

Toward Large-Scale Shared Memory Multiprocessing

John K. Bennett
John B. Carter
Willy Zwaenepoel

Computer Systems Laboratory
Rice University

Abstract

We are currently investigating two different approaches to scalable shared memory: Munin, a distributed shared memory (DSM) system implemented entirely in software, and Willow, a true shared memory multiprocessor with extensive hardware support for scalability. Munin allows parallel programs written for shared memory multiprocessors to be executed efficiently on distributed memory multiprocessors. Unlike existing DSM systems, which only provide a single mechanism for memory consistency, Munin provides multiple consistency protocols, matching protocol to data object based on the expected pattern of accesses to that object. We call this approach *type-specific coherence*. Munin also employs a relaxed consistency model to mask network latency and to minimize the number of messages required for keeping memory consistent. Willow is intended to be a true shared memory multiprocessor, providing memory capacity and performance capable of supporting over a thousand commercial microprocessors. These processors are arranged in cluster fashion, with a multi-level cache, I/O, synchronization, and memory hierarchy. Willow is distinguished from other shared memory multiprocessors by a layered memory organization that significantly reduces the impact of *inclusion* on the cache hierarchy and that exploits locality gradients. Willow also provides support for *adaptive cache coherence*, an approach similar to Munin's type-specific coherence, whereby the consistency protocol used to manage each cache line is selected based on the expected or observed access behavior for the data stored in that line. Implementation of Munin is in progress; we are still designing Willow.

MUNIN

Introduction

Shared memory programs are generally easier to develop than distributed memory (message passing) programs, because the programmer does not

This work was supported in part by the National Science Foundation under Grants CDA-8619893, CCR-8716914, and CCR-9010351 and by NASA and NSF Graduate Fellowships.

SCALABLE SHARED MEMORY MULTIPROCESSORS

need to explicitly initiate data motion. Distributed memory machines, however, scale better in terms of the number of processors that can be supported. Hence, our goal is to provide the best of both worlds: the relative ease of programming of the shared memory model and the scalability of a distributed memory machine. Our performance goal is to provide execution efficiency nearly equal to that of hand-crafted distributed memory (message-passing) code for the same application.

Munin is a distributed shared memory (DSM) system that allows parallel programs written for shared memory multiprocessors to be executed efficiently on distributed memory multiprocessors. Munin's user interface is a consistent global address space, with thread and synchronization facilities like those found in shared memory parallel programming systems. Munin provides this interface via a collection of runtime library routines that use the system's underlying message passing facilities for interprocessor communication and that use kernel support for page fault handling. Munin is unique among existing DSM systems [Cha89, Li86, Ram88] both in that it provides *type-specific coherence* and in that it uses a relaxed model of consistency for shared memory.

Munin provides a suite of consistency protocols so that individual data objects are kept consistent by a protocol tailored to the way in which that object is accessed. Several studies of shared memory parallel programs have indicated that no single consistency protocol is best suited for all parallel programs [Ben90a, Egg88, Egg89]. Furthermore, within a single program, different shared data objects often are accessed in fundamentally different ways [Ben90a], and a particular object's access pattern can change during the execution of a program. Existing DSM systems have not taken advantage of these observations, and have limited their functionality to providing a single consistency protocol for all programs and all objects. Munin uses program annotations, currently provided by the programmer, to choose a consistency protocol suited to the expected access pattern of each object, or to change protocols during execution.

DSM systems such as Ivy, Clouds, and Amber [Li86, Ram88, Cha89] are based on *sequential consistency* [Lam79]. Sequential consistency requires that each modification to a shared object become visible imme-

TOWARD LARGE-SCALE SHARED MEMORY MULTIPROCESSING

diately to the other processors in the system. In a DSM system, this means that the system is required to transmit at least one message to invalidate all remote copies of an object and then must stall until these messages are acknowledged. Recently, several researchers in the shared memory multiprocessor community have advocated the use of relaxed consistency models that force the programmer to make synchronization events in the program visible to the memory consistency mechanism. By requiring this visibility, the memory hardware is able to buffer writes between synchronization points, thus reducing the latency of processor stalls [Dub86, Sch87, Gha90]. Munin achieves similar advantage by using a software *delayed update queue* to buffer pending outgoing write operations. Use of the delayed update queue allows the runtime system to merge modifications to the same object transparently, which greatly reduces the number of messages required to maintain consistency and causes Munin programs to combine data motion and synchronization as is done in hand-coded message passing programs. All objects use the same delayed update queue, so updates to different objects destined for the same remote processor can be combined as well. The delayed update queue is flushed whenever the consistency semantics in force require strict ordering of write operations, e.g., when a local thread releases a lock or arrives at a barrier.

Munin differs from recent scalable shared memory multiprocessors [Aga90, Len90] that have used relaxed consistency models to minimize processor stalls during writes to shared memory. These multiprocessors can suffer from frequent read misses, which significantly affect performance. Munin's use of type-specific coherence addresses this problem of excessive read misses. We have found that an update-based consistency protocol is superior to an invalidation-based protocol if the invalidate protocol will result in an excessive number of reloads after invalidation (reloads are read misses). By taking advantage of semantic hints, Munin can frequently load shared data before it is accessed, for example, by moving the data protected by a lock when lock ownership is transferred. This allows us to reduce or hide the latency associated with the corresponding read misses.

In addition to the delayed update mechanism, Munin employs several other well known consistency mechanisms as part of its multiple protocol

SCALABLE SHARED MEMORY MULTIPROCESSORS

approach to type specific coherence [Ben90b]. These mechanisms include replication, migration, invalidation, and remote load/store. We use each of these mechanisms for the particular types of shared data objects for which they are most appropriate.

We have based many of our design decisions on the results of a study of sharing and synchronization behavior in a variety of shared memory parallel programs, in which we observed that a large percentage of shared data accesses fall into a relatively small number of access type categories that can be supported efficiently [Ben90a].

Our approach to the design of Munin can therefore be summarized as follows:

1. Understand sharing and synchronization in shared memory parallel programs.
2. Exploit this understanding by developing efficient consistency mechanisms for the observed sharing behavior.
3. Match mechanism to type using user-provided information.

The remainder of this section describes our progress toward achieving these objectives.

Sharing in Parallel Programs

Type-specific coherence requires that there be a relatively small number of identifiable shared memory access patterns that characterize the majority of shared data objects, and for which corresponding consistency mechanisms can be developed. In order to test our approach, we studied a number of shared memory parallel programs written in C++ [Str87] using the Presto programming system [Ber88] on the Sequent Symmetry shared memory multiprocessor [Lov88]. The selected programs are written specifically for a shared memory multiprocessor so that our results are not influenced by the program being written with distribution in mind and accurately reflect the memory access behavior that occurs when programmers do not expend special effort towards distributing the data across processors. Presto programs are divided into an initialization

TOWARD LARGE-SCALE SHARED MEMORY MULTIPROCESSING

phase, during which the program is single-threaded, and a computation phase.

Six programs studied in detail were: Matrix multiply, Gaussian elimination, Fast Fourier Transform (FFT), Quicksort, Traveling salesman problem (TSP), and Life. Matrix multiply, Gaussian elimination, and Fast Fourier Transform are numeric problems that distribute the data to separate threads and access shared memory in predictable patterns. Quicksort uses divide-and-conquer to dynamically subdivide the problem. Traveling salesman uses central work queues protected by locks to control access to problem data. Life is a “nearest-neighbors” problem in which data is shared only by neighboring processes. Other programs studied include parallel versions of minimum spanning tree, factorial, SOR, shellsort, prime sieve, and a string matching algorithm.

Methodology

We collect logging information for a program by modifying the source and the run-time system to record all accesses to shared memory (13 microseconds to record each access). These program modifications are currently done by hand. A call to a logging object is added to the program source after every statement that accesses shared memory. The Presto run-time system was modified so that thread creations and destructions are recorded, as are all synchronization events. The end of each program’s initialization phase is logged as a special event so that our analysis tool can differentiate between the initialization and the computation phase.

A program executed with logging enabled generates a series of log files, one per processor. Each log entry contains an *Object ID*, a *Thread ID*, the *Type of Access*, and the *Time of Access*. Examples of *Type of Access* include creation, read, write, and lock and monitor accesses of various types. *Time of Access* is the absolute time of the access, read from a hardware microsecond clock, so the per-processor logs can be merged to form a single global log.

We can specify the granularity with which to log accesses to objects. The two supported granularities are *object* and *element*. At object granularity, an access to any part of an object is logged as an access to the

SCALABLE SHARED MEMORY MULTIPROCESSORS

entire object. At element granularity, an access to a part of an object is logged as an access to that specific part of the object. For example, the log entry for a read of an element of a matrix object indicates only that the matrix was read at object granularity, but indicates the specific element that was read at element granularity.

Our study of sharing in parallel programs distinguishes itself from similar work [Egg88, Sit88, Web89] in that it studies sharing at the programming language level, and hence is relatively architecture-independent, and in that our selection of parallel programs embodies a wider variation in programming and synchronization styles.

An important difference between our approach and previous methods [Sit88, So86] is that we only log accesses to shared memory, not all accesses to memory. Non-shared memory, such as program code and local variables, generally does not require special handling in a distributed shared memory system. A useful side effect of logging only accesses to shared memory is that the log files are much more compact. This allows us to log the shared memory accesses of relatively long-running programs in their entirety, which is important because the access patterns during initialization are significantly different from those during computation.

Logging in software during program execution combines many of the benefits of software simulation [So86] and built-in tracing mechanisms [Sit88], without some of the problems associated with these techniques. As with software simulation, with software logging it is easy to change the information that is collected during a particular run of the program. For example, if only the accesses to a particular object are of interest, such as the accesses to the lock protecting a central queue, only the logging associated with that object need be enabled. On the other hand, software-based logging does not slow down program execution to the extent that software simulation of the program and architecture does. Unlike with address tracing techniques, it is possible to collect higher-order information about particular accesses. For example, we can log an attempt to acquire a monitor, successful acquisition of the monitor, or sleeping on a monitor condition variable. This information is not easily recreated from a standard address trace.

The flexibility, power and low overhead of our system does not come without cost. Only accesses to shared memory performed by the appli-

cations program and run-time system are collected, so our system suffers from what Agarwal refers to as *omission distortion* [Sit88], the inability of a system to record the complete address stream of a running program. The omission distortion is not significant in this case because we are not trying to determine how any particular cache consistency mechanism will perform, but rather are attempting to characterize patterns of sharing that are common in parallel applications programs. Also, because only accesses to shared memory are collected, our logs may experience *temporal distortion* in the sense that periods with frequent accesses to shared memory will be slowed down to a greater extent than periods when accesses to shared memory are infrequent. Since the temporal distortion in this case is limited by synchronization events, which constrain the relative ordering of events, temporal distortion is also not a serious problem.

Categories of Sharing: Intuitive Definitions

The results of our study support our approach, in that we have identified a limited number of shared data object types: *Write-once*, *Write-many*, *Producer-Consumer*, *Private*, *Migratory*, *Result*, *Read-mostly*, and *Synchronization*. We classify all shared data objects that do not fall into one of these categories as *General Read-Write*.

Write-once objects are read-only after initialization. *Write-many* objects frequently are modified by several threads between synchronization points. For example, in Quicksort, multiple threads concurrently modify independent portions of the array being sorted. *Producer-Consumer* objects are written (produced) by one thread and read (consumed) by a fixed set of other threads. *Private* objects, though declared to be shared data objects, are only accessed by a single thread. Many parallel scientific programs exhibit “nearest neighbors” or “wavefront” communication whereby the only communication is the exchange of boundary elements between threads working on adjacent sub-arrays. The boundary elements are *Producer-Consumer* and the interior elements are *Private*. *Migratory* objects are accessed in phases, where each phase corresponds to a series of accesses by a single thread. Shared objects protected by locks often exhibit this property. *Result* objects collect results. Once written, they are only read by a single thread that uses the results. *Read-*

SCALABLE SHARED MEMORY MULTIPROCESSORS

mostly objects are read significantly more often than they are written. *Synchronization* objects, such as locks and monitors, are used by programmers to force explicit inter-thread synchronization points. Synchronization events include attempting to acquire a lock, acquiring a lock, and releasing a lock. The remaining objects, which we cannot characterize by any of the preceding classes, are called *General Read-Write*. The categories define a hierarchy of types of shared data objects. When we identify an object's sharing category, we use the most specific category possible under the following order (from most specific to least specific): *Synchronization*, *Private*, *Write-once*, *Result*, *Producer-Consumer*, *Migratory*, *Write-many*, *Read-mostly*, and *General Read-Write*.

Results of Sharing Study

The general results of our analysis can be summarized as follows:

General Results

- There are very few *General Read-Write* objects. Coherence mechanisms exist that can support the other categories of shared data objects efficiently, so a cache consistency protocol that adapts to the expected or observed behavior of each shared object will outperform one that does not.
- The conventional notion of an object, as viewed by the programmer, often does not correspond to the appropriate granularity of data decomposition for parallelism. Often it is appropriate to maintain consistency at the object level, but sometimes it is more appropriate to maintain consistency at a level smaller or larger than an object. Thus, a cache consistency protocol that adapts to the appropriate granularity of data decomposition will outperform one that does not.
- Type-specific coherence significantly reduces the amount of bus traffic required to maintain consistency. This improvement is caused by the fact that many programs perform a large number of writes to shared data objects between synchronization points, so many updates to the same data object are combined before they are eventually propagated. Additionally, type-specific coherence requires

the same amount of bandwidth as write-invalidate, but when the cache line size is small, it does so with fewer messages (thus, the average message is proportionally larger).

Specific Results

- With object-level logging, *Write-many* accesses dominate other forms of shared data access. The other sharing category into which a large portion of the accesses fall at object-level granularity is *Write-once*.
- Parallel programs in which the granularity of sharing is fine tend to have their underlying fine grained behavior masked when the logging is performed on a per-object basis. The access behavior of many programs are considerably different when examined per element. For example, in the Life program, when examined on a per-object basis, virtually all shared accesses are *Write-many*. However, when examined by element, 82 percent of the shared data is in fact *Private* (the interior elements of the board) and 17 percent is *Producer-Consumer* (the edge elements).
- The average number of different objects accessed between synchronization points indicates the average number of delayed updates that will be queued up at a time. If this number is small, as the data indicate, managing a queue of delayed updates (thus providing a relaxed model of memory consistency) does not require significant overhead.
- *Write-many* objects are written about one-half as many times as they are read. Large numbers of accesses occur between synchronization points. We call a series of accesses to a single object by any thread between two synchronization points in a particular thread a “no-synch run”. The large size of the no-synch runs indicate that delayed update offers substantial performance improvement. No-synch runs differ from Eggers’s “write-runs” [Egg88] in that they do not end when a remote thread accesses the object, but rather whenever a thread synchronizes. Intuitively, write-runs end when a standard consistency mechanism, such as a write-invalidate or

SCALABLE SHARED MEMORY MULTIPROCESSORS

write-update scheme that enforces sequential consistency, would enforce consistency. No-synch runs end when the programmer requires consistency.

- The data recorded for locks indicate that the same thread frequently reacquires the same lock, thus facilitating local optimization of lock acquisition. Also, the number of threads waiting on the same lock is usually quite small, indicating that lock arbitration will not require excessive network traffic.

Type-specific Coherence Mechanisms

Existing software distributed shared memory systems [Cha89, Li86, Li89, Ram88] have provided only a single memory consistency protocol. These systems typically use either an invalidation-based or an update-based consistency protocol, but not both. Munin allows a separate consistency protocol for each shared data object, tuned to the access pattern of that particular object. Moreover, the protocol for an object can be changed over the course of the execution of the program. We have shown that a large number of shared memory accesses can be captured by a small set of access patterns, for which efficient consistency protocols exist, indicating that this approach is both manageable and advantageous [Ben90a]. We have developed memory consistency techniques that can efficiently support the observed categories of shared data objects [Ben90b]. This section briefly describes these mechanisms in the context of the sharing categories that they serve.

Write-many objects appear in many parallel programs wherein several threads simultaneously access and modify a single shared data object between explicit synchronization points in the program. If the programmer knows that individual threads access independent portions of the data, and the order in which individual threads are scheduled is unimportant, the program can tolerate a controlled amount of inconsistency between cached portions of the data. The programmer uses explicit synchronization (such as a lock or monitor) to denote the points in the program execution at which such inconsistencies are not tolerable. *Delayed updates* allow *Write-many* objects to be handled efficiently. When a thread modifies a *Write-many* object, we delay sending the update

TOWARD LARGE-SCALE SHARED MEMORY MULTIPROCESSING

to remote copies of the object until remote threads could otherwise indirectly detect that the object has been modified. In this manner, by enforcing only the consistency required by the program's semantics, we avoid unnecessary synchronization and reduce the number of network packets needed for data motion and synchronization.

If the system knows that an object is shared in *Producer-Consumer* fashion, it can perform *eager object movement*. Eager object movement moves objects to the node at which they are going to be used in advance of when they are required. In the nearest neighbors example, this involves propagating the boundary element updates to where they will be required. In the best case, the new values are always available before they are needed, and threads never wait to receive the current values.

Migratory objects are accessed by a single thread at a time [Web89]. Typically, a thread performs multiple accesses to the object, including one or more writes, before the next thread starts accessing the object. Such an access pattern is typical of shared objects that are accessed only inside a critical section or through a work queue. The consistency protocol for migratory objects *migrates* the object to the new thread, provides it with read and write access (even if the first access is a read), and invalidates the original copy. This protocol avoids a write fault and a message to invalidate the old copy when the new thread first modifies the object.

Synchronization objects are supported by distributed locks. More elaborate synchronization objects, such as monitors and atomic integers, can be built on top of this. When a thread wants to acquire or test a global lock, it performs the lock operation on a local proxy for the distributed lock, and the local lock server arbitrates with the remote lock servers to perform the lock operation. Each lock has a queue associated with it that contains a list of the servers that need the lock. This queue facilitates efficient exchange of lock ownership. This mechanism is similar to that proposed by Goodman, et al [Goo89b].

Several categories of shared data objects can be handled in a straightforward fashion. *Private* objects are only accessed by one thread, so keeping them coherent is trivial. Replication is used for *Write-once* objects. *Read-mostly* objects are also candidates for replication since reads predominate writes. *Result* objects are handled by maintaining a sin-

SCALABLE SHARED MEMORY MULTIPROCESSORS

gle copy and propagating updates to this copy using the delayed update mechanism. Finally, *General Read-Write* objects are handled by the most convenient of the available consistency mechanisms.

Status

We are currently implementing Munin on an Ethernet network of SUN workstations. This implementation will allow us to assess the runtime costs of the delayed update queue and the other type-specific coherence mechanisms, as well as their benefits relative to standard consistency mechanisms.

In the Munin prototype system, the server associated with each processor is a user-level process running in the same address space as the threads on that processor. This makes the servers easier to debug and modify, which serves our goal of making the prototype system expandable, flexible and adaptable. We will be able to add mechanisms should we discover additional typical memory access patterns. We will be able to profile the system to evaluate system performance, and determine the performance bottlenecks. Running at user-level, the Munin servers will have access to all operating systems facilities, such as the file server and display manager, which will facilitate gathering system performance information.

The Munin prototype currently supports only six sharing types: *Read-only*, *Migratory*, *Producer-Consumer*, *Concurrent-write-shared*, *Result*, and *Reduction*. The sharing types *Write-once* and *Write-many* have been renamed *Read-only* and *Concurrent-write-shared*, respectively, since these terms more closely describe the manner in which objects of these types are accessed. *Reduction* is a new category for objects that are accessed via **Fetch_and_Φ** operations. Such operations are equivalent to a lock acquisition, a read, a write, and a lock release. Reduction objects are implicitly associated with a lock, and this lock is created automatically by the system at the time that the reduction object is created. An example of a *reduction* object is the global minimum in a parallel minimum path algorithm, which would be maintained via a **Fetch_and_min**. Reduction objects will be treated as migratory objects, but Munin will execute the operation in-place at a fixed location.

WILLOW

Overview

Although we believe that Munin will provide an effective computing environment for a large class of shared memory applications, programs that exhibit fine-grain parallelism and synchronization cannot be adequately supported due to the high latency associated with accessing remote memory. However, with true shared memory multiprocessors, contention for shared memory usually becomes a limiting bottleneck above a few tens of processors. Recent research efforts have begun to address this issue, and to investigate the feasibility of providing shared memory on large-scale multiprocessors [Len90, Aga90, Che91, Goo88, SCI90]. The Willow project at Rice University represents one of these efforts. It is our belief that large-scale multiprocessors, providing both shared memory and fine-grain parallelism, will offer an advantage, in terms of both cost and ease of programming, over existing approaches to large-scale multiprocessing. Therefore, the research question that we have posed is: Is it possible, using available technologies, to design a true shared memory machine capable of supporting on the order of 1000 processors? Our work to date suggests that an affirmative response is indicated.

We are currently designing a sixty-four processor prototype of Willow, a shared memory multiprocessor intended to ultimately provide memory capacity and performance capable of supporting over a thousand commercial microprocessors. These processors are arranged in cluster fashion, with a multi-level cache and memory hierarchy. Willow is distinguished from other shared memory multiprocessors by several characteristics:

- a layered memory organization that significantly reduces the impact of *inclusion* [Bae88] on the cache hierarchy, and that exploits locality gradients (variations in locality between “local” and “remote”),
- support for *adaptive cache coherence*, an approach similar to Munin’s type-specific coherence, whereby the consistency protocol used to manage *each cache line* is selected based on the expected or observed access behavior for the data stored in that line,

SCALABLE SHARED MEMORY MULTIPROCESSORS

- an efficient distributed update protocol that supports both read combining and write merging in the cache hierarchy,
- support for a range of relaxed consistency protocols so as to avoid the adverse performance impact of unnecessary synchronization, and to allow aggressive buffering and reordering of write operations,
- the use of layered I/O to provide symmetric multiprocessing, and
- hardware support for hierarchical synchronization.

Our goal in the design of Willow is to provide hardware support in those areas where such support is most beneficial to performance, and to relegate to software those areas of system support requiring greatest flexibility. Thus, in the design of Willow, we take particular care to provide efficient support for lightweight threads, simple synchronization (locks and barriers), fast context switching, and low-latency memory access. One of Willow's novel features is the manner in which cache consistency is supported. Instead of attempting to devise a single protocol best suited for all applications, we employ an adaptive scheme. With adaptive caching, each cache line can be managed with a consistency protocol most appropriate to the manner in which the cache line is being used. Cache lines can employ any of the following update mechanisms: write-through, write-back on any synchronization (lock release or acquire), write-back on release, write-back on acquire, write-back whenever convenient, or no write-back (i.e., read-only). Supporting different update mechanisms allows us to provide a range of consistency models:

Sequential all reads and writes are totally ordered [Lam79]

Processor allows reads to bypass buffered writes [Goo89a]

Weak allows reads and writes to be buffered, but all must complete prior to any synchronization (e.g., a lock release or acquire) [Dub86]

Release allows reads and writes to be buffered, but all must complete prior to release [Gha90]

Status

We have identified and addressed what we believe to be the most serious impediments to scalability: enforcing sequential consistency, inefficient synchronization, memory latency and bandwidth limitations, bus saturation, memory contention, the necessity to enforce inclusion on lower-level caches, and non-symmetric I/O. We are evaluating the preliminary design of Willow, a scalable shared memory multiprocessor whose design addresses each of these issues in a substantive manner. We have defined a basic system architecture and are validating this design using detailed simulation and analytic techniques. This architecture is depicted in Figure 1.

Willow has a tree-like bus hierarchy that contains two types of modules. At the processor level, the leaves of the tree are processor-cache modules. All other levels consist of cache-memory modules. Figure 1 depicts Willow with a clustering factor of four, that is, four processor-cache modules are arranged in a cluster that share the processor-level bus with a memory-cache module. Four memory-cache modules on this level are arranged in a cluster that shares a bus with a memory-cache module on the second level. This fan-in continues until the lowest level is reached where only one memory-cache module exists. At this level, a global interconnect provides direct communication between the cache component of the lowest level module and all of the memory components of all the memory-cache modules. In a symmetrical Willow system of 1024 processors, a clustering factor of four implies one level of processors and five levels of memory-cache modules.

Each memory-cache module contains memory, a cache, and control and synchronization hardware. Each memory communicates with the level above it (i.e., toward the processors) via an intracluster bus, and communicates with any other memory using the global interconnect. Physical memory is divided among the memory-cache modules. These modules are uniformly addressed via a global address space. When a request appears on a bus, the memory and cache components of the module simultaneously look up the address. The address space is partitioned so that a memory module can determine if it can satisfy a memory request. If the data is not located in the memory or cache of the module, the request is propagated down to the next level. If data is not

SCALABLE SHARED MEMORY MULTIPROCESSORS

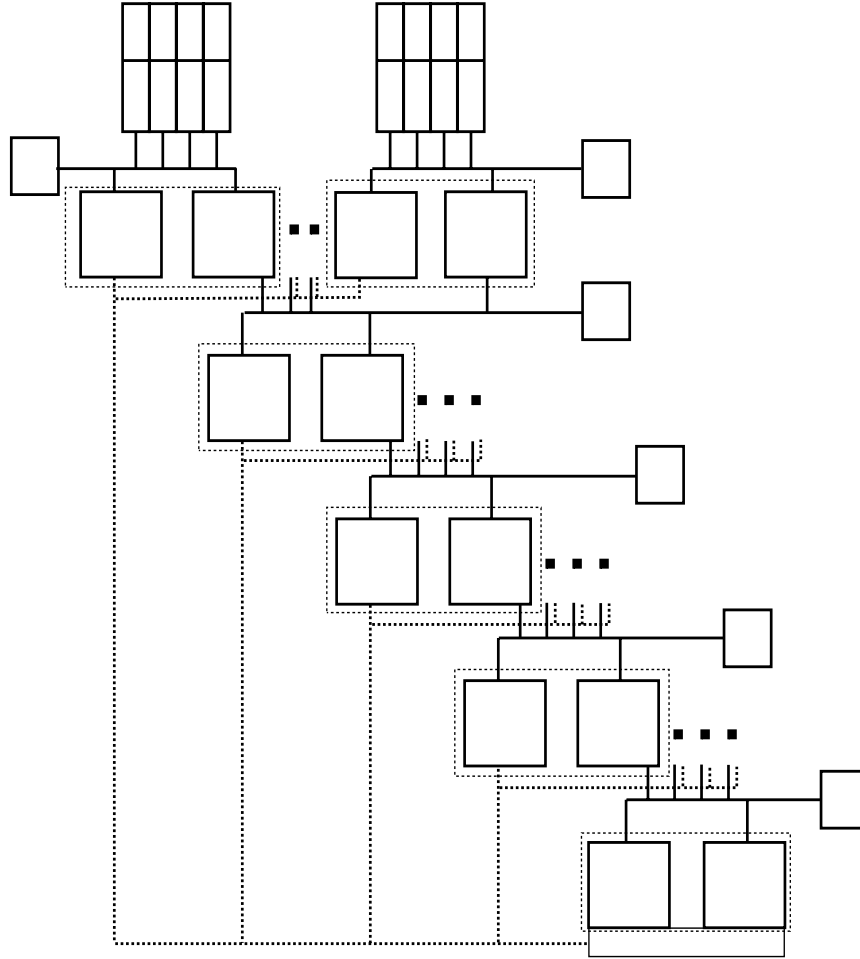


Figure 1: Willow System Architecture

found on the path from the processor to the memory module at the root of the hierarchy, the request is satisfied by the correct module over the global interconnect. Since all of the processors can access any part of the address space, Willow is a true shared memory system.

Data placement assumes special importance in Willow. Because we expect most parallel programs to exhibit strong locality, we have optimized the Willow architecture to be able to exploit this locality. Ex-

TOWARD LARGE-SCALE SHARED MEMORY MULTIPROCESSING

amples of this optimization include distributing memory over the levels, using adaptive caching, and placing I/O on the intrasystem busses. Part of our approach has been to try to build upon our relevant experience with Munin, a distributed shared memory system developed at Rice, by exploiting information about shared data access behavior and by incorporating our ideas about adaptive caching and relaxed consistency.

CONCLUSIONS

We have described two different approaches to scalable shared memory that we are currently developing at Rice University: Munin, a distributed shared memory system implemented entirely in software, and Willow, a true shared memory multiprocessor with extensive hardware support for scalability. Munin is designed to allow parallel programs written for shared memory multiprocessors to be executed efficiently on distributed memory multiprocessors. Willow is intended to be a true shared memory multiprocessor providing memory capacity and performance capable of supporting over a thousand commercial microprocessors. Implementation of Munin is in progress; we are still designing Willow.

We have described our goals and methods in the design of both systems, and we have described their distinguishing features. For Munin, these features include support for multiple consistency protocols, matching protocol to data object based on the expected pattern of accesses to that object (type-specific coherence), and a relaxed model of memory consistency to mask network latency and to minimize the number of messages required for keeping memory consistent.

The distinguishing features of Willow include a layered memory organization that significantly reduces the impact of *inclusion* on the cache hierarchy and that exploits locality gradients, and support for *adaptive cache coherence*, an approach similar to Munin's type-specific coherence, whereby the consistency protocol used to manage each cache line is selected based on the expected or observed access behavior for the data stored in that line.

ACKNOWLEDGEMENTS

Other members of the Computer Systems Laboratory have participated in the development of many of the ideas that we have presented. We thank Jim Carson, John Mellor-Crummey, Valerie Darbe, Elmootazbellah Elnozahy, Kathi Fletcher, Jay Greenwood, David Johnson, Pete Keleher, Mark Maxham, Rajat Mukherjee, and Peter Ostrin for their contributions.

References

- [Aga90] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiawicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.
- [Bae88] Jean-Loup Baer and Wen-Hann Wang. On the inclusion property for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, May 1988.
- [Ben90a] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134, May 1990.
- [Ben90b] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pages 168–175, March 1990.
- [Ber88] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.
- [Cha89] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

TOWARD LARGE-SCALE SHARED MEMORY MULTIPROCESSING

- [Che91] David R. Cheriton and Hendrik A. Goosen. Paradigm: A highly scalable shared-memory multicomputer architecture. *Computer*, 24(2):33–46, February 1991.
- [Egg88] Susan J. Eggers and Randy H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–383, May 1988.
- [Egg89] Susan J. Eggers and Randy H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Systems*, pages 257–270, April 1989.
- [Gha90] Kouros Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.
- [Goo88] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, May 1988.
- [Goo89a] James R. Goodman. Cache consistency and sequential consistency. Technical Report Technical report no. 61, SCI Committee, March 1989.
- [Goo89b] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessor. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Systems*, April 1989.
- [Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- [Len90] Dan Lenoski, James Laudon, Kouros Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

SCALABLE SHARED MEMORY MULTIPROCESSORS

- [Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. Ph.D. thesis, Yale University, September 1986.
- [Li89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [Lov88] Tom Lovett and Shreekanth Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.
- [Ram88] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, June 1988.
- [Dub86] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, May 1986.
- [Sch87] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243, May 1987.
- [Sit88] Richard L. Sites and Anant Agarwal. Multiprocessor cache analysis using ATUM. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 186–195, June 1988.
- [So86] K. So, F. Darema-Rogers, D. George, V.A. Norton, and G.F. Pfister. PSIMUL: A system for parallel simulation of the execution of parallel programs. Technical Report RC11674, IBM Research, 1986.
- [SCI90] P1596 Working Group of the IEEE Computer Society Microprocessor Standards Committee. SCI: An overview of extended cache-coherence protocols. Draft 0.59 P1596, February 5, 1990.
- [Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.
- [Web89] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for*

TOWARD LARGE-SCALE SHARED MEMORY MULTIPROCESSING

Programming Languages and Systems, pages 243–256, April 1989.