Willow: A Scalable Shared-Memory Multiprocessor

John K. Bennett
Sandhya Dwarkadas
Jay Greenwood
Evan Speight

Department of Electrical and Computer Engineering
Computer Systems Laboratory
Rice University
Houston, TX 77251-1892

Phone: 713-527-8101 Ext. 2272
Fax: 713-524-5237
Email: jkb@rice.edu

Abstract

We are currently developing Willow, a shared-memory multiprocessor whose design provides system capacity and performance capable of supporting over a thousand commercial microprocessors. Most recently, we have focused our attention on the design of a sixty-four processor prototype that tests most of our ideas about scalability. The design of such a multiprocessor poses a number of challenges to the computer architect. In this paper we describe the factors that traditionally have limited the scalability of shared-memory systems. These include: enforcing sequential consistency, inefficient synchronization, memory latency and bandwidth limitations, bus memory contention, the necessity to enforce inclusion on lower-level caches, and limited I/O bandwidth. We then describe how the Willow architecture addresses each of these issues. Finally, we present data that evaluates the effect of the major architectural innovations in Willow on the performance of several parallel applications. These innovations include a hierarchical memory, cache, synchronization, and I/O structure that exploits program locality at all levels in the hierarchy, support for adaptive cache coherence, whereby the coherence protocol used to manage each cache line is chosen based on the expected or observed access behavior for that line, the use of a relaxed cache consistency model and aggressive write buffering, and an efficient access combining protocol within the cache hierarchy. Our data was obtained and confirmed using two different simulators, one a detailed hardware-level simulator, the other an execution-driven simulator whose accuracy was validated against the detailed simulator. The data related to I/O was obtained from an analytical model developed for that purpose.

This work was supported in part by the National Science Foundation under Grant CCR-9010351, and by a National Science Foundation Graduate Fellowship.

1 Introduction

There is an increasing demand and a compelling need for higher performance computing resources to address computing problems of significant magnitude. Virtually all designers of high performance computers, confronted with serious hardware limitations, have turned to some form of parallel computing in order to obtain the raw processing power required to begin to address these tasks. Having chosen to support multiple processors, the next issue facing the computer architect is the structure of the memory subsystem. Basically, memory may be *distributed* (each program has a block of memory directly accessible only to that processor), or *shared* (all processors can access all of memory). Variations in the middle ground are also possible. NUMA (Non-Uniform Memory Access) multiprocessors are a prevalent variation. In choosing between a shared, distributed, or hybrid memory system, the designer has previously had to choose between competing design maxims:

1. **Shared memory machines are easier to program.** This is because the programmer does not need to explicitly specify data motion (as is required with distributed memory machines).

2. **Distributed memory machines are easier to build.** It is easier to scale (increase the number of processors in) a distributed memory machine because no single point of contention, such as a bus connecting processors to memory, becomes a limiting bottleneck as the number of processors increases.

We are therefore faced with the challenge of designing a shared memory machine that scales well. Distributed shared memory, providing an abstraction of shared memory on distributed memory hardware, has met with some success in this regard. Our experience with distributed shared memory [**?**] has led to an understanding of the capabilities and limitations of this approach to scalable shared memory. The primary disadvantage of distributed shared memory systems is that light-weight parallelism cannot be supported due to the high latency associated with accessing remote memory. The traditional problem with true shared memory is that contention for shared memory 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 true shared memory on large-scale multiprocessors [**?**, **?**, **?**, **?**]. The Willow project at Rice University represents one of these efforts. We contend that large-scale multiprocessors, providing both shared memory and light-weight parallelism, offer an advantage, in terms of both cost and ease of programming, over existing approaches to large-scale multiprocessing.

We are currently designing a sixty-four processor prototype of Willow, a shared memory multiprocessor that will ultimately provide memory capacity and performance capable of supporting over a thousand commercial microprocessors. Our choice of a thousand as the target number of processors was somewhat arbitrary. We wanted a number large enough to expose all fundamental scalability problems, but small enough not to confront us with engineering and packaging challenges beyond the scope of an academic effort.

In this paper we first describe the factors that traditionally have limited the scalability of shared-memory systems. These include: enforcing sequential consistency, inefficient synchronization, memory latency and bandwidth limitations, bus contention, the necessity to enforce inclusion on lower-level caches, and limited I/O bandwidth. We then describe how the Willow architecture addresses each of these issues. Finally, we present the results of a number of experiments that evaluate the effect on application performance, both individually and collectively, of the major architectural innovations in Willow.

2 Why is Scalable Shared Memory Hard?

2.1 Bus and Memory Bandwidth

Shared-memory multiprocessors have traditionally had a structure like that depicted in Figure **??**. In such a system, each processor first presents an address to its local cache, then, in the case of a cache miss, the address is presented to the single shared bus. A response is then generated by either main memory or another cache connected to the bus. The scalability flaw in this arrangement is readily apparent: as the number of processors increases, the bus will saturate at the point where the memory can not keep up with the combined processor cache miss rate. For slow processors

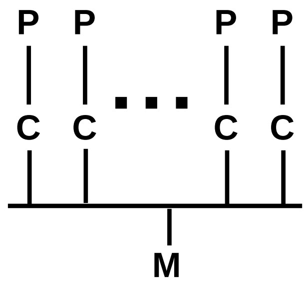

FIGURE 1: Basic Shared Memory System

and fast memory, systems like that depicted in Figure **??** may be able to support twenty or thirty processors, e.g., the Sequent Symmetry [?]. For fast processors the situation may be much worse. Consider a system that uses 50Mhz RISC processors, and whose processor caches provide a hit rate of 98% (2% miss rate). Assume further that cache lines are thirty-two bytes in size, that the bus has a 64-bit data path, and that on average memory can respond in $80ns$ (4 processor clocks). In such a system, a cache miss will require 16 processor clocks, during which time the bus will be unavailable to other processors. Thus, one out of every 50 processor cycles will incur a cache miss penalty of 15 cycles. In the scenario that we have just described, all available bus bandwidth will be utilized by as few as *four* processors, even when exacerbating circumstances such as cache line victimization, page faults, I/O, and maintaining cache consistency are ignored. Even if bus bandwidth were to be dramatically increased, contention for memory would lead us to the same conclusion: shared memory systems as we have known them are no longer a viable design alternative for large-scale systems.

We are thus led to consider a hierarchical memory organization, such as that depicted in Figure **??**. Here, lower-level intermediate caches are placed between the processor caches and global memory. Since the intermediate caches are two to four times faster than memory, we ensure a performance benefit by reducing the processor cache miss penalty. If intermediate cache C9 is two

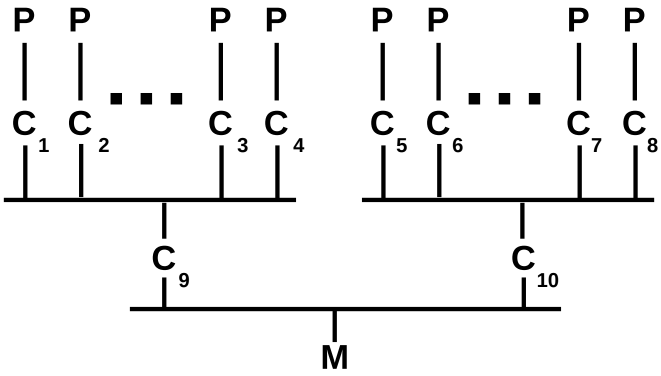

FIGURE 2: Hierarchical Shared Memory System

time faster than memory, and its hit rate is a modest 75%, the processor cache miss penalty will be reduced from 15 to 9 cycles. Unfortunately, the benefits of a hierarchical memory organization have been limited by the attendant increase in memory latency, and by the necessity to enforce *inclusion* on intermediate caches [?]. In particular, inclusion has limited the depth of the memory hierarchy, as we will see in the next section.

2.2 Multi-Level Inclusion

Figures **??** and **??** demonstrate the significance of the multi-level inclusion (MLI) property. Initially, the memory locations containing `a` and `b` are read by processors P1 and P2, respectively. Assume that MLI holds for intermediate cache C3, but does not hold for intermediate cache C4. Suppose that P1 updates `a` and P2 updates `b`. The resulting situation is shown in Figure **??**. Now consider what occurs when P1 reads `b` and P2 reads `a`. P2's read will miss in caches C2 and C4, and the (correct) current value will be supplied by cache C3 (depending on the exact cache protocol, C3 may first obtain an updated value from C1). P1's read, on the other hand, will miss in caches C1 and C3, but since C4 does not support MLI, the old (out-of-date) value of `b` will be provided by memory. The resulting situation is shown in Figure **??**. In this way, it is possible for processor caches C1 and C2 to each hold `b` in a cache line in an exclusive modified state [?], a condition likely to lead to incorrect program execution.

The need to enforce MLI thus requires that each intermediate cache contain the union of the contents of all caches above it (nearer to the processor). The resultant geometric growth in cache size effectively limits the depth of most cache hierarchies to two levels, thus directly impacting the scalability of these systems.

2.3 Consistency and Synchronization

The scalability of memory hierarchies is also limited in systems that enforce *sequential consistency* [?]. The necessity of propagating either invalidates or updates to topologically remote

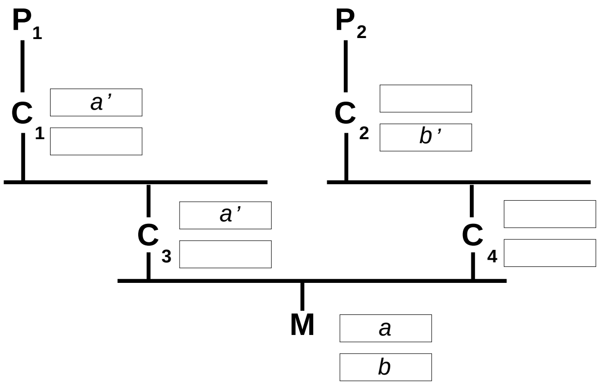

FIGURE 3: Multi-Level Inclusion (a)

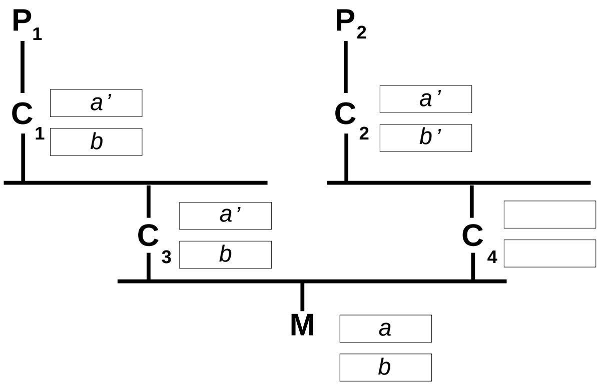

FIGURE 4: Multi-Level Inclusion (b)

processor-level caches forces some writes to exhibit very high latency. Furthermore, since most shared memory multiprocessors employ an invalidation-based protocol, and a write to any part of a cache line will force an invalidation of the entire line, considerable interconnect bandwidth can be wasted when fine-grained data sharing causes cache lines to "ping-pong" between caches. This *false sharing* can serverely limit scalability. Update protocols, particularly write-through protocols, can also consume interconnect bandwidth with unnecessary updates.

Synchronization between processors can also consume considerable interconnect bandwidth. In most multiprocessors, synchronization primitives are implemented utilizing atomic read and read-modify-write (RMW) operations, e.g., `Test`, and `Test & Set`. RMW operations attempt to read and then indivisibly modify a synchronization variable (which may or may not be an ordinary variable). In many multiprocessors, RMW operations are only supported in a particular region of the address space, typically a globally accessible portion of the I/O space. This approach to synchronization support limits scalability in several ways. Perhaps the most serious limitation is increased bus traffic due to RMW operations that fail (to acquire the synchronization variable), e.g., spinlocks. In this case RMW operations may be repeated many times, consuming interconnect bandwidth without contributing to the forward progress of the computation. Synchronization operations that cause the processor to be relinquished in the event of synchronization failure avoid this problem, but only for programs whose parallelism is very coarse-grained.

A second scalability limitation arises from the fact that more processors result in greater contention for synchronization resources. This added contention, coupled with the associated increased memory latency, results in a situation where processors will tend to remain longer in critical sections. This in turn will increase contention, further increasing average bus utilization and memory latency, etc.

Finally, hardware support for synchronization, if provided, has traditionally been provided as a global resource. All of the arguments made against global memory in Section **??** apply equally well against global synchronization support. In fact, the argument is stronger against global synchronization support, since synchronization accesses cannot usually be buffered. Systems that use cache lines to ameliorate this effect do so at the expense of increased cache miss rates due to synchronization variables victimizing cache lines used for instructions and data. This effect is pronounced in applications that require considerable synchronization, since a single synchronization variable must typically utilize an entire cache line.

Two methods for mitigating these effects suggest themselves: (1) reduce the number of highly contended locks, and (2) prevent failed RMW operations from acquiring shared resources. In Section **??** we will see that Willow employs both methods.

2.4 Input/Output

In a traditional shared-memory multiprocessor, such as that depicted in Figure **??**, I/O exhibits the same scalability problems as those encountered with bus and memory resources. If I/O devices are a global shared resource, every I/O access must make use of the shared bus. As the number of processors generating I/O accesses increases, bus bandwidth and latency increase and the bus saturates. This is a serious limitation on scalability, especially since I/O accesses and global memory accesses compete for bus the same bandwidth.

While a hierarchical memory structure (that retains global I/O) alleviates the single-bus bandwidth problem to the extent that access to global memory no longer dominates bus bandwidth, this approach does not reduce the effect of the I/O bottleneck. Moreover, it creates an equally serious problem, since global I/O can now cause invalidation and/or update traffic throughout the memory hierarchy, thus severely impacting cache hit rates and miss penalties. We are thus led to the conclusion that I/O resources must also be distributed throughout the memory hierarchy. We will see how this is done in Willow in the next section.

3 The Willow Multiprocessor

In Section **??**, we saw that the design of a scalable shared memory multiprocessor presents a number of special problems to the computer architect. In Willow, we have attempted to address these issues in an integrated manner, rather than in isolation. Perhaps the single common theme in the Willow design is *exploit locality at every opportunity.* This theme is readily apparent in Figure **??**. All major system resources: memory, caches, synchronization support, and I/O, are placed in a system hierarchy that permits local access to these resources without impacting more global resources of the same type. The processors in Willow are thus arranged in cluster fashion, with a multi-level hierarchy of system resources beneath them. Willow is distinguished from other shared memory multiprocessors by several characteristics:

- a hierarchical memory, cache, synchronization, and I/O organization that significantly reduces the impact of *inclusion* [?] on the cache hierarchy, and that exploits program locality at all levels in the hierarchy,
- support for *adaptive* cache coherence, whereby the coherence protocol used to manage each cache line is chosen (or changed) based on the expected or observed access behavior for that line,
- the use of a relaxed cache consistency model and aggressive buffering of writes to avoid the adverse performance impact of unnecessary invalidation and synchronization, and
- an efficient combining protocol that supports the merging of access requests within the cache hierarchy.

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.

3.1 Hierarchical Architecture

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. The simplicity of this scheme is of practical importance; a Willow system of arbitrary size is constructed of only two kinds of printed circuit boards. Figure **??** 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 next level. This fan-in continues until the lowest level is reached where only one memory-cache module exists. At this level, a high-speed (approximately 1 Gb/sec) 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. The appropriate clustering factor is an implementation technology dependent decision; we discuss this choice further in Section **??**.

Each memory-cache module in Willow contains system memory, an intermediate cache, and control and synchronization support 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 an access request is presented on a bus, the memory and cache components of the module simultaneously compare the address. Since each memory module occupies a unique portion of the global address space, memory modules can immediately determine if they can satisfy a given memory request. If the data is not

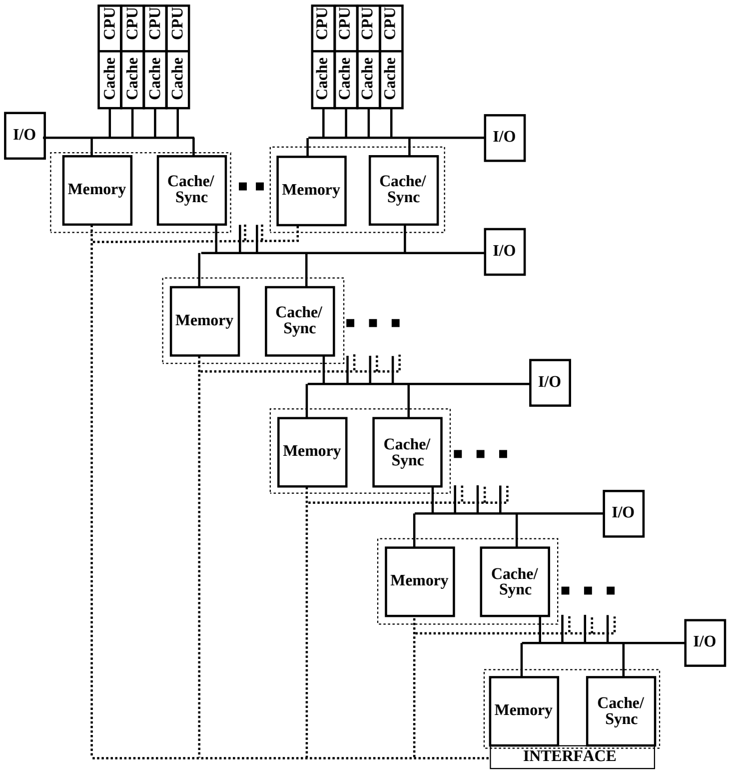

FIGURE 5: Willow System Architecture

located in the local memory, and also misses in the local cache of the current module, the request is propagated down to the next level. If data is not found on the path from the processor to the memory module at the root of the hierarchy, the request is satisfied by the correct memory module over the global interconnect. Thus all system memory in Willow is logically dual ported. One port is connected to the local bus, the other to the global interconnect. Depending upon the expected or observed access behavior for the remotely accessed data, this request is satisfied in one of two ways: (1) If the access is expected to be infrequent, a software directory-based cache entry is maintained at the target memory level that points to the requesting processor. On subsequent accesses, this directory cache will update or invalidate (over the global interconnect) remote copies of the data as appropriate. (2) If the remote access is a precursor to a significant number of accesses by several processors neighboring the requesting processor, it is more efficient to allow the data request to be satisfied through the most global bus-based cache, and then to "bubble-up" through the cache hierarchy. In this way, subsequent accesses by neighboring processors will be satisfied by a cache near the processor level, thus reducing the impact on global system resources. Placing memory in the hierarchy in this way provides two architectural advantages. First, inclusion-induced cache growth in lower-level caches is sublinear, rather than geometric, as a result of this memory "absorbing" localized references into the hierarchy. This architecture is also particularly well suited to multiprogrammed workloads, since it is possible for multiple parallel applications to proceed without competition for any system resources.

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

3.2 Adaptive Caching and Relaxed Consistency

Traditionally, shared memory multiprocessors have only provided a single memory consistency protocol. For example, existing machines use an invalidate-based or an update-based consistency protocol, but not both. Willow supports multiple consistency protocols, selected on a per-cache-line basis based upon the expected or observed behavior of the data object of which the cache line is a part. Moreover, this protocol can be changed during the course of the execution of the program. In an earlier study [**?**], we found that a large percentage of shared memory accesses can be characterized by a small set of access patterns, and for which efficient consistency protocols exist. This indicates that a system with multiple consistency protocols should improve the performance of most parallel applications, and that this approach is both manageable and advantageous.

Recently, several researchers in the shared memory 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 [**?**, **?**, **?**, **?**]. Willow's *adaptive caching* protocols can therefore support a variety of well-known consistency models:

Sequential Consistency – all reads and writes are totally ordered [**?**]

Processor Consistency – allows reads to bypass buffered writes [**?**]

Weak Consistency – allows reads and writes to be buffered, but all must complete prior to any synchronization access [**?**]

Release Consistency – allows reads and writes to be buffered, but all must complete prior to release [**?**]

The protocol to be employed for a particular cache line is held within the state bits for that line. These bits are set when loading the new line based upon the address of the shared data within the global address space. The shared data address space is partitioned by convention for this purpose. This simple scheme allows the user or compiler to control cache behavior via simple directives to the loader. Many variations in cache consistency protocols are possible. A protocol is defined by the answers to several questions related to how accesses are handled:

1. On a write, are other copies of shared data updated or invalidated?
2. When is the update or invalidate propagated, i.e., can it be buffered?
3. If writes are buffered:
   (a) When must the buffer be flushed (e.g., on which synchronization events)?
   (b) Can reads bypass buffered writes?

Thus coherence protocols determine when writes by one processor become visible to other processors.

3.3 Synchronization Support

We have designed two forms of hardware support for synchronization. The first is a `Conditional Test & Set` operation [?] that significantly reduces the number of failed RMW operations. The second is a distinguished synchronization memory that provides efficient access to synchronization variables.

`Conditional Test & Set` is an extension of the well-known `Test & Set` operation that is supported in many extant multiprocessor architectures. Our implementation of `Conditional Test & Set` reduces the number of bus accesses required for a successful RMW operation to the minimum number attainable in the single bus case, and to a number near to the minimum in a hierarchical bus architecture. This is achieved by conditionally scheduling all synchronization bus accesses. A `Conditional Test & Set` operation performs a `Test & Set` in the processor cache, which will write-through to the bus if the `Test` phase returns a zero value (indicating that the lock is not currently held). These bus writes arbitrate for control of the bus with all other processors attempting to acquire the lock variable. The first such processor that successfully arbitrates for bus mastership performs the write. This action updates the value of the lock variable in all other processor caches, thus causing subsequent `Test & Set` operations to spin in the processor cache. This successful bus write also unschedules the pending bus writes of the other processors, and a failure to hold the lock variable is returned to these requesting processors.

Willow provides a dedicated region of memory for synchronization variables that is replicated at each level of the memory hierarchy. For access purposes, this memory appears as an extra region in each cache, but the synchronization memory differs from cache memory in two important ways. Since the synchronization memory is fully mapped (as if it were always resident in the cache), there is no need for cache address tags to be associated with each "line" of synchronization memory. Eliminating these tags reduces synchronization latency by a factor of two, since an address compare is not required to determine a synchronization cache "hit" prior to a data access. Furthermore, the synchronization memory does not consume either memory or cache resources. In particular, by not using ordinary cache lines to hold synchronization variables, we avoid the adverse impact on cache miss rates due to synchronization-induced victimization.

The synchronization region is quite large (256K locks). This is feasible because the amount of state required to implement a synchronization variable is small. Even tree-structured barriers can be implemented with a few bits per line. This region directly maps all synchronization variables accessible at that level. More global regions are effectively replicated in higher-level synchronization memories, thus allowing synchronization accesses to exploit program locality in the same manner as other access. In the extremely unlikely event of an application that requires more than the provided number of the hardware synchronization variables, the programmer transparently uses regular memory and cache to supplement the synchronization memory provided.

3.4 Cache Design

Figure **??** depicts the (simplified) internal structure of an intermediate cache in Willow. The upper and lower sets of tags allow simultaneous address comparison on the upper and lower buses. Only in the event of a cache hit is the actual data accessed. Write buffers are provided for both the upper and lower buses, the "Up Queue" and "Down Queue", respectively. These buffers are snooped, and provide the necessary state to support *split transactions*. Both read and write combining is supported in the cache, thus reducing traffic forwarded to more global regions of the memory hierarchy.

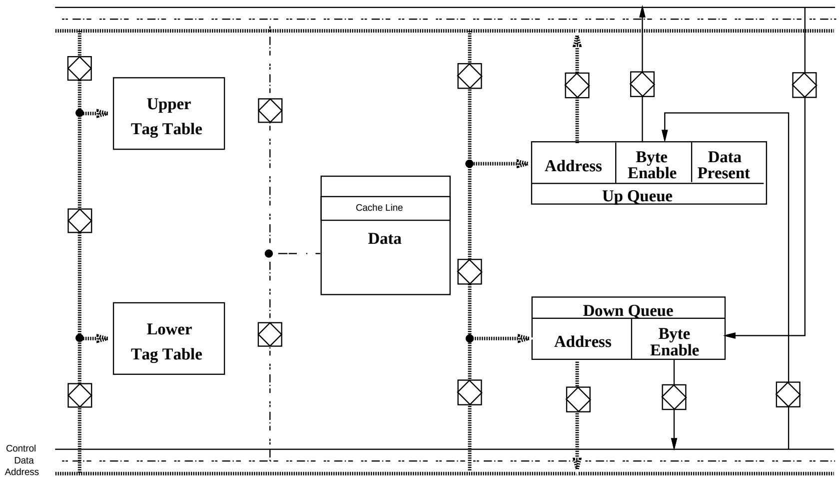

FIGURE 6: Simplified Cache Architecture

4 Performance Evaluation

In order to validate our design, we simulated the execution of several parallel applications on a variety of architectural variations. We compared these results to the performance of the same applications on a baseline single-bus configuration. We report the results of four programs: MP3D, SOR, Gaussian Elimination, and Matrix Multiplication. We used two different simulators to evaluate our design, one using the execution-driven approach (PARCSIM, a.k.a. RPPT [?, ?]), and the other using a cycle-level simulator derived from the the Sun Microsystems MPSAS simulator [?]. Most of the results reported use the execution-driven simulator. The detailed simulator was used to validate the execution-driven results, and to obtain detailed information regarding program execution, resource contention, pipeline effects, cache contention, write buffer management, and bus arbitration.

4.1 Results

Our primary metric for comparing the performance of different schemes has been processor utilization. Our goal was to minimize the frequency and duration of processor stalls due to memory access and synchronization latency. As expected, a hierarchical structure always outperformed the single bus architecture as the number of processors is increased, which supports the results reported in [?]. Our results indicate that the decrease in contention for the shared resources provided by a hierarchical scheme outweighs any increase in memory access latency.

Except as noted below, all caches were direct-mapped with a 32 byte line size and used a write-back-with-invalidate protocol for the caches at all the levels of the hierarchy. We employed a split transaction bus. The processor level caches were 64 kbytes in size and the ratio of the cache sizes from one level to the next in the hierarchical bus architecture was 1:4. We used a clustering factor of four; in other words, there were four processors/node per cluster. Figures **??** through **??** present speedup results for the indicated applications. Speedup was computed by comparing the execution time for each run with the execution time on a single processor with a 64 Kbyte direct mapped cache and a 32 byte line size.

Our implementation of SOR partitions the matrix into horizontal bands, which, depending on the matrix dimensions, may cause more cells to exist on partition boundaries than with other partitioning schemes. The advantage to this approach is that we can vary the amount of sharing, without having to increase the data size, by changing only the dimensions of the matrix. SOR (Figure **??**) shows superlinear speedup up to 64 processors. This is due to thrashing in the cache when the number of processors is small, because the total cache memory of all the processors will not hold all the data. As the number of processors is increased, the total cache size is increased, and the algorithm runs faster. It is appropriate to question the claim of superlinearity, since the hardware resources available in the multiprocessor case are considerably greater than those available in the single processor case. To explore this claim we performed an experiment assuming an infinite cache for the uniprocessor case. Using this result as the baseline for speedup comparisons, instead of the more realistic 64KB first level cache, we achieve a speedup of 52 for the 64 processor case.

Matrix multiplication (Figure **??**) requires essentially no communication between the processes working to compute their individual elements. As a result we expect large speedups. We achieve a maximum speedup of 55 using 64 processors. The reason that we do not achieve perfect speedup is that the processor-level caches are not large enough to hold all of the data. Therefore, these caches experience misses due to cache victimization during the computation for each result element. This miss rate does not change as processors are added since the matrices map on top of one another in the cache, and every processor reads the entire source matrix. The source matrices were not duplicated and placed in memory at each processor level, but were global, magnifying the problem as the number of layers in the hierarchy was increased.

MP3D is a particle simulator; the program divides up the space into cells and keeps track of each individual particle as it moves within this space. When two or more particles enter the same cell in the same time step, the particles collide in some random way. We ran the program configured as in [**?**]. MP3D (Figure **??**) performs relatively poorly, even with a hierarchical architecture. As reported in [**?**], this is because of the large number of invalidation misses caused by the random nature of the sharing of the space array. The performance of MP3D on Willow is comparable to that reported for the Dash multiprocessor [**?**].

Gaussian elimination (Figure **??**) achieves reasonable speedups even with a matrix size of 128×128 on 64 processors. Since each row of the matrix is read-shared during only one iteration, and is written by only one processor during the previous iterations, a write-back-with-invalidate protocol is appropriate. The speedup achieved is about 40 for 64 processors. This is because of the small size of the matrix used and the fact that the amount of work done by each processor per iteration decreased on every iteration. Hence, the amount of time spent in synchronization, relative to the time spent in computation, is large.

Willow Prototype Performance

The results presented thus far have assumed a copyback cache at every level in the hierarchy. Since the Willow prototype will use the Cypress CYC605 MMU/Cache Controller [?] for the processor-level cache, chip set [?], we evaluated configurations that were restricted to the cache protocol options available in the CYC605. Figure **??** depicts how the previous configuration compares with a system implemented using the CYC605 as the processor-level cache. Since the Cypress 605 does not support split-transactions on its bus, we expected this configuration to do less well than the previous system. This was indeed the case, with the copyback architecture showing speedups of 61.5 and 55 for the oblong and square SOR matrices, respectively. The CYC605-based system (write-through at first level, non-split transaction bus) exhibited a speedup of 56.5 on 64 processors for the oblong (8192×8) case, and 49.5 on 64 processors for the square (256×256) case.

The main goal of our study of hierarchical multiprocessors is to find the right combination of architectural features that will yield a scalable multiprocessor. The complement of this theme is to remove those features that do not significantly contribute to performance. With that in mind, we examined several architectural features of Willow in isolation.

Hierarchical Memory

Some hierarchical bus multiprocessors place all memory on the most global bus, e.g., [?]. This is reminiscent of interconnection multiprocessors with processors and caches separated by a switching network. We have designed Willow with memory modules on each of the hierarchies of buses. This allows us to take advantage of locality in any level of the memory hierarchy. This is especially useful when all of the data fits in the local memory and the second-level cache. For SOR, we see an 8% improvement using hierarchical memory against the case in which the same data is moved to the global memory. Somewhat counter-intuitively, hierarchical memory may in some cases do *worse* than the global memory case. This occurs when the data fits entirely within the second-level cache and the memory modules on that bus are significantly slower than the second-level cache (in our case they are twice as slow). Figure **??** depicts this situation. The architecture simulated was identical to the previous experiments, except for varying the placement of memory and the size of the cache (in the infinite cache case).

We expect that hierarchical memory will be most useful in two common but diverse cases. The first case is when the application program exhibits poor parallelism. In this case, the operating system can map the program to a memory module closer to the processors it will be using, which will reduce the contention for more global system resources (memories and buses). The second case that hierarchical memory will be useful is when the parallel program contains more data than any cache can hold, and this data is used primarily by a small set of closely-coupled processors. Many experiments with natural phenomena exhibit this behavior.

Adaptive Caching, Write Buffers, and Relaxed Consistency

In Section **??**, we described how researchers in the shared memory community have advocated the use of relaxed consistency models. Figure **??** shows the performance of SOR as the cache protocol and write buffer size is varied. The configuration of the hierarchy is the same as before, with a 64

Kbyte processor cache and split transaction buses. We simulated several protocols, including write-back-with-invalidate-when-shared and write-through-when-shared. As can be seen from Figure **??**, in the absence of write buffers, write-through-when-shared did poorly compared to write-back. We varied the write buffer size from 1 to 8 words. The performance of the write-through-when-shared protocol improved as the write buffer size increased, but still did worse than the write-back-with-invalidate protocol. This is because the sharing in SOR is fairly coarse grained. A process writes every byte in a line before it is read by its neighboring process. Hence, a fairly large write buffer is needed to hide the latency of the writes, and write-through exhibits worse performance than write-back because of the larger number of visible bus accesses.

Synchronization

According to Amdahl's Law, the maximum achievable speedup of a parallel program is limited by the fraction of the program that must be executed serially. The limit imposed by Amdahl's Law is in fact a conservative estimate. In general, parallel programs will perform more poorly than Amdahl's Law predicts [**?**, **?**], because communication costs in a parallel computation tend to increase as the number of processors involved in the computation increases. These communication costs include process partitioning and scheduling, and synchronization. It was clear from our earlier studies that synchronization would be a major performance consideration in systems with a large number of processors. For example, in Figure **??** we see that a global barrier implemented with a single counter protected by a `Test & Set` (`TAS`) lock requires fewer than 5000 cycles to synchronize for multiprocessors with eight or fewer processors. This is less than one percent of the execution times for these applications. Using the `Test & Test & Set` (`TATAS`) lock to protect the global barrier brought the overhead down to 2000 machine cycles. Unfortunately, the overhead associated with both `TAS` and `TATAS` increases exponentially as the number of participating processors increases. A `TATAS` barrier requires 60 thousand cycles to synchronize when implemented in a three-level hierarchy with 64 processors. The `TAS` barrier takes over one million cycles to synchronize.

There are two problems with global barriers: (1) bus contention when locks are released, and (2) the serialization of accesses to the counter associated with the counter. Using `Conditional Test & Set`, we eliminate the first problem, however barrier access is still serialized. To solve the second problem, we can create a hierarchy of barriers so that multiple processors can update the counters within their barriers in parallel. Processors that reach their respective barriers last thus enter the barrier at the next level. Although this hierarchical series of barriers is more complicated than a centralized barrier, removing the serial bottleneck significantly decreases the time required to synchronize.

Figure **??** presents results for the SOR algorithm using an 8192×8 matrix and several software barrier mechanisms. The configuration of the hierarchy is the same as before. `TATAS` synchronization was used for the barrier, so contention was an important factor in these results. An oblong matrix reduces the amount of sharing relative to the data size, which means that most of the loss in performance is due to the barrier synchronization at the end of each iteration. We plot graphs for SOR using a central barrier, an arrival tree barrier, and a *combining tree barrier* [**?**]. As can be seen, the performance of the central barrier is poor since it serializes each process's arrival at the barrier. Contention, although a secondary consideration, is also a factor. The arrival tree barrier does better, but does not benefit from performance gains obtained by distributing the barrier. The combining tree barrier does the best, approaching optimal performance. Using `Conditional Test & Set`, instead of `Test & Test & Set`, we saw a small (less than 1%) improvement in overall performance. The benefits of `Conditional Test & Set` are small in this instance because the barriers are distributed and the number of processors accessing each barrier is small (four in this case).

Clustering Factor

The fan-in at each level of the hierarchy affects program performance. If this clustering factor is too high (high fan-in), then contention becomes a concern since potential bus masters experience high bus access latencies. If the clustering factor is too low, maximum attainable speedup is reduced. We simulated several different configurations with varying clustering factors at each level to determine how sensitive the performance of applications programs was to the clustering factor. We employed an architecture most like the Willow prototype: three levels, 64 processors, a write-through cache protocol at the processor-level caches and a write-back protocol for the rest of the caches.

We ran SOR with a matrix size of 256×256 and 288×288. The reason we use different matrices is so all processors can be assigned the same number of rows to work on. Since 3 is not a factor of 256, we use the slightly larger matrix for the configurations with three processors on the first level bus. The average amount of time per matrix element stays nearly the same for both matrices, so the comparison is valid. Figure **??** depicts the sensitivity of SOR to the clustering factor. Configurations of four processors at the first level perform worse than most configurations with fewer processors at the first level, even though the second level buses are heavily loaded. This is because the write-through protocol generates enough traffic to increase the average time for a request to be satisfied. With fewer processors doing writes, the first level bus is better able to handle the write traffic. Efficiency also falls off if the second level bus is over-loaded, which occurs when there are more than 16 clusters connected to the second level bus.

To show that the main difference between the configurations is the added write traffic on the first level bus, we examined the performance of SOR with an 8192×8 element input matrix. This matrix is the same size as the previous square matrix, and the same amount of write traffic takes place, but the contention for shared elements is reduced. We see that performance improves because the latency of invalidating shared copies on updates is greatly reduced. By eliminating most of the shared traffic, we also reduce the number of transactions outside of the first level cluster, thus exposing the costs of the write transactions on the first level bus.

It is possible to improve the performance of the four-processor-per-first-level-bus case. Since SOR exhibits a large fraction of writes to private data, we could reduce the write traffic by implementing a copy-back or write-through-on-shared protocol.

Hierarchical Input/Output

The Models

We have developed an analytic model for the purpose of evaluating the I/O behavior of large-scale shared memory multiprocessors. Three different models were developed: a multilevel, multiprocessor system like Willow that implements layered I/O; a similar system without layered I/O; and a baseline system with one system bus shared by all processors, one large disk cache, and several disks. These models were developed using Yacsim, a discrete event-driven simulation language [**?**].

The Baseline Model

The baseline model consists of a single bus system with one disk cache and 21 disks attached to the system bus. Each of the 21 disks is modeled as a queue with an exponentially distributed average service time of $25ms$ and uses the shortest job next (SJN) scheduling algorithm. To model contention, the bus itself is modeled as a queue with a first come first serve (FCFS) scheduling discipline and an exponentially distributed service time whose average is $4.5ms$. The disk cache is not modeled directly as the cache access time is dwarfed by the service time of the other resources in the system. The service time required for disk cache access may be considered to be included in the arbitration time for the bus queue.

There are 64 processors, each of which may issue one I/O job at a time. In our simulations, these processors block until the I/O job is completed. The model can therefore be viewed as a closed network of 64 jobs that are either queued for service or delayed at the processor. When issued, a job requests and receives service from the global bus queue. If a cache hit is obtained, the job returns immediately to the issuing processor. Otherwise, the job requests and obtains service from a randomly selected disk and returns across the global bus to the issuing processor. The processor then "thinks" for an exponentially distributed amount of time (whose average is varied over different runs of the simulation) and re-issues its job. The simulation is run for 42,000 disk accesses, and the average time a job waits in each queue for service is calculated at each step in the simulation. The results of these simulations are shown in Figure **??**. The various curves in each figure represent different average times between I/O requests ("think times"). The Y axis represents the average time a job must wait in the resource's queue, including service time by the resource. The X axis is the baseline level cache hit rate for the disk cache. The effects of cold-starts and cache filling are modeled by adjusting this baseline cache hit rate according to the number of accesses determined to have hit in the cache. Thus, the hit rate will vary over the life of the simulation.

Hierarchical System with Non-Layered I/O

The second model is that of a hierarchical system, again with 64 processors. Each group of 4 processors is attached to a local bus and 4 of these buses are attached to a cluster bus. The resulting 4 cluster buses are connected to the global bus yielding a 3 level system with 64 processors at the highest level. A disk cache is associated with each bus. Each bus is represented by a separate FCFS queue in the model. The local bus queues have an average service time of $4.5ms$. The 4 cluster bus queues have service times of $3ms$, as does the global bus queue. The model contains 21 disk queues (with the same characteristics as in the base model) attached to the global bus.

The flow of a job through the system is as follows. The local level disk cache, cluster level cache, and finally the global cache are tested for a cache hit. On a hit at any level, the job returns to the processor by passing through the bus queues of the levels above it. On a miss, the job passes to the bus queue directly below it. On a global cache miss, the job enters the disk queue of a randomly selected disk (for simulation purposes). The job then returns to the processor via the global bus xxx

queue, the cluster bus queue, and finally the local bus queue. In addition, the block is "cached" in every level between the global bus and the local bus of the requesting processor. This is simulated by incrementing the number of hits in each cache by 1. The termination criterion for the simulation was once again the completion of 42,000 disk requests. The results are shown in Figure **??**.

Hierarchical System with Layered I/O

The final model implements layered I/O by distributing the 21 disks among the various local and cluster buses described in the preceding model, one disk per bus. In addition, the system contains a queue representing the global interconnect described in Section **??**. This FCFS queue has an exponentially distributed service time with an average of $3.85ms$. This includes the time to send/receive a request for data, or the data itself, as well as the time spent waiting to gain access to the global interconnect token ring network.

When a request for a disk block is issued from a processor, the block address will contain information that specifies exactly which disk contains the needed data. Only those disk caches in the path between the processor and the target disk need be checked for a cache hit. For example, if the block is known to be on the cluster disk, only the local and cluster disk caches are tested before the job enters the cluster disk queue.

If an address for a block that is located in a remote disk (one located on another cluster or local bus) is generated, the request will first be filtered down to the global level. A miss in all three levels will cause the job to pass through the interconnection network, gain access to the bus of a randomly selected disk, pass through the disk queue for service, and return to the requesting processor through the interconnect. The simulation results for 42,000 disk requests are shown in Figure **??**. The cache hit rates shown are varied according to the same scheme as the other two models.

Simulation Results

We expected the results for the base system case to be the worst of the three due to the fact that every request issued from any processor must pass through a single bus queue. This global bus queue quickly becomes a bottleneck for the system. In Figure **??**, we see that when the I/O requests are generated quickly enough (as in the $50ms$ curve), the global bus queue becomes saturated, as the vast majority of jobs in the system wait for control of the bus. The $50ms$ curve is invariant with increasing cache hit rates because every request, whether it hits in the cache or not, must arbitrate for the bus. As requests are spread further apart, some jobs in the bus queue have time to complete their service requirement before others get queued, resulting in a lower wait time. Wait time falls off at high cache hit rates as jobs are allowed more "think time". Results for the disk queue show relatively low waiting times regardless of "think time" or cache hit rates with an average time around $29ms$ (recall that the service time of one job is $25ms$). As expected, the bus queue is the major bottleneck in this system, preventing many jobs from queuing at the global disk.

In Figure **??**, we see an improvement in the wait times for the global bus queue, especially at higher cache hit rates. As more requests get satisfied in the local and cluster caches, fewer jobs require service from the global bus and the wait time decreases. Again, spreading the I/O requests out further reduces the wait times by allowing jobs to empty out of the queue before new ones take their place. The data for the disk queues again shows an average waiting time of around $29ms$ for all cache hit rates.

Figure ?? shows the results of implementing layered I/O in a hierarchical system such as Willow. Moving the disks into the upper levels of the hierarchy reduces the chance that a request will have to filter through the entire tree before being serviced. Thus, far fewer requests will reach the global bus, thereby reducing the amount of traffic on the global bus and lowering the waiting time for this resource.

In order for layering to be an effective method, the impact on other system resources of placing the disks at different levels must be minimal. The effects on the other queues in the system are shown in Figures ?? through ??. Figure ?? shows waiting times for the layered I/O case normalized to the non-layered I/O case. These results show a higher wait time for service by the global disk than in the unlayered case. This is because there is only 1 global disk in the hierarchical scheme instead of 21, causing more requests to be queued at the global level. Figures ?? and ?? represent the local and cluster bus wait times for the layered approach normalized to the unlayered, hierarchical approach. Figure ?? shows that there is a slightly longer average wait time on the local bus for the layered I/O approach for frequently issued I/O requests (the 50 and 200 *ms* curves). This is because requests are being generated at the local level that will not be passed down to the cluster level (if the data is known to be in the local disk). The overall time required for the local disk accesses is less than in the non-layered case in which the job will travel the entire tree before being serviced in the global disk. This extra latency in the non-layered case gives the local bus queue slightly longer to empty out, resulting in a lower average wait time for the local bus queue. Figure ?? shows that the cluster bus waiting times are almost identical for the layered and non-layered cases.

5 Related Work

Previous research in this area has taken two primary paths: (1) providing an abstraction of shared memory on distributed memory hardware [?, ?, ?, ?, ?, ?], and (2) providing support for true shared memory [?, ?, ?, ?, ?].

Our first efforts focused on distributed shared memory. We began by examining the sharing and synchronization characteristics of a variety of shared memory parallel programs [?]. We found that the access patterns of a large percentage of shared data objects fall into a small number of categories for which efficient, and fairly simple, coherence mechanisms exist. Based on these findings, we developed a multiple-protocol *adaptive* cache coherence mechanism that exploits semantic information about the expected or observed access behavior of particular data objects. Using adaptive caching, we were able to develop Munin, a high-performance distributed shared memory (DSM) system [?].

One of the first multiprocessors to recognize the need for non-uniform memory access (NUMA) was the Cm* [?]. The Cm* uses commercially available microprocessors in a general purpose cluster bus-based shared-memory multiprocessor configuration. However, the Cm* does not use caches, resulting in possibly excessive traffic for reasons other than coherency maintenance. Other examples of multiprocessor designs that do not use caches are the the BBN Butterfly [?] and the ILLIAC IV [?], both of which use multistage interconnection networks instead of buses.

Cedar [?] is a cluster-based architecture that connects clusters of eight processors through a global shuffle-exchange based interconnect to shared memory. Each processor has local memory associated with it. Cedar uses a combination of high speed shared memory, a macro data-flow model of computation, and compiler technology to achieve high performance.

The Data Diffusion Machine (DDM) [?] and Kendall Square Research's KSR1 [?] are both cache-only memory architectures. They use distributed main memory and directory-based cache coherence. One disadvantage of this approach is that allowing migration of data at the memory level requires a mechanism to locate the data on a miss [?]. The coherence protocol is also made more complex since the last copy of a data item must be migrated, rather than replaced.

The Wisconsin Multicube [?] is a cache-coherent, shared-memory multiprocessor whose design scales to over a thousand processors. Its topology is that of a grid of buses, with processing elements and caches at every intersection in the grid. It uses an invalidation based protocol, with a maximum of 32 processors per bus and very large snoopy caches. Our research has shown that placing 32 high performance processors on a single bus taxes the architecture heavily, resulting in potential poor performance for many algorithms.

The Stanford DASH multiprocessor [?] is another NUMA machine that distributes memory among the processing nodes. It uses distributed directory-based cache coherence and maintains cache coherence by sending point-to-point messages between the nodes on an interconnection network. Each node is a bus-based cluster of four processors, thereby utilizing the bus's snooping capability. DASH hides network latency through the use of *release consistency*, and by providing memory access operations for prefetching and delivering data.

Like DASH, Alewife [?] is a cache-coherent, directory-based, shared-memory multiprocessor whose network is a low-dimension mesh. Its main difference structurally is each node contains a single SPARC-based processor, rather than a cluster. The April processor was designed to hide the latency of network communication by supporting fast context switching, coarse grain multithreading, and full/empty bit synchronization. We have also studied the effect of using fast context switching to hide network latency [?].

Several commercial multiprocessors (Sequent Symmetry [?], DEC Firefly [?], Encore Multimax [?]), have been built with a shared bus architecture. Since shared bus architectures do not generally scale well, several hierarchical-bus architectures have been proposed.

Wilson [?] proposes a cache-coherent multiprocessor architecture with hierarchical buses. Like Willow, caches are placed at every level, linking processor to bus and every level to every other level. Wilson's design does not provide for memory at each level, which introduces a bottleneck if the caches are not large enough to fit all the program's accessed addresses. Wilson also uses a single cache consistency protocol.

The ParaDiGM shared-memory multiprocessor [?] is structurally similar to our multiprocessor. It is a hierarchical-bus cluster multiprocessor. Its memory is distributed amongst the clusters and at the bottom of the hierarchy. The hierarchy of buses is complemented by a network connecting the clusters together. Therefore, data can be passed between clusters in two fashions – through the hierarchy of buses or over the network. The caches connecting levels of buses are large, directory-based caches, whose coherence protocols are implemented primarily in software. Software-based coherence gives the programmer flexibility but increases the off-cluster access time.

Archibald [?] presents a distributed write adaptive snooping protocol that dynamically determines whether a block is being actively shared. He extends the protocol to cluster-based hierarchical multiprocessor organizations, introducing cluster ownership for writing. Copies in other clusters are invalidated on a write in any cluster. Since cluster controllers do not contain data, all traffic must go to the processor level to obtain modified information. Willow provides copies of shared data throughout the hierarchy to reduce latency for reads and to reduce contention on the processor and intermediate buses.

6 Conclusions

We have described the architecture of the Willow shared-memory multiprocessor, and have motivated our design with a discussion of the architectural challenges associated with such an undertaking. We discussed the issues that affect scalability, most notably enforcing sequential consistency, inefficient synchronization, memory latency and bandwidth limitations, bus contention, the necessity to enforce inclusion on lower-level caches, and limited I/O bandwidth. We described how each of these issues is addressed in the Willow design. We then presented an evaluation of the Willow architecture based on the execution performance of several parallel programs. This data was obtained using two different simulators, one a detailed hardware-level simulator, the other an execution-driven simulator.

Additionally, the data related to I/O was obtained from an analytical model developed for that purpose. Using this model, we showed that layering I/O in a hierarchical system reduces contention for shared resources, particularly the global bus, while at the same time incurring little additional penalties for service at other resources. These results were moderately dependent on the cache hit rates at the various bus levels. Further work with actual applications needs to be performed to confirm these results. We are presently developing this capability for the Willow architecture simulator.

The results of our evaluation demonstrate the scalability of the Willow architecture to the extent that we have thus far been able to test it. We are currently developing a sixty-four processor prototype that tests most of our ideas about scalability. This machine will be used to further evaluate the Willow architecture.

Acknowledgements

The Willow Project has benefited from the efforts of many of our colleagues within the Computer Systems Laboratory. In particular, Valerie Darbe helped develop some of the features of the Willow intermediate cache.

References

[1] *SPARC RISC User's Guide*, second edition, February 1990.

[2] S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2–14, May 1990.

[3] A. Agarwal, B. 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.

[4] J. Archibald. A Cache Coherence Approach for Large Multiprocessor Systems. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 337–345, July 1988.

[5] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[6] J. Baer and W. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 73–80, May 1988.

[7] Henri E. Bal and Andrew S. Tanenbaum. Distributed Programming with Shared Data. In *Proceedings of the IEEE CS 1988 International Conference on Computer Languages*, pages 82–91, October 1988.

[8] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 125–134, May 1990.

[9] J. K. Bennett and R. Mukherjee. Fast Context Switching on a Register Window Architecture. Technical Report 9202, Rice University, Department of Electrical and Computer Engineering, March 1992.

[10] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, February 1992.

[11] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

[12] 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 12th ACM Symposium on Operating Systems Principles*, April 1989.

[13] D. R. Cheriton, A. Gupta, P. D. Boyle, and H. A. Goosen. The VMP Multiprocessor: Initial Experience, Refinements, and Performance. In *Proceedings of the 15th International Symposium on Computer Architecture*, May 1988.

[14] D. R. Cheriton, H. A. Goosen, and P. D. Boyle. Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture. *Computer*, 24(2):33–46, Feb 1991.

[15] R. G. Covington, S. Dwarkadas, J. R. Jump, S. Madala, and J. B. Sinclair. The Efficient Simulation of Parallel Computer Systems. *International Journal in Computer Simulation*, 1:31–58, January 1991.

[16] 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.

[17] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, Coherence, and Event Ordering in Multiprocessors. *Computer*, pages 9–21, February 1988.

[18] S. Dwarkadas, J. R. Jump, and J. B. Sinclair. Efficient Simulation of Cache Memories. In *Proceedings of the Winter Simulation Conference*, December 1989.

[19] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3), March 1989.

[20] S. J. Eggers and R. H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 373–383, May 1988.

[21] Encore Computer Corporation. Multimax Technical Summary. January, 1990.

[22] M. Federwisch and L. Ball. *MPSAS: A Programmer and User Manual.* Sun Microsystems, 1990.

[23] H. P. Flatt and K. Kennedy. Performance of Parallel Processors. *Parallel Computing*, 12(1):1–20, 1989.

[24] Brett D. Fleisch and Gerald J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[25] G. Feierback and D. Stevenson. The Illiac-IV, in *Infotech State of the Art Report on Supercomputers*, Maidenhead, England, 1979.

[26] S. H. Fuller, J. K. Ousterhout, L. Raskin, P. I. Rubinfeld, P. J. Sindhu, and R. J. Swan. Multi-microprocessors: An Overview and Working Example. *Proceedings of the IEEE*, 66(2):216–228, February 1978.

[27] D. D. Gajski, D. J. Kuck, D. H. Lawrie, and A. H. Sameh. CEDAR - A Large Scale Multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, pages 524-529, August 1984.

[28] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[29] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. 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.

[30] A. Glew and W. Hwu. Snoopy Cache Test-and-Test-and-Set Without Excessive Bus Contention. *Computer Architecture News*, 18(2):25–32, June 1990.

[31] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report #1006, University of Wisconsin-Madison, February 1991.

[32] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 422–431, May 1988.

[33] E. Hagersten, S. Haridi, and D. H. D. Warren. *Cache and Interconnect Architectures in Multiprocessors*, chapter, The Cache Coherence Protocol of the Data Diffusion Machine. Kluwer Academic Publishers, 1990.

[34] J. R. Jump. *YACSIM Reference Manual.* Rice University, 1992.

[35] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs. *IEEE Transactions on Computers*, c-28(9):690–691, September 1979.

[36] Dan Lenoski, James Laudon, Kourosh 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.

[37] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[38] T. Lovett and S. Thakkar. The Symmetry Multiprocessor System. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[39] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21-65, February 1991.

[40] Myrias Corporation. System Overview. Edmonton, Alberta, 1990.

[41] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *Proceedings of the 1989 Conference on Parallel Processing*, pages II-160–II-169, June 1989.

[42] R. Retberg and R. Thomas. Contention is No Obstacle to Shared Memory Multiprocessing. *Communications of the ACM*, 29(12), December 1986.

[43] 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.

[44] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report, Stanford University, 1991.

[45] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 1992 International Symposium of Computer Architecture*, to appear, May 1992.

[46] C. P. Thacker and L. C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[47] A. W. Wilson Jr. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 244–252, May 1987.

FIGURE 7: SOR, 256*256 Matrix

FIGURE 8: Mat Mult, 128*128 Matrix

FIGURE 9: MP3D Speedup

FIGURE 10: GE, 128*128 Matrix

FIGURE 11: SOR, 8192*8 Matrix

FIGURE 12: SOR: Dist. vs. Global Memory

FIGURE 13: SOR: Various Protocols, 256*256 Matrix

FIGURE 14: SOR: Different Software Barriers, 8192*8 Matrix

levels of hierarchy	type of lock protecting the counter			
	TAS		TATAS	
	processors	cycles	processors	cycles
one level	8	5400	8	2100
	16	23000	16	6300
	64	one million	64	60000
two levels	8	1900	8	1500
	16	6100	16	3300
	64	21300	64	7700
three levels	8	1200	8	1200
	16	2000	16	2000
	64	3000	64	3000

FIGURE 15: Average Number of Cycles to Get from One Barrier to the Next

FIGURE 16: Efficiency of Different Branching Factors

Figure 17: Base Global Bus Queue

Figure 18: Non-Layered Global Bus Queue

Figure 19: Layered Global Bus Queue

Figure 20: Normalized Global Disk Queue

Figure 21: Normalized Local Bus Queue

Figure 22: Normalized Cluster Bus Queue