The following paper was originally published in the

Proceedings of the 3rd USENIX Windows NT Symposium

Seattle, Washington, USA, July 12–13, 1999

WINDOWS NT IN A CCNUMA SYSTEM

B. Brock, G. Carpenter, E. Chiprout, E. Elnozahy, M. Dean, D. Glasco,
J. Peterson, R. Rajamony, F. Rawson, R. Rockhold, and A. Zimmerman

Windows NT in a ccNUMA System

B. Brock, G. Carpenter, E. Chiprout, E. Elnozahy, M. Dean, D. Glasco,
J. Peterson, R. Rajamony, F. Rawson, R. Rockhold, A. Zimmerman

IBM Austin Research Laboratory
Bldg. 904, MS-9460, 11501 Burnet Road
Austin, TX 78758
mootaz@us.ibm.com

Abstract

We have built a 16-way, ccNUMA multiprocessor prototype to study the feasibility of building large scale servers out of *Standard High Volume* (SHV) components. Using a cache-coherent interconnect, our prototype combines four 4-processor SMPs built using 350MHz Intel Xeon™ processors, yielding a 16-way system with a total of 4 GBytes of physical memory distributed over the nodes. Such an environment poses several performance challenges to Windows NT®, which assumes that memory is equidistant to all processors. To overcome these problems, we have implemented an abstraction called a *Resource Set*, which allows threads to specify their execution and memory affinity across the ccNUMA complex.

We used a suite of parallel applications to evaluate the scalability and performance of the system. Our results confirm the feasibility of building ccNUMA systems out of SHV components, and suggest that memory allocation affinity should be incorporated as part of the standard Windows NT API. Also, the performance degradation due to poor bus bandwidth in the current generation of Intel-based processors often dominates the degradation due to the latency of remote memory accesses.

1. Introduction

There is an increasing need for powerful servers to meet the processing demands of modern distributed systems, transaction processing systems, and Internet data providers. Traditional server systems use proprietary mainframe computers or other large computer systems that are powerful and robust, though expensive. Recently, there has been an increasing demand for servers built using commodity, off-the-shelf processors and components. In particular, symmetric-multiprocessor systems (SMP) that use Intel's x86 processors and run Windows NT are in increasing favor due to their low cost, application software availability, and success in the personal computer and workstation markets. However, physical limits impose restrictions on the size of SMP systems, and most SMP systems based on the Intel Pentium® II processor contain at most 4 processors.

In order to investigate the feasibility of using these SHV components to build larger servers, we have used a cache-coherent interconnect to connect four 4-processor SMP nodes. Each node contains 1 GByte of main memory and four 350MHz Intel Xeon processors. The resulting system is a cache-coherent, non-uniform memory access (ccNUMA) machine, which poses several performance challenges to Windows NT since it is written to assume that all of system memory is equidistant to the processors. The focus of this paper is our addition to Windows NT to support memory allocation affinity.

Our enhancement to Windows NT 4.0 includes extending the Basic Input Output System (BIOS) and Hardware Abstraction Layer (HAL) to present the operating system with a single system image. These extensions do not require any modifications to the NT source code (to which we did not have access), and allow Windows NT to treat the system as a single 16-way SMP. The second major component of our enhancement is an implementation of a *Resource Set* abstraction (RSet), which allows application programs to control resource allocation in order to improve performance. The current implementation of RSets consists of a collection of Application Program Interfaces (APIs), Dynamic Link Libraries (DLLs), and a kernel-mode device driver that allows applications to control where memory is allocated.

We used RSets to tune a suite of six parallel programs, and studied the scalability of the applications under different system configurations. Without affinity in memory allocation, several applications scale poorly as the system's size increases. This result demonstrates that adding support for memory affinity to Windows NT's standard Application Programming Interface (API) and tuning the application to benefit from this extended API are necessary for ccNUMA systems such as ours. However, such tuning may not be necessary if the machine were to include a remote memory cache and larger processor caches.

Our results also suggest that the poor bus bandwidth (100MHz) of the current generation of Intel-based SMPs often has more of a detrimental effect on performance than the latency of accessing remote memory across the interconnect. This result is somewhat surprising since one would expect the latency of the NUMA interconnect to be the major source of overhead in a ccNUMA system.

Finally, applications that scale well on an SMP were found to continue to scale reasonably as the system size was increased, until the configuration parameters (bus bandwidth, remote memory latency, and L2 cache sizes) prevented any further scaling. On the other hand, applications with poor scalability trends on small systems were predictably unable to scale better on larger ones.

The remainder of the paper is organized as follows. Section 2 reviews previous work. Section 3 gives an overview of our implementation effort. Section 4 describes resource sets. Section 5 presents the results of the experimental evaluation. Our conclusions are presented in Section 6.

2. Previous Work

Over the years several research ccNUMA machines have been constructed, such as Alewife [Ale], DASH [Len] and FLASH [Flash]. These systems have been built either by constructing processor and memory nodes from scratch, or by combining pre-existing hardware using a combination of hardware modifications and interconnection fabrics. Recent years have seen the introduction of several affordable, general purpose ccNUMA and scalable shared-memory implementations such as the Silicon Graphics Origin™ [SGI], the Sequent NUMA-Q™ [Sting], the Data General Aviion™ NUMALiiNE™ [DG] and the Unisys Cellular MultiProcessor [Uni]. Except for the Origin, all of these systems use Intel IA-32 processors. The Sequent and Data General systems are built using standard high-volume 4-processor SMP nodes. The Origin uses the R10000™ processor and unique memory and I/O structures within as well as between 2-processor nodes. Unisys' Cellular MP also uses a unique arrangement of 2-processor nodes and claims uniform access times to memory. Our hardware prototype is based on an extension of the Fujitsu Synfinity™ interconnect used in Fujitsu's *team*-server™ [FJST].

Until recently, most of the work done on ccNUMA systems used a variant of UNIX as the operating system. For instance, SGI Irix™ 6.4 (Cellular Irix™) supports the ccNUMA features of the Origin while Sequent has historically had its own implementation of UNIX known as Dynix/ptx®. To the best of our knowledge, Microsoft's direction for scaling Windows NT[NT] beyond standard SMPs has been to provide clustering through the Microsoft Cluster Service[MSCS]. We are also not aware of any existing or planned support for ccNUMA systems in the standard release of Windows NT. However, some of the recent ccNUMA systems, including the Sequent NUMA-Q and the Fujitsu *team*server, have supported NT either as an alternative to UNIX, or as the primary operating system provided on the system. Our NT work is based on the Fujitsu implementation for two-node systems.

Although the Splash [Spl] and Splash-2 [Spl2] benchmark suites have been widely used in the academic community to measure multiprocessor performance [Shrimp, Len], most of the commercial server vendors have been more concerned about reporting performance data using the Transaction Processing Performance Council's TPC-C and TPC-D benchmarks [Gray, TPC]. However, while the TPC benchmarks are good indicators of overall transaction processing performance, they require a large investment to set up and execute properly. For this reason, we have chosen to study a subset of Splash-2 and other scientific workloads. With the exception of a few studies on distributed shared memory systems [Brazos, Sch], most of the Splash-2 studies reported to date have been carried out on UNIX systems.

The value of affinity scheduling has been recognized for SMP machines where many systems attempt to run a thread on the same processor that it ran on last in the hope of reusing data that is already in the cache [Vas]. Our affinity implementation may be viewed as an extension of this notion by colocating threads with the physical memory that they access. Although our implementation is indirect (since we do not have NT source code access), it should be relatively straightforward for Microsoft to implement similar functions directly in the NT executive.

The work of the FLASH project on improving data locality for ccNUMA machines used page migration and replication rather than affinity allocation to improve memory reference locality [Ver]. The Silicon Graphics Origin adds specialized hardware to support an efficient implementation of page migration. The Sequent, Data General and Unisys implementations all split the machine into communicating partitions, each of which runs a distinct copy of the operating system.

3. System Overview

3.1 Hardware Overview

We have constructed a 16-processor ccNUMA system by using a Synfinity interconnect switch to connect four PentiumII-based, Fujitsu *teamserver* SMP nodes. Each node contains four 350 MHz Intel Xeon processors, each with a 1MB L2 cache, 1 GByte of RAM, a standard set of I/O peripherals, and a Mesh Coherence Unit (MCU). The MCU provides coherent access to the memory and I/O devices that exist on other nodes. We designed a hardware card to attach the MCU to the Synfinity switch, which connects the four nodes together to form the 16-processor system. We configured the switch to provide 720 MB per second per link per direction in the prototype. A remote memory access is approximately 3 times slower than a local one.

The MCU in each node snoops the node's local memory bus and uses a directory-based cache coherence protocol to extend memory coherence across nodes. The MCUs exchange point-to-point messages over the switch to access remote memory and to maintain cache coherence over the entire system. The MCU defines a 4-node memory map that effectively partitions a standard 4 GByte physical address space into 4 areas of 1 GByte each, one for each of the nodes in a 4-node system. In addition to memory, memory-mapped I/O and I/O port addresses are also remapped to allow a processor to access the memory-mapped I/O and I/O ports of remote nodes.

3.2 Enabling NT on a ccNUMA System

We enhanced the BIOS and the NT Hardware Abstraction Layer (HAL) supplied by Fujitsu in order to enable Windows NT to run on the 16-processor system (the Fujitsu implementation could support a maximum of two nodes). When powered on, the system starts booting as four separate SMP systems. After the BIOS code on each node is executed, the system executes a BIOS extension (eBIOS) before booting the operating system. The eBIOS reconfigures the four SMP nodes into one 16-way ccNUMA system. Our modifications to the NT HAL support remote interprocessor interrupts, and provide access to remote I/O devices and I/O ports by remapping them as necessary. The combination of the HAL and eBIOS code presents Windows NT with a machine that appears to be a 16-processor SMP with 4 gigabytes of physical memory.

The eBIOS allows the system to be partitioned at boot time into smaller NUMA systems. For example, the eBIOS can partition the system into two 2-node systems, each with 8 processors and 2 GBytes of physical memory. Each partition runs a distinct copy of Windows NT. Other configurations for partitioning the 16-way system into separate systems are also possible. The eBIOS can also "deactivate" processors in a node at boot time allowing us to create nodes with fewer processors for configuration benchmarking purposes.

4 Supporting Memory Affinity in NT

Operating systems on SMP architectures try (when other constraints permit) to schedule threads on the same processor on which they have previously executed. Creating an affinity between a thread and its cache footprint in this manner results in good cache hit ratios, contributing to an application's performance. In addition to supporting such implicit "bindings", Windows NT also permits threads to explicitly specify the subset of processors on which they should be scheduled for execution.

If the performance of a ccNUMA system is to scale as more nodes are added, the operating system must accommodate the variability in memory access times across the system. In particular, a thread's memory allocation requests must be satisfied such that the majority of its memory accesses are served by the node on which it executes. Affinitizing memory allocations in this manner enables applications to take full advantage of the system hardware by reducing interconnect traffic. Indeed, an application may suffer in performance if most of its accesses are to memory residing on remote nodes.

Currently, Windows NT 4.0 considers all of the physical memory in a system to be equidistant to the processors. Since the physical memory frames are indistinguishable, NT does not have any mechanism for affinitizing memory allocations. We have implemented a solution that works around the performance penalties of this limitation. Our solution provides the application with an API permitting it to exercise control over the physical memory used to satisfy explicit memory allocation requests. In our experience, we find a resulting improvement in application performance suggesting that this kind of support should become part of the standard API for Windows NT.

Our ccNUMA API is based on a Resource Set (RSet) abstraction. Intuitively, an RSet groups several resources in such a way that a thread that is bound to a resource set consumes resources exclusively from that set. For example, one could specify an RSet containing the processors and physical memory available to one node. A thread that is bound to such an RSet will execute only on processors in that node, and have its memory allocations backed only by physical memory on that node.

RSets are flexible. They can combine the resources in two different nodes, include resources spanning different nodes, contain a partial set of the resources on one node, or any other combination that suits the application needs. Furthermore, they can be manipulated using union and intersection operations and can also form hierarchies, whereby one large RSet is made to contain several smaller RSets. To simplify the interface, our library provides a global RSet that contains all resources in the system. Thus, an application can build additional RSets by specifying subsets of the global one. We have implemented the RSet abstraction using an additional HAL call (through which we find the resources available in the system) and a combination of DLLs, backed by an NT kernel-mode device driver.

The RSet implementation provides fine-grained affinity control. Functions in the API fall into the following categories:

- Determining the system configuration.
- Creating and manipulating RSets.
- Allocating virtual memory that is backed by the physical memory contained in an RSet.
- Binding processes and threads to the processors in an RSet.

We have implemented the RSet abstraction using a combination of DLLs, backed by an NT kernel-mode device driver. Furthermore, we also provide a higher level API that provides a simplified interface to the RSet abstraction similar to traditional thread packages. Thus, an application programmer can use the RSet facility indirectly through the familiar interface of a thread library, or can access it directly to exercise greater control.

4.1 Allocating Virtual Memory Based on an RSet

There is no mechanism in Windows NT to constrain the set of physical memory pages that should back a range of virtual memory addresses. We have provided an interface similar to *VirtualAlloc*(), which supports the specification of an RSet. This interface allocates *locked* virtual memory that is backed by system memory as specified by the nodes identified through **memory_rset**:

```
void* NumaVirtualAllocLocked (
        void* start_addr,
        size_t *pages,
        RSet *memory_rset);
```

Despite the lack of directed memory allocations in NT, if one can ensure that the system memory backing a range of pages satisfies our requirements, locking the pages in memory forces the mapping to remain unchanged as long as the application is active. The challenge is to coerce NT into backing the virtual pages with memory from the requested node(s). To accomplish this, we have implemented the following approach. First, the *NumaVirtualAllocLocked* routine allocates the number of pages requested, mapping the virtual addresses into the caller's address space. It then increases the working set size of the calling process by the number of pages requested and uses *VirtualLock*() to lock the range into memory. Next, it passes the address and length of the virtual memory range to our *NumaMem* device driver which returns a list of the nodes whose real memory backs each page. For each page that is not "correctly" backed, that page is modified, released, and reallocated. This process repeats until all pages are correctly backed. The modify step was added once we observed empirically that it decreased the likelihood of NT handing the same page back to us.

The *NumaMem* device driver translates a given virtual memory address to its physical address, determines the node which "owns" that physical address, and returns that node identifier to the caller. For improved efficiency, a virtual address range can be passed to the driver, and a list of node identifiers (one per virtual page) will be returned. To enable the mapping between a physical address and a node identifier (as required by the *NumaMem* driver), we have exported an interface from our HAL implementation which provides not only the memory-range-to-node-id mapping, but also the system topology information (e.g. number of nodes, which processors are in which node, etc.).

The lack of NT source code access or an appropriate NT API dictates that we indirectly manipulate the page-table structure. As a result, the time required to set up the memory affinity mappings is large and unpredictable. Consequently, we would not advocate this implementation as a permanent solution to the lack of memory allocation affinity in Windows NT. Instead, this implementation allows us to study the potential benefits of including such support in the system. Our results suggest that this support should become an integral part of Windows NT as it moves to scale up to large system configurations.

A constraint of our approach is that we can only affinitize memory that is allocated through our API. Thus, an application must use our API instead of *malloc*() in order to allocate affinitized memory. In addition, we do not have any control over the placement of the program text, the data+BSS regions, and the individual thread stacks. These can be affinitized (or replicated in the case of program text) easily when memory affinity support is integrated within Windows NT

4.2 Page Coloring

Preliminary experiments on our prototype indicated that page coloring has a significant impact on application performance. With physically addressed caches that are not fully associative, a poor virtual to physical address space mapping can cause cache conflicts. Page coloring is a mechanism that can potentially reduce these conflicts. Each physical memory page is assigned a "color" such that similarly colored pages map into the same cache region. By cycling through the available colors when mapping contiguous virtual memory pages to physical pages, cache conflicts can be reduced when spatially close data structures are accessed. A significant advantage of page coloring is that it makes application performance predictable.

When virtual memory is committed using the *VirtualLock*() function, we discovered empirically that Windows NT backs up the virtual range with a set of physical pages that are almost perfectly colored. We wrote our *NumaMem* driver such that the pages it returns are perfectly colored. Lacking the ability to directly manipulate the page-table structure, *NumaMem* uses information about the cache size and the identity of the physical page that backs a virtual page to iterate until the pages are perfectly colored.

4.3 Affinity Policies

Using our Windows NT device driver, we have implemented two different classes of affinity policies: one for thread execution and one for memory allocation. The thread affinity policies are:

- **Float:** This is the default NT policy. Threads are eligible to run on any processor at any time. In order to maximize cache reuse, NT tries (when other constraints permit) to schedule a thread on the same processor on which it has previously executed [NT].
- **Fill:** In this policy, as many threads are bound to a node as there are processors before we continue to the next node.
- **Round Robin:** In this policy, threads are bound such that the first thread is assigned to the first node, the second thread to the second node, and so on in a round robin fashion.

We have several memory affinity policies, including:

- **Any:** The allocated virtual memory is backed by physical memory from any node in the system. This differs from the default policy only in that the virtual memory range is locked.
- **Any-striped:** This differs from **Any** in that the pages in the range are uniformly "striped" across the nodes in the system.
- **Local:** The allocated virtual memory is backed by physical memory local to the node on which the thread executes.
- **Remote:** The allocated virtual memory is backed by physical memory this is *not* local to the node on which the thread executes. This policy allows us to determine an application's sensitivity to memory affinity.

Generally, there is no single combination of these choices that yields the best performance for *all* the applications we studied. The next section presents a performance study using these allocation policies.

5 Experience

To test the effects of using the RSet abstraction to provide memory affinity support in Windows NT, we have conducted several experiments to study the performance of parallel applications on our prototype. The application suite consists of four applications from the Stanford Splash-2 benchmark suite [Spl, Spl2], a parallel program for matrix multiplication, and an implementation of a successive over-relaxation algorithm. The six applications are:

- Ocean-contiguous: 4-D 514x514 grids
- Raytrace: balls4 scene, Resolution = 256x256
- 3DFFT: 20 iterations, 64x128x64 3D array
- Water-spatial: 5 steps on 32768 molecules
- Matrix Multiply: 1024x1024 matrices
- Jacobi: 500 iterations on a 2000x600 array

The problem sizes were chosen such that even at 16 processors, the collective caches in the system will not accommodate the entire application data set. Note however, that the applications may work for the majority of their lifetimes with a much smaller working set.

We have modified each application to make use of the RSet abstraction. The extent of the modification depended on the application. For all of the applications except matrix multiply, the desired memory affinity could be achieved by just modifying the memory allocation calls of the key data structures at the beginning of the program. For matrix multiply, the code had to be rewritten to replicate one of the multiplier matrices over the entire NUMA machine complex. In this application, the second multiplier matrix is accessed by each thread, making it ineffectual to affinitize it to any particular node. Thus, in the absence of a remote memory cache, a large portion of the memory accesses are to remote memory. Replicating the second matrix avoids the performance hit.

We executed each of the modified applications on several system configurations under each of the policies for thread and memory placement listed in Section 4.2. Each experiment was repeated several times to ensure statistical accuracy. The time we report includes the time for running the application, but it does not include the time taken for initializing the memory pools in the *NumaMem* driver. This is consistent with our goal from the performance study, which is to identify the potential benefits of including memory affinity support inside Windows NT. The cost of initialization of the memory pools in the *NumaMem* driver is not relevant to this goal, since we are not advocating this as a technique for memory allocation anyway. Notice also that the initialization occurs at the beginning of the program, and before any processing or memory allocation takes place. Finally, in order to eliminate any effects of operating system synchronization overheads, all of our applications use user-level spin locks.

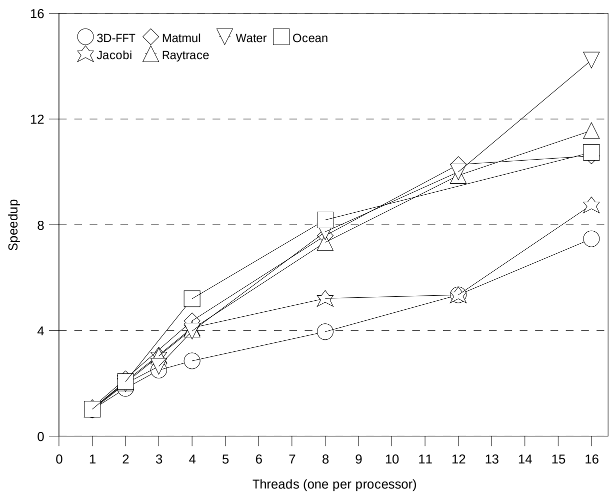

Figure 1: Best-case scalability

Figure 1 presents the best speedup that could be achieved as a function of the number of threads for each of the six applications. The base single-thread case was obtained by configuring the machine to run with a single processor and no remote memory. This base case corresponds to running the application on a uniprocessor machine containing the same amount of memory as an SMP node in our system, with the identical type of processor. The individual data points presented in this figure represent the best performance for each application and configuration over all thread and memory allocation policies. No one combination proved to be the best for all applications and thread configurations. Therefore, this figure serves to establish an upper bound on the scalability of our implementation.

Water exhibits the best scaling. This application has good locality in memory references, and not much sharing. Moreover, as the configuration grew larger, the aggregate total size of all L2 caches could store more of the application's working set. This effect also manifests itself in Jacobi, where the transition from 12 to 16 processors shows better scaling than from 8 to 12.

Three other applications (Matmul, Ocean and Raytrace) continued to improve as the number of processors increased, but the improvement started to taper off around 12 processors, due to a combination of bus bandwidth limitations and the effects of remote memory access. This phenomenon will be explained later in greater detail.

3D-FFT did not scale as well as the others. This application has substantial sharing among its threads causing it to scale poorly. Notice that 3D-FFT does not scale well on a single SMP.

Therefore it is reasonable to expect that it would not scale any better on larger machines.

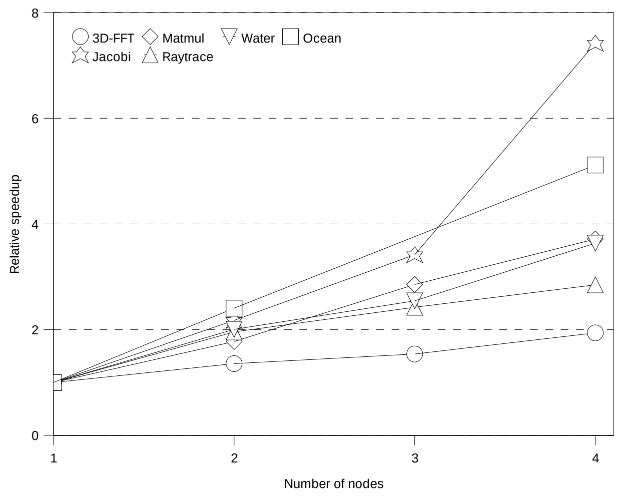

Figure 2: NUMA scalabiliy

Figure 2 shows the scalability of the applications on different ccNUMA configurations. This figure examines the benefits of implementing Intel-based multiprocessor machines larger than 4-processor SMP's. The baseline for this figure is the case where four threads execute on a single, standalone SMP node. The figure shows the scalability for the applications we studied as we run them on 8-way, 12-way, and 16-way ccNUMA machines. These are denoted in the figure by 2-node, 3-node, and 4-node, respectively. Each configuration was executed with as many threads as processors allocated **Round Robin**, and with memory allocated **Local**. Note that the scalability of some applications does not coincide with the best performance available. This is because the base case in Figure 2 is four threads running on a single SMP node, while the 4-thread data point in Figure 1 may correspond to an entirely different configuration. For instance, the best performing 4-thread case for Jacobi uses 4 nodes with the threads allocated **Round Robin** and **Local** memory allocation.

Figure 2 shows that NUMA scales approximately linearly for all applications except Jacobi. Ocean benefits from the increased collective cache capacity available in the system as nodes are added. Other applications such as 3D-FFT scale at a lower rate because of the increased inter-thread data sharing. Jacobi shows the best performance improvement as it scales from the 3-node to the 4-node configuration, because the aggregation of the L2 caches in the 4-node system can contain the entire working set of the application.

5.1 Effects of Allocation Policies

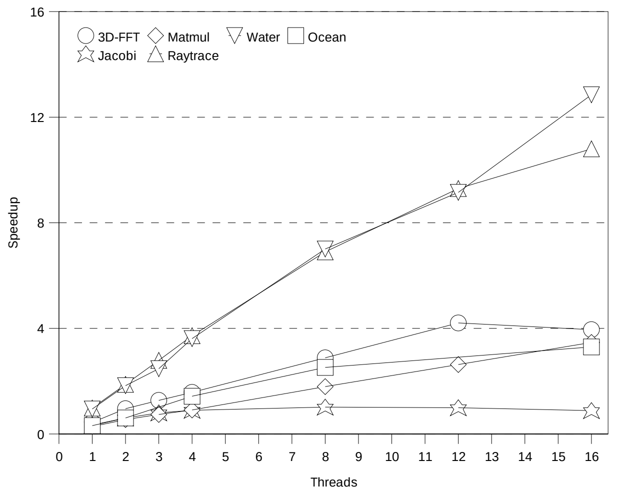

Figure 3: Base scalability: threads float, standard malloc

Figure 3 shows the scalability of the six applications when left unmodified. In this configuration, the threads are not bound to particular processors and memory allocation is done through the standard operating system mechanism. The results show that only two applications (Water and Raytrace) performed well in this configuration. The primary working sets of these applications fit within the L2 caches of the processors. Furthermore, there is no substantial sharing of data among threads during the computation. The remaining four applications did not scale well. For 3D-FFT and Jacobi, performance actually degraded when moving from 12-way to 16-way. This figure shows that scalability in performance will require application tuning and operating system support for memory affinity in ccNUMA machines built out of SHV components with no remote caches.

Figures 4 and 5 show the effect of tuning the applications to use Rsets. Two measurements are shown for each application, one with the application unchanged (broken line) and one with the application modified to benefit from Rsets (unbroken line). When the application is run unchanged, it uses NT's standard thread and memory allocation policies. When the application is tuned, it uses RSets to control thread and memory allocation. The thread and memory allocation policies used for these experiments are **Round Robin** and **Local,** respectively. The base single-thread case for these measurements was obtained by running the application with one thread on a single processor machine with no remote memory.

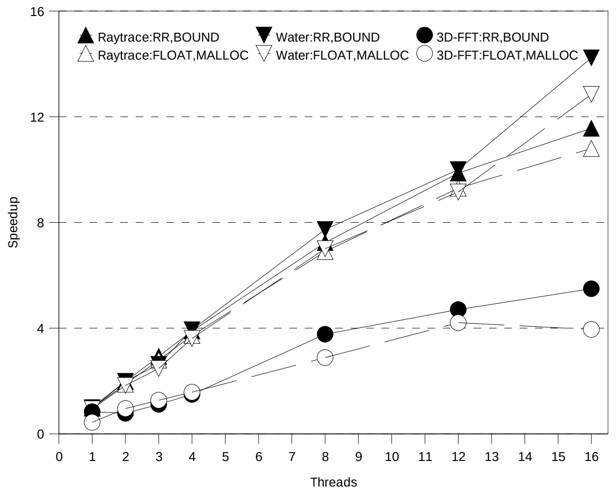

Figure 4: Applications exhibiting no affinity effects

Figure 4 shows that for 3D-FFT, Water, and Raytrace, there is not much to gain by modifying the application to use Rsets. Raytrace and Water work well with the caches in the system. Memory allocation in 3D-FFT is difficult to affinitize because over time, each thread accesses most of the application's data structures.

For these three applications, NT's default allocation policies yield good performance. Even though the unmodified versions of these applications do not specify a thread allocation policy, NT tries to reschedule each thread on the same processor on which it has recently run, so as to improve affinity in cache references [NT]. The NT memory allocator also worked well because we have observed that it uses page coloring to maximize the L1 and L2 cache performance. Since the primary working sets of Raytrace and Water fit within the cache especially in the large configurations, this policy yields good performance.

Figure 5 shows a different situation, where intelligent use of Rsets by the applications has a substantial impact on performance. When Rsets are not used, NT's default allocation policies do not allow the applications to overcome the effects of costly remote memory accesses. The difference in performance depends on the application, but in all cases, there is a clear gain from intelligently using Rsets. One can conclude from this figure that on ccNUMA machines that are architected out of SHV components with no remote caches, operating system support and tuning will be necessary to make these applications perform well. This result suggests that memory affinity support should become part of the standard Windows NT API if NT is to efficiently support this type of architecture. Although these conclusions depend on the fact that our system has no remote caches, it is not clear whether a remote cache will eliminate all the performance problems caused by NT's default memory allocation policies.

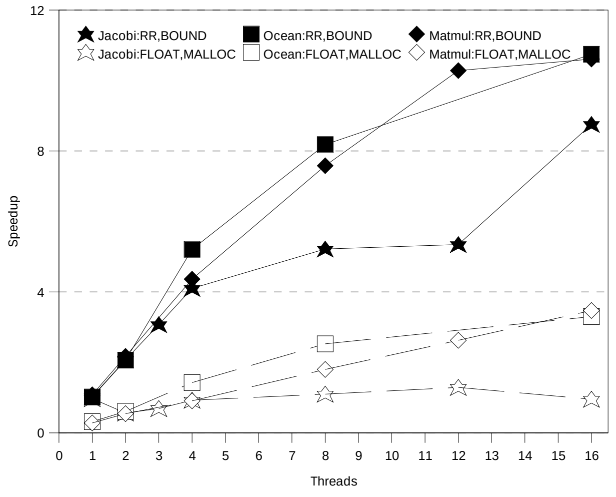

Figure 5: Applications exhibiting affinitization effects.

5.2 Effects of Local Bus Contention

Local bus contention is a serious problem as modern processors increase in speed. Figure 6 compares the speedup for the six applications in our suite under 3 different configurations. The configuration 1N4 was obtained by configuring the system as a 4-processor SMP, while 4N1 was obtained by configuring the system as a ccNUMA architecture with 4 nodes, each containing one processor. The 2N2 configuration was obtained by configuring the system as a ccNUMA architecture with 2 nodes, each containing two processors. Each configuration has exactly four processors. While 1N4 is a pure SMP system, 4N1 is a pure ccNUMA system. 2N2 represents a hybrid system.

The purpose of this experiment is to determine the impact that the local bus and the inter-node interconnect have on application performance. In the 1N4 case, the application threads face the maximum local bus contention and no interconnect effects. In the 4N1 case, the threads face the least local bus and maximum interconnect effects. The 2N2 case lies in between. Each thread was bound to a processor in this experiment. The memory allocation policy was **local**.

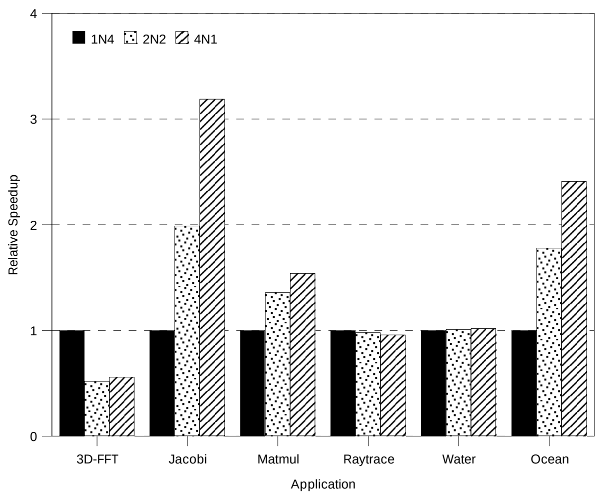

Figure 6: Local bus bandwidth effects.

For Raytrace and Water, there was not much of a performance difference across the different configurations. As mentioned earlier, these applications have small primary working sets that fit within the L2 caches, and exhibit modest inter-thread data sharing.

For 3DFFT, the effect of remote memory accesses dominates, and performance degrades as we move from an SMP to a ccNUMA architecture. The sharing pattern of this application is such that there is a continuous exchange of data among the threads. It is apparent that the bus bandwidth on a single SMP is adequate for the amount of data to be exchanged. Note that for this application, a remote cache would be of little help, because the contents of such a cache would be continuously invalidated as remote threads continue to modify the corresponding data.

For Jacobi, Matrix Multiply, and Ocean, the 2N2 and 4N1 configurations outperformed the SMP one. The reason is that the SMP case with four threads encounters significant local bus contention since the amount of data being accessed requires more bus bandwidth than is available with the existing Intel memory bus architecture. When the applications are modified to intelligently use RSets, remote memory accesses on the ccNUMA configurations have less of an effect. Therefore, the individual threads benefit from having less contention on the local bus.

5.3 Effects of Algorithmic Changes

Programs written to run in an SMP system will run without modifications on a ccNUMA system. However, it has been argued that NUMA-aware programs could further exploit the performance advantages of ccNUMA architectures, for instance by changing the algorithm used to exploit the characteristics of the NUMA environment (e.g. large amounts of physical memory). In our experiments, we have implemented a modified matrix multiplication algorithm in which the multiplier matrix is replicated at each node. Interestingly, this is similar to distributed algorithms that solve the same problem using message passing. Perhaps this suggests that transforming message-passing programs to run on NUMA systems will be more fruitful for performance tuning applications than just running existing SMP code.

6 Conclusion

We have built a 16-processor ccNUMA multiprocessor system using SHV 4-processor Intel Xeon SMPs. Windows NT was designed primarily to run on small SMP environments in which all processors have equally fast access to all the system memory. It therefore faces performance challenges in an environment where the processor-to-memory speed varies across a system. To overcome these problems and enable NT to run in this environment efficiently, we implemented an abstraction called the Resource Set that allows threads to specify where memory is to be allocated across a NUMA complex. Thus, threads can specify that memory should be allocated from banks that are close to the processors on which they run. This affinity in memory allocation can result in several performance benefits when running parallel applications. Our results indicate:

- The approach of building ccNUMA architectures out of SHV components is viable. For five out of six applications we studied, performance continued to improve in various degrees as we increased the number of processors. In general, it seems that in many cases the penalty of remote memory accesses can either be masked or does not have much of an effect to begin with.
- On architectures such as ours, where there are no special hardware assists or remote caches, scaling the performance of the application may require application tuning and operating system support for memory affinity.
- Memory allocation affinity should become a part of the standard API of Windows NT, as ccNUMA machines become increasingly common.
- For some applications, local bus saturation is the dominant performance impediment. This is somewhat surprising since one would expect

the latency of the NUMA interconnect to be a significant source of overhead in a ccNUMA system.

- Generally speaking, applications that scale well on an SMP system seem to also scale well on a ccNUMA environment, and vice versa.

Acknowledgments

We would like to thank HAL Computer Systems and Fujitsu for providing us with the MCU that we have used in our prototype, and also for providing their 2-node eBIOS and HAL code, which we extended to support a 4-node system. We also would like to thank the anonymous referees for their comments and Rich Oehler from the program committee for his help.

Appendix

The tables at the end of this paper contain the raw data used to create Figures 1 through 6.

Table 1 lists the execution times of the applications for different 16-processor configurations. All times are in seconds. BEST refers to the best-case scalability. For each data point, the configuration that yielded the least execution time is also provided. The BASE case provides the execution times when standard NT allocation policies are used. The TUNED case presents the execution time when the applications intelligently make use of Rsets. The data in this table was used to create Figures 1, 3, 4, and 5. The base single-thread execution time used to generate speedup numbers is given under each application. The single-thread execution was measured on a 1N1,t3,m0 system.

Tables 2 lists the execution times of the applications for different NUMA configurations. All times are in seconds. The allocation policy in each case is t0,m2. The data in this table was used to create Figures 2 and 6.

References:

[Alewife] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B. H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. Proceedings of the 22nd International Symposium on Computer Architecture, June 1995.

[Brazos] E. Speight and J. K. Bennett. Brazos: A third generation DSM System. Proceedings of the 1997 USENIX Windows/NT Workshop, August, 1997.

[DG] Data General's NUMALiiNE Technology: The Foundation for the AV 25000 Server, http://www.dg.com/about/html/av25000_foundation.htm

[FJST] W. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke, The Synfinity Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers, Proceedings of ISCA'97, the 24th Proceedings of the Annual International Symposium on Computer Architecture.

[Flash] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In Proceedings of the 21st International Symposium on Computer Architecture, Chicago, IL, April 1994.

[Gray] Jim Gray, ed., The Benchmark Handbook, 2nd edition, Morgan Kaufman, 1993.

[Len] D. E. Lenoski and W. Weber, Scalable Shared-Memory Multiprocessing, Morgan Kaufman, 1995.

[MSCS] Vogels, W., Dumitriu, D., Birman, K. Gamache, R., Short, R., Vert, J., Massa, M., Barrera, J., and Gray, J., "The Design and Architecture of the Microsoft Cluster Service - A Practical Approach to High-Availability and Scalability", Proceedings of the 28th symposium on Fault-Tolerant Computing, Munich, Germany, June 1998.

[NT] David Solomon, Inside Windows NT, 2nd edition, Microsoft Press, 1998.

[Sch] Martin Schulz and Hermann Hellwagner, Extending NT Virtual Memory by SCI-based Hardware DSM 2nd USENIX Windows NT Symposium, August 1998, Seattle WA, USA.

[SGI] James Laudon and Daniel Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server, Proceedings of the 24th International Symposium on Computer Architecture, 1997, pp 241-251.

[Shrimp] Yuanyuan Zhou, Liviu Iftode and Kai Li, Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems, Second Symposium on Operating System Design and Implementation, Seattle, WA, 1996.

[Spl] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. In Computer Architecture News, vol. 20, no. 1, pp 5-44.

[Spl2] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In Proceedings of the 22nd International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995, pp 24-36.

[Sting] Tom Lovett and Russell Clapp, STiNG: A CC-NUMA computer system for the commercial marketplace, Proceedings of the 23rd Annual International Symposium on Computer Architecture, May 1996

[TPC] Transaction Processing Performance Council, http://www.tpc.org.

[Uni] Unisys, Cellular MultiProcessing Architecture, http://www.marketplace.unisys.com/ent/Cmpwh.pdf.

[Vas] Raj Vaswani and John Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. Proc. 13th Symposium on Operating System Principles, October, 1991.

[Ver] Ben Verghese, Scott Devine Anoop Gupta, and Mendel Rosenblum, Operating system support for improving data locality on CC-NUMA compute servers. Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996.

Trademarks

App	Config	1-thread	2-thread	3-thread	4-thread	8-thread	12-thread	16-thread
3D-FFT 39.60	BEST	39.00 1N4,t3,m0	21.90 1N2,t3,m0	15.80 1N3,t3,m0	13.90 1N4,t3,m0	10.00 4N2,t0,m5	7.40 4N4,t1,m5	5.30 4N4,t0,m5
	BASE-NT	89.50	41.00	31.00	25.10	13.70	9.40	10.00
	TUNED	46.80	50.30	35.40	26.40	10.50	8.40	7.20
Jacobi 86.70	BEST	83.90 4N4,t0,m2	41.80 4N4,t1,m2	28.20 4N4,t1,m2	21.10 4N4,t1,m2	16.60 4N4,t1,m2	16.20 4N4,t1,m2	9.90 4N4,t0,m2
	BASE-NT	252.30	142.40	107.30	96.80	84.30	87.50	97.70
	TUNED	83.90	41.80	28.20	21.10	16.60	16.20	9.90
Matmult 102.50	BEST	97.10 1N4,t3,m1	56.00 1N2,t0,m1	38.90 1N3,t3,m1	33.50 1N4,t3,m0	42.40 4N4,t0,m2	32.40 3N4,t0,m2	28.10 4N4,t0,m2
	BASE-NT	358.20	186.30	138.50	112.20	56.80	38.90	29.50
	TUNED	102.20	143.40	114.90	93.60	49.50	34.90	28.10
Raytrace 114.50	BEST	110.70 4N4,t3,m1	56.60 1N2,t0,m1	37.70 1N3,t0,m1	28.20 1N4,t0,m1	15.60 4N4,t0,m5	11.60 3N4,t0,m2	9.90 4N4,t0,m1
	BASE-NT	116.10	60.60	40.90	30.70	16.60	12.30	10.60
	TUNED	116.80	57.80	39.00	29.40	15.80	11.60	9.90
Water 239.10	BEST	237.70 1N4,t3,m1	119.60 2N1,t0,m2	90.10 3N1,t0,m2	60.10 4N1,t0,m2	30.90 4N4,t1,m2	23.90 4N3,t0,m2	16.80 4N4,t0,m2
	BASE-NT	252.80	130.10	96.30	66.00	34.10	26.10	18.60
	TUNED	238.50	119.90	90.50	60.60	30.90	23.90	16.80
Ocean 17.20	BEST	16.80 4N4,t0,m2	8.30 4N4,t1,m2	x	3.30 4N4,t1,m2	2.10 4N4,t1,m2	x	1.60 4N4,t0,m2
	BASE-NT	52.60	27.90	x	12.00	6.80	x	5.20
	TUNED	16.80	8.30	x	3.30	2.10	x	1.60

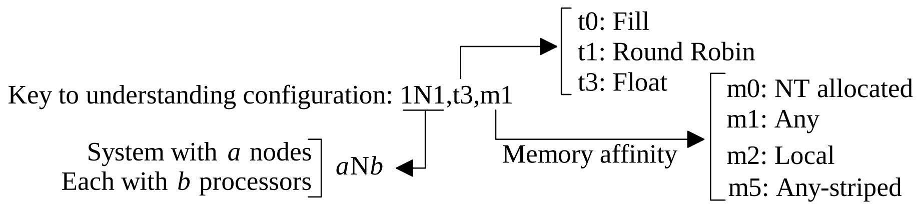

Table 1: Execution times for base, best, and modified cases.

Application	4N1	2N2	1N4	2N4	3N4	4N4
3D-FFT	24.90	26.70	14.00	10.30	9.10	7.20
Jacobi	23.00	36.90	73.40	33.80	21.50	9.90
Matmult	23.48	26.70	36.25	20.40	12.66	9.75
Raytrace	29.30	28.90	28.20	14.40	11.60	9.90
Water	60.10	60.20	61.20	30.40	24.00	16.80
Ocean	3.40	4.60	8.20	3.40	x	1.60

Table 2: Execution times for different t0,m2 NUMA configurations