivate the use of small clusters.

We have demonstrated that data and thread locality are crucial to performance in systems with non-uniform memory hierarchies. Proper placement of data can double performance in a three-level memory hierarchy. SALSA takes advantage of the clustering in the system, and provides primitives for explicit data and thread placement. Previous systems with lightweight threads, e.g., Presto, Amber, do not provide hierarchical thread pools.

We presented novel techniques for providing user-control over memory allocation and TLB locking to provide private pages in a shared address space.

SALSA supports scheduling strategies such as space sharing, as significant performance improvements are possible (25-35%). The techniques described here are required for achieving shared memory scalability.

## References

- [1] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104 – 114, May 1990.
- [2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 95 – 109, October 1991.
- [3] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. Technical Report 87-09-01, University of Washington, Department of Computer Science, September 1987.
- [4] R. B. Garner. SPARC Scalable Processor Architecture. *SunTechnology*, pages 42 – 63, 1988.
- [5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15 – 26, May 1990.
- [6] A. Gupta, J. Hennessy, K. Gharachorloo, T. C. Mowry, and W.-D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43 – 63, May 1991.
- [7] R. Mukherjee. *The Interaction of Architecture and Operating System in the Design of a Scalable Shared Memory Multiprocessor*. PhD thesis, Rice University, Houston, Texas, November 1993.
- [8] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159 – 166, December 1989.