Experience with building a commodity Intel-based ccNUMA system

by B. C. Brock
G. D. Carpenter
E. Chiprout
M. E. Dean
P. L. De Backer
E. N. Elnozahy
H. Franke
M. E. Giampapa
D. Glasco
J. L. Peterson
R. Rajamony
R. Ravindran
F. L. Rawson
R. L. Rockhold
J. Rubio

Commercial cache-coherent nonuniform memory access (ccNUMA) systems often require extensive investments in hardware design and operating system support. A different approach to building these systems is to use Standard High Volume (SHV) hardware and stock software components as building blocks and assemble them with minimal investments in hardware and software. This design approach trades the performance advantages of specialized hardware design for simplicity and implementation speed, and relies on application-level tuning for scalability and performance. We present our experience with this approach in this paper. We built a 16-way ccNUMA Intel system consisting of four commodity four-processor Fujitsu® Teamserver™ SMPs connected by a Synfinity™ cache-coherent switch. The system features a total of sixteen 350-MHz Intel® Xeon™ processors and 4 GB of physical memory, and runs the standard commercial Microsoft Windows NT® operating system. The system can be partitioned statically or dynamically, and uses an innovative, combined hardware/software approach to support application-level performance tuning. On the hardware side, a programmable performance-monitor card measures the frequency of remote-memory accesses, which constitute the predominant source of performance overhead. The monitor does not cause any performance overhead and can be deployed in production mode, providing the possibility for dynamic performance tuning if the application workload changes over time. On the software side, the Resource Set abstraction allows application-level threads to improve performance and scalability by specifying their execution and memory affinity across the ccNUMA system. Results from a performance-evaluation study confirm the success of the combined hardware/software approach for performance tuning in computation-intensive workloads. The results also show that the poor local-memory bandwidth in commodity Intel-based systems, rather than the latency of remote-memory access, is often the main contributor to poor

©**Copyright** 2001 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8646/01/$5.00 © 2001 IBM

scalability and performance. The contributions of this work can be summarized as follows:

- **The Resource Set abstraction allows control over resource allocation in a portable manner across ccNUMA architectures; we describe how it was implemented without modifying the operating system.**
- **An innovative hardware design for a programmable performance-monitor card is designed specifically for a ccNUMA environment and allows dynamic, adaptive performance optimizations.**
- **A performance study shows that performance and scalability are often limited by the local-memory bandwidth rather than by the effects of remote-memory access in an Intel-based architecture.**

1. Introduction

There is an increasing need for powerful servers to meet the processing demands of modern transaction-processing systems and Internet data providers. Several technologies are available to meet these demands, the most popular of which are clusters and symmetric multiprocessor machines (SMPs) [1]. Clusters are in widespread use because of their cost, reliability, and scalability advantages, but they also require substantial overhead in system management and maintenance. In contrast, SMP systems offer better performance, a simple programming model, and a single system image, which simplifies system management and maintenance. However, SMPs cannot scale beyond a limited number of processors because of technology limitations. Both cluster and SMP technologies are now entrenched in the mainstream of computing, with commercial pressures emphasizing commodity, off-the-shelf processors and components. In particular, SMP systems that use Intel** x86 processors and run Windows NT** [2] are increasing in favor because of their low cost, application software availability, and success in the personal computer and workstation markets.

Cache-coherent nonuniform memory access (ccNUMA) systems represent a third approach that overcomes the scalability limits of SMP systems while still providing a single-system image that simplifies management and maintenance [3]. A typical design in such systems uses several SMPs as computing nodes and connects them with a cache-coherent switch that supports shared memory across all processors. Special care is necessary to tune the hardware system for optimal performance. Several commercial systems use this technology [4–9].

There are many advantages to ccNUMA systems, including scalability, ease of management, and workload consolidation leading to reduced maintenance costs. Such systems also offer the option of partitioning the machine for failure containment, workload isolation, and management. However, ccNUMA systems pose several performance challenges because of their nonuniform memory access times. Internode memory accesses occur when a processor on one computing node requests a cache line that resides on a different node. Such memory transactions span two memory buses and a switch, in contrast to the case of an SMP, in which all accesses occur over a single shared bus, taking the same time to complete. Cache hierarchies can reduce the impact on performance by keeping the data close to the processors that use them. Commercial ccNUMA systems typically deploy a "remote" cache in each computing node, containing the recent references made by the processors in one node to memory banks in other nodes.

Caching techniques can reduce the performance impact on ccNUMA systems only up to a point. There are situations in which an application may be simultaneously writing to two independent regions within a cache line. Such *false sharing* within a cache line causes the coherence mechanism to move the line back and forth between different computing nodes. Additionally, some workloads can exhaust the capacity of the caches and the directory tables that manage coherence. Solving these performance problems is not trivial, and requires substantial investment in hardware and operating system tuning. This paper describes our experience with an alternative approach that simplifies the design and improves implementation turnaround time. It uses commodity hardware and software components as building blocks with minimal additional investments in hardware and software. To mitigate the performance problems, our approach relies on application-level tuning. This paper describes the design and evaluation of a system built using this approach.

We used a cache-coherent interconnect to connect four four-processor SMP nodes [9]. Each node contains four 350-MHz Intel** Xeon** processors and 1 GB of main memory. The resulting system is a 16-way machine that supports shared memory across all processors and aggregates the memories of all nodes into a single system image of 4 GB. Our design builds on top of the individual node design, which includes an additional "processor slot" in the memory bus for hosting a mesh coherence unit (MCU). The MCU module supports memory coherence across a switch.

The node design thus allows extension and scalability, but without any system-level tuning. In particular, it is not possible to add a remote cache or extend the cache hierarchy beyond what is available at the processor level. Also, some arcane instructions that access data across cache lines are not supported. We relied on tuning the application software to overcome these problems. Toward this goal, we have implemented a programmable performance-monitor card to assist programmers in understanding application behavior in the ccNUMA

platform. The monitor[1] tracks only remote accesses to application-specified regions of main memory. This allows the programmer to detect access patterns that cause performance problems. It does not impose any overhead in performance, and is different from traditional analysis tools based on tracing. While the monitor allows sophisticated dynamic performance adaptation, we have not exploited that feature in the work discussed in this paper.

We ported three operating systems to the prototype: Microsoft** Windows NT 4.0, SCO** (Santa Cruz Operation, Inc.) Unixware** 7.0, and Linux. We focus in this paper on the Windows port. This work was challenging because we had to work around the scalability limitations of Windows NT without having access to its source code. We used the device-driver model for kernel extensions that NT supports, and added several abstractions at the application level to overcome the performance limits of the system. We also extended the Basic Input–Output System (BIOS) and the NT 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, 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 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. Our results also suggest that the poor local-memory bandwidth 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.

The paper outline is as follows: Sections 2 through 4 describe the basic implementation effort, including hardware, performance monitor, operating system support, and the Resource Set abstraction. Section 5 describes the support for partitioning, and Section 6 presents the performance evaluation. The paper concludes with a discussion of related work in Section 7 and concluding remarks in Section 8.

[1] The card name is Opium, an acronym based on Olifant, the internal code name for the project. It is unrelated to recreational drugs.

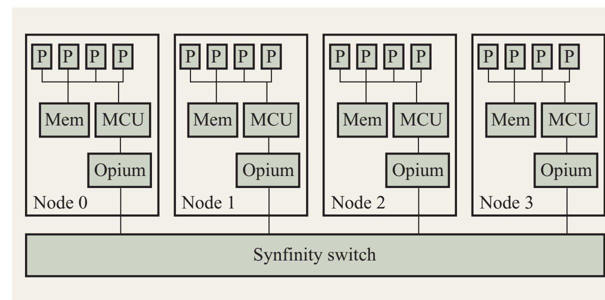

Figure 1

Hardware diagram for the ccNUMA system.

2. Basic hardware overview

Basic hardware description

We used a Synfinity** interconnect switch [9] to connect four Pentium** II-based, Fujitsu** Teamserver** SMP nodes, resulting in a 16-way ccNUMA system. Each node contains four 350-MHz Intel Xeon processors, each with a 1MB L2 cache, 1 GB 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 to form the 16-processor system. We configured the switch to provide a bandwidth of 720 MB per second per link per direction in the prototype. The original Fujitsu configuration supports a two-node ccNUMA system, wherein the MCUs of the two nodes are directly connected to each other.

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 four-node memory map that effectively partitions a standard 4GB physical address space into four areas of 1 GB each, one for each of the nodes in a four-node system. In addition to memory, memory-mapped I/O and I/O port addresses are also remapped to be accessible globally from any processor in the system. **Figure 1** shows a descriptive diagram of the hardware architecture.

The Opium performance monitor card

Description

The Opium performance monitor card consists of field-programmable gate arrays (FPGAs) and a 512K × 36-bit SRAM memory. **Figure 2** shows a simplified block diagram

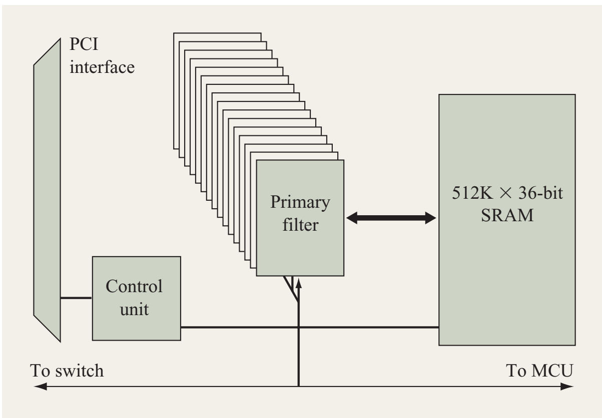

Figure 2

Block diagram of the Opium performance monitor board.

of the performance card. The FPGAs contain the necessary logic to capture information about the desired memory transactions on the ccNUMA interconnect; this information is then stored into the SRAM memory. The card has two interfaces. The first monitors the packets that go through the MCU by tapping into the ccNUMA interconnect. The second conforms to the PCI standard and plugs into the PCI bus of a host node. Packet monitoring is nonintrusive and does not affect packet timing; hence, the card monitors the remote-memory accesses that occur among the system's nodes without affecting their performance. The card does not, however, monitor the memory transactions on the local-memory bus on the node where it resides. Incorporating local accesses would have required a more powerful card with substantially greater resources, and it was not clear during the design whether adding this complexity would have been useful in achieving our aim of reducing remote-memory accesses.

The PCI interface serves several purposes, among which the most important are mapping the Opium card memory into the main memory of the host node and issuing interrupts to a local host processor whenever the FPGA logic detects the occurrence of a programmable set of events. The card is also programmed through the PCI interface to identify the memory-access transactions of interest that should be monitored, whether inbound to the SMP node, outbound, or both.

The FPGA on the card has 32 programmable filters that allow it to monitor memory references made to specific ranges of physical addresses. An application or the operating system programs a filter by setting five registers within the filter. The registers contain 1) the starting address of the region to be monitored, 2) its size, 3) the granularity of the measurement, 4) a bit mask of the transaction types to be monitored, and 5) the starting counter within the card memory. Each filter receives a copy of each memory transaction that travels on the ccNUMA interconnect. If a transaction has a type and memory address that match the filter, the latter increments a corresponding counter within the card memory. The base counter for the region specified within the filter and the measurement granularity determine the counter to be incremented. Thus, the programmable granularity within a filter allows the application to count references to multiple page-size memory regions or multiple cache-line-size memory regions. Each of the 512K counters can be programmed to generate an interrupt when a specific number of references occurs. The interrupt is issued as a regular PCI interrupt and given to the host processor. This feature allows the operating system or the application to obtain dynamic feedback about the "hot spots" within the monitored regions on the fly. (We do not discuss this feature or its use further in this paper.)

In addition to the region filters, there are 16 secondary counters, each of which contains a similar filter. These counters are programmed such that they shadow a set of the region counters, and are co-incremented whenever any such counter is incremented because of a given memory transaction. These counters reflect the cumulative counts of accesses to an area of virtual memory that may be allocated to scattered physical pages by the operating system. Moreover, the secondary counters can be programmed to count only a subset of the transaction types that are monitored by the region counters. This way, one can differentiate between read and write accesses to a particular memory region by setting different secondary counters to track each type of memory access.

Optimization methodology

A programmer can use the Opium board to isolate performance problems caused by nonlocal memory accesses and accordingly control the relative placement of threads and heap memory. The software interface consists of libraries that control access to the Opium card through a regular device-driver interface. During initialization, an application must make an initialization call to the library. As the library allocates memory, it captures the virtual-to-physical mappings (see Section 3) and writes a file that associates application data with its associated physical address (this is also saved in memory). Once the application has completed its initialization, it makes another library call to start the monitoring, specifying the regions to be monitored and the granularity at which the monitoring is to be done. When the application is ready to stop monitoring, it makes a corresponding library call. The result, along with the virtual-to-physical address mappings, is then input to a postprocessing program whose output is

a table listing each application data area with the number of references from each node.

The ability to specify variable-sized ranges readily suggests a measurement methodology based on successive refinements. First, start with coarse-grained memory regions such that the granularity of measurement in each region is large, say at the page level. Then, concentrate on the most frequently referenced pages by reducing the number of regions to be monitored and the granularity of the memory areas being monitored while collecting counts on the various parts of the pages. If necessary, we can continue this process down to individual cache lines. This successive-refinement approach is meant for the common situation in which access hot spots are not clearly defined, making it difficult to determine where to initially focus the performance optimization effort.

Analysis

Unlike many performance-optimization techniques that are used to guide the design of the hardware and the operating system, our focus is on optimizing application software performance after the system is designed. Thus, the performance-debugging methodology we advocate here is different from traditional techniques based on tracing or program annotations. A trace of an application's memory references can identify data-placement problems and performance hot spots [10]. To generate a trace, the application is run, and trace records are generated and stored for each memory reference of interest. Tracing can be done by hardware, e.g., using a logic analyzer, giving rise to no performance perturbation [11]. Alternatively, it can be done in software by instrumenting the application, but the perturbation introduced by the tracing code must be compensated for during postmortem analysis. Either way, tracing memory references generates voluminous quantities of data that are difficult to reduce and complex to map back to changes in program behavior. Furthermore, the management and engineering of these traces are not trivial.

An alternative solution to tracing is to use program annotation. This technique provides affinity in memory access through a combination of compiler analysis and operating system support for data placement. Often, source-level annotations may also be added to provide hints that aid the analysis. This technique tends to be useful when the code is well understood or is amenable to automatic analysis. However, in practice, application complexity makes it difficult to add the appropriate annotations, or makes compiler analysis intractable.

In contrast to tracing and annotation, our solution stresses flexibility, which we achieve through the programmable interface of the Opium monitor. The application or operating system sets the granularity of monitoring within each region, with the granularity ranging from 2 MB to a 32-byte cache line. Each filter has a range of counters on the card, such that there is a counter for each monitoring unit within the region. The output of the card is a list of the frequencies of remote-memory references that occur during monitoring, so that hot spots can be identified by high counts, while "local" memory accesses or those that are fielded by the local caches are identified by low counts.

To summarize, the novel aspects of our work which distinguish it from previous research are the following:

- Flexibility in monitoring through a fully programmable card that can be put under operating system or application control. This contrasts with approaches in which the monitoring granularity is fixed. Our approach also lends itself to a systematic performance-tuning process based on successive refinements.
- Ease of use by avoiding the complex engineering tasks of collecting and maintaining traces, or annotating programs. The histograms we provide can easily be projected on a spreadsheet to reveal access patterns and guide performance tuning.
- A nonintrusive technique, and one that can be applied to production systems without affecting performance.
- Focus on ccNUMA systems and limiting monitoring to remote-memory references. We believe that this choice allows us to concentrate on the most pressing problem in ccNUMA systems.
- Ability to generate performance-related interrupts to enable future research in dynamic performance tuning and adaptation.

These advantages come at some cost. The monitoring system cannot capture temporal changes in the remote-access references. Furthermore, since the card resides on the interconnect side of the MCU, it cannot identify the actual processor within a node that is responsible for an access. Nevertheless, our experience has shown that these problems do not prevent the successful application of the approach. Section 6 contains a performance evaluation using this approach.

3. Basic software support

The independent SMP nodes forming the ccNUMA system were designed to function as independent nodes. Thus, each node has its own independent boot program with no preexisting coordination to bring up the system as one unit instead of four independent ones. Additionally, none of the operating systems available to us were designed for this mode of operation, or for supporting ccNUMA. In particular, the operating systems were not designed to handle remote I/O devices, remote memory, or the special mechanisms for sending processor interrupts across the interconnect.

We have ported three operating systems to our platform with varying degrees of success. These are Microsoft Windows NT, SCO Unixware, and the Linux operating system from the open-source community. We describe here how we handled the problems and our experience in porting the various operating systems.

Extending BIOS, NT support, and Unixware

We enhanced the Basic Input–Output System (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 begins to boot as four separate SMP systems. After the BIOS code on each node is executed, the system executes our 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 GB 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 two-node systems, each with eight processors and 2 GB 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.

Porting Windows NT took one team member about two months to complete without any serious problems. We also ported Unixware with comparable effort. Both ports were stable and enabled many benchmarking runs and demonstrations. This is remarkable given the lack of source code for Windows NT save for the HAL. For Unixware, we had access to the source code, and the modularity of the operating system and the strict layering of software modules greatly simplified the port.

Porting Linux

Porting Linux to our ccNUMA system was attractive because of its popularity, the free availability of the source code, and the existence of a large base of users and expertise. However, our experience in porting Linux was not a positive one. It took three team members working more than six months, and we admit that the resulting port is not stable enough to run industry-standard benchmarks. The problems lie in the poor modularity of the operating system, its outdated technology, and the development process of the open source community.

Linux remains a workstation operating system developed and tested primarily on individual user workstations. By virtue of the large number of CPUs, I/O devices, and large amount of memory present in our system, our hardware stressed many areas of the operating system that were not usually tested. The initial porting efforts concentrated on extending the basic symmetric multiprocessor (SMP) capabilities and the large memory support of the Linux kernel itself. At the time we started the Linux port, Linux on the Intel x86 family supported only 1 GB of RAM. Later, the Linux community removed this restriction and added limited support for ccNUMA operation.

Our progress was slowed by having to keep up to date and cope with the rapid evolution of the Linux kernel. There were two kinds of changes which affected us directly. The first consists of frequent and significant changes introducing a major new functionality or structural change. The second consists of changes which are primarily cosmetic in nature but often are difficult to deal with. For these reasons, we have focused our benchmarking studies and performance evaluation on the Windows NT platform.

4. The Resource Set abstraction and its implementation

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. 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. Creating affinity for 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.

NUMA-aware applications thus require information about the underlying architecture in order to optimize performance. Applications could, for instance, use this information to colocate threads with the data structures they most often use, thereby reducing the frequency of costly remote-memory accesses. However, this requirement poses two problems: how to represent the information about the underlying architecture, and how to avoid creating platform-specific dependencies in the software that compromise application portability to other ccNUMA systems, including future generations of the current

platform. The Resource Set (RSet) abstraction solves these two problems. In brief, it intuitively captures the most common characteristics of ccNUMA machines and presents the application programmer with a high-level, portable abstraction. Applications written to this model do not require low-level information about the underlying architecture, and can easily be ported to other ccNUMA platforms. To implement this model, the operating system or application libraries must provide an efficient mapping from the RSet model to the actual architecture.

Description

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 can 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 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 needs of the application. 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 of the resources in the system. Thus, an application can build additional RSets by specifying subsets of the global one. An implementation must provide the RSet implementation with a mechanism to identify the resources available in the system. For instance, such support can be implemented in Windows NT 4.0 by using an additional call in the HAL.

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 dynamically linked libraries (DLLs), backed by an NT kernel-mode device driver. Furthermore, we also include a higher-level API which provides a simplified interface to the RSet abstraction that is 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.

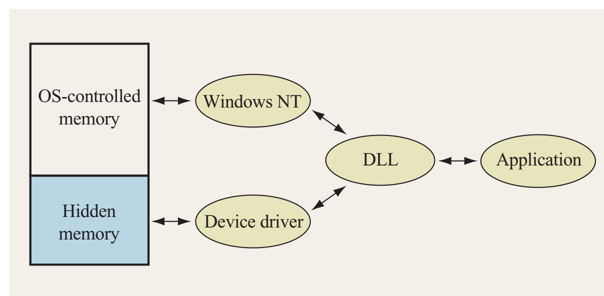

Figure 3

Memory affinity support application uses DLL and device driver to bypass operating system and controls memory allocation and placement.

Implementation

We implemented support for RSets in Windows NT 4.0, which already has processor-affinity support that allows an application to bind threads to particular processors. The RSet implementation uses this feature to provide control over thread placement. Windows NT 4.0, however, assumes that memory is equidistant from all processors in a system, and therefore does not supply any primitives to control memory placement. Adding such support directly to the operating system requires source-code access, so we resorted to using a device driver and DLLs to provide control over memory affinity. The device driver implements the mechanism to control the placement of data in physical memory, while the DLL implements the allocation policies and a high-level, machine-independent abstraction to enhance application portability. This way, the application can organize its data structures so that they will be close to the threads that use them the most.

The device driver is supported by a modified HAL which "hides" a portion of the physical memory on each node from the operating system. The HAL modifies the tables that the firmware constructs at boot time to describe the available memory. The modified tables are then passed to the Windows NT memory-management initialization code. The device driver manages this "hidden" memory as though it were a memory-mapped device. More specifically, the device driver uses Windows NT support routines that map physical memory into ranges of virtual memory. **Figure 3** is a diagram of the implementation.

The DLL implements the RSet interface and additionally exports an API for redirecting standard memory-allocation calls such as *malloc*. Applications use the DLL to issue normal memory-allocation requests,

which in turn are redirected by the DLL into the device driver interface. According to the policy specified by the RSet, the DLL translates the request to the device driver, which in turn carries out the request.

Thus, the application or middleware embedded in the library itself can allocate memory from a specific node to create affinity between threads and data structures. The "hidden" memory is not subject to paging or movement, since the Windows NT memory manager does not control it. Once memory is allocated, the pages remain locked in physical memory, and the virtual-to-physical translation does not change. Additionally, the device driver ensures that the virtual memory allocated during an application request will be backed by physically contiguous memory. As described in the next subsection, this feature reduces the resources required on the performance monitor card.

Memory-allocation policies

We have implemented several memory-affinity policies which, along with thread affinity, fix a particular relationship between the processor doing the computation and the memory being referenced. Several policies were implemented for the purpose of experimentation. A complete list of all implemented policies is not relevant to this discussion, so we focus on the most important ones. The first two policies enforce thread affinity, and the remainder, memory affinity. An application specifies one of each.

1. *Fill*: In this policy, as many threads are bound to a node as there are processors before we continue to the next node.
2. *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.
3. *Float*: With Float memory allocation, there is no enforced affinity. This allocation bypasses the driver-based memory allocator and uses the standard Windows NT memory-allocation primitives.
4. *Local*: With Local memory allocation, memory is physically located on the node where the allocating thread is running. This establishes an affinity between the calling thread and the allocated memory.
5. *Remote*: With Remote memory allocation, memory is allocated on some node other than the one where the calling thread is active. This policy is useful in performance debugging and experimentation.
6. *Striped* (with stride): With Striped memory allocation, each successive page of memory (or a specified number of pages—a "stride") is allocated from the node that is logically next in sequential order, so that the first "stride" pages are from node 0, the second "stride" from node 1, and so forth in a round-robin fashion. Many programs have the property that their threads use different sections of a common data structure. Striped memory allocation is appropriate for such structures, although to make the best use of them, it may be necessary to pad or insert fill into the data structures to ensure that the threads access disjoint subsets of the pages occupied by the data structure. Also, this policy could be used as a default when the relationship between threads and memory cannot be described.

5. Partitioning support

A principal advantage of a ccNUMA machine is that it offers a single system image across a large number of processors. It is often desirable, however, to partition a large ccNUMA machine into smaller, isolated configurations. Such partitioning is useful for isolating workloads, containing faults, providing high availability, and supporting different operating systems on the same machine [12]. Partitioning thus adds a high degree of flexibility. For example, a banking system may designate a portion of the resources in a ccNUMA machine to serve requests incoming via the Web, while the rest of the machine runs the bank's database. By strictly isolating these two workloads, the configuration eliminates interference in performance between the two applications, and enables them to use two different operating systems if required.

There are two types of partitioning. On one hand, *static* partitioning draws the boundaries between partitions at boot time. The boundaries can be changed only if the machine is restarted. On the other hand, *dynamic* partitioning can change the boundaries between partitions while the machine is running. The second form is a more powerful abstraction than the first, because it provides greater flexibility without affecting the availability of the machine. However, it requires nontrivial support.

Static partitioning

We introduced extensions to the BIOS to enable the user to control how the machine is partitioned at boot time. To simplify the implementation, the unit of partitioning was chosen to be an SMP node. Thus, it is possible to partition the machine into four independent four-way SMPs, two eight-way ccNUMA systems, one SMP and one twelve-way ccNUMA system, or one eight-way ccNUMA and two independent four-way SMPs, in addition of course to running the entire machine as one 16-way ccNUMA system.

Each SMP starts independently and proceeds until it finishes most BIOS start-up functions. At that point, the BIOS stops and prompts the user to select a partitioning scheme from the configurations described above. After the selection, the various partitions proceed to boot their

respective operating systems. It is important to note that it is possible to reboot one partition or more without affecting the others. Thus, this solution provides a reasonable degree of isolation between partitions.

In retrospect, the advantage of this scheme was its simplicity and practicality. On the other hand, it requires each SMP node to have its own console, and thus negates the "single-system image" advantage of the ccNUMA system. One may argue, however, that a single-system image is incompatible with the notion of running multiple partitions.

Our experience in this regard revealed a serious problem. Windows NT identifies file systems by drive letters. This proved very cumbersome, because the drive letters must be managed so that they appear identical over all of the various configurations. For example, a disk on SMP 2 may be drive "C" for the configuration that contains only this SMP node as an independent partition. This same disk will have a different letter in a configuration where the entire system forms a 16-way partition. Given that each SMP can belong to different partitions under different configurations, some disks will have to be accessible to applications under different drive letters without careful management. Unfortunately, Windows applications do not handle this situation very well. Worse yet, Windows NT insists that there be a drive "C" to store the operating system and boot information. The situation is even more complicated if one attempts to do software striping or exploit the logical volume features of NT over disks that belong to multiple SMPs.

Our conclusion is that Windows NT is not the appropriate platform for supporting partitioned ccNUMA systems, because of its legacy requirement of using drive letters to represent file systems, and its inability to mount these file systems in a portable manner as in UNIX**. The problem seems to be fundamental, and it suggests either using UNIX variants when partitioning is required, or expending considerable resources in managing the drives in an NT system and limiting the possible configurations.

Dynamic partitioning

Dynamic partitioning is more attractive than static partitioning, since the boundaries for the partitions can be changed while the system is running without rebooting any of the partitions. For example, consider the banking example that we described before. The bank may wish to devote a large portion of its resources to serving user requests at times when demand is high, at the expense of the resources devoted to running the database. At night, it may be useful to move some of these resources back to the database partition when user demand is low. Thus, the additional resources can enable the database to provide more resources for nightly system audits and backups. In the morning, the resources can be shifted in the opposite direction without shutting down either partition or affecting the availability of the services.

We have selected Linux as the operating system to support dynamic partitioning. This decision was motivated by the availability of the operating system source code, and our earlier experience with Windows NT. It was also necessary to have access to the source code to enable the partition boundaries to move at run time without shutting down the affected operating systems. To simplify the implementation, the partition boundaries are still along the SMP node boundaries, as described in the preceding subsection. Thus, one can move one SMP at a time between configurations. Adding a node requires modifying the cache-coherency control registers within the mesh coherence unit (MCU) to include the added node in the coherence domain of the receiving configuration. The next step is to search the memory of the added node for the configuration tables that identify its associated resources (CPUs, memory, I/O buses). The kernel data structures of the receiving configuration are then modified to reflect the additional resources, while the CPUs in the added node run through the boot phase, during which the local timers of the added CPUs are synchronized with those of the receiving nodes. The added CPUs then enter the scheduling loop to begin executing application processes.

When a node is deconfigured, we first eliminate its CPUs from the affinity masks of all existing processes and force a reschedule. This step is necessary to prevent an application process from being scheduled on a CPU that will no longer be available to the system. The next step is to disassociate the memory to be deconfigured from the coherency domain of the existing configuration. Thus, future accesses to the deconfigured memory will not interfere with the existing configuration.

Some restrictions to the above scheme are necessary because of the Linux operating system. Deconfiguring the memory of a node requires that all usage of the memory on that node be eliminated. Since the Linux operating system requires zero-based addressing of the physical memory, and it places many of its kernel's data structures in the low-address range, removing the "first" node is very difficult. Removing a different node from the memory point of view is accomplished by first ensuring that no further memory is allocated from that node. Then all process page tables are scanned to identify whether any of the mapped pages fall into the memory range to be deconfigured. If so, the relevant pages are unmapped, and their contents are copied into pages that will continue to belong with the existing partition. The newly allocated pages are mapped to replace the deconfigured pages. If a new page cannot be allocated at that point because of memory pressure, the target page is paged out. Removing memory would be substantially more difficult to accomplish if it were to reside in the first node. Since such memory

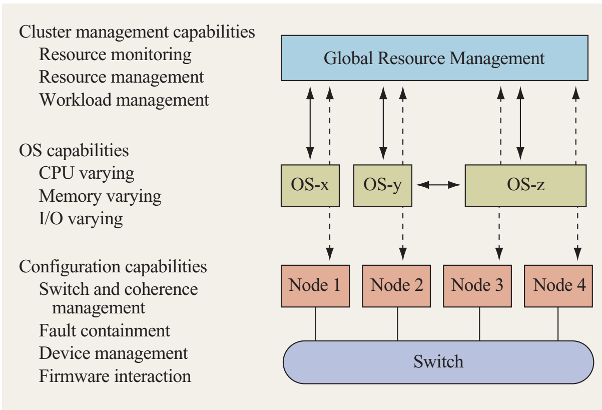

Figure 4

Architecture for global resource migration.

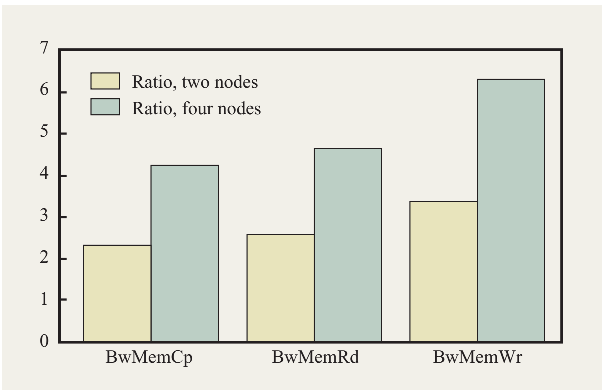

Figure 5

Local vs. remote memory access for two configurations.

can be assigned anywhere in the kernel (e.g., device-driver usage, internal data structures for process table), one would need to track the usage of each page and require a callback for the memory to be freed, which at this point is not provided in Linux.

The overall dynamic resource-migration capabilities of an operating system are an integral part of an overall system, where a global resource manager (GRM) monitors resource utilization and determines how resources should be assigned to the various partitions (dependent on goals set forth for the entire cluster) and when resources must be migrated. **Figure 4** shows this architecture.

6. Performance evaluation

Our performance evaluation consists of three parts. First, we report the component costs of simple operations using synthetic tests. The results of these tests provide a foundation for understanding the results of more complex applications. Second, we report on the performance of several applications from the Splash-II benchmark [13], a computationally intensive benchmark widely used to test the scalability of shared-memory systems. We used the Opium board to tune applications, resulting in generally good scaling. Finally, we report on the performance of Web-oriented benchmarks. This result shows that the poor local-memory access performance of Intel-based SMPs limits the performance of larger ccNUMA configurations.

Synthetic tests

The synthetic tests measure the ratios of local to remote memory access latencies and bandwidth. We ported and instrumented several benchmark programs from the Hbench-OS and Lmbench suites [14, 15], using native Win32 calls to replace C library routines on the paths being measured. Additionally, we extended the benchmarks by adding control for memory and thread affinities and reporting on the performance of memory accesses under different NUMA configurations.

Local vs. remote memory latency

We used three single-threaded C applications to measure memory latency:

- *BwMemCp* uses the Win32 MemoryCopy( ) call to copy one area of memory to another.
- *BwMemRd* reads and sums an area of memory.
- *BwMemWr* writes to an area of memory.

The programs were compiled with Microsoft Visual Studio** 6.0 (SP3), with all optimizations available to the "release" configuration. Each program was executed in two modes. In the local mode, the thread is bound to a specific processor, and all data structures are allocated using the local-memory allocation policy. The remote mode uses the remote-memory allocation policy. We tested two different NUMA configurations. The four-node configuration uses all four nodes and the switch. The two-node configuration uses two nodes with a direct connection between their MCUs.

Figure 5 shows the ratios of remote to local-memory access times (determined using the Intel processor performance counters) for the three benchmark programs. In the two-node system with a direct connection between the MCUs, remote-memory accesses take two to three times longer to complete than local accesses. Introducing the switch to scale the system to 16 processors almost doubles the latency of remote-memory accesses. In the worst case that we measured, a factor of 6 slowdown was observed. These results quantify the penalty for careless or uncontrolled memory placement in our architecture,

and confirm the necessity of software tuning in the absence of a remote-memory cache.

Local-memory bandwidth

During our measurements, we noticed several anomalies in which some programs would perform better if executed with a small number of threads (4 to 8) spread over the NUMA system, than they did on an SMP [16]. We report on some memory bandwidth tests on the basic SMP to quantify the performance of the basic nodes. For this purpose, we used multithreaded versions of the programs described in the preceding subsection: *BwMTMemCp*, *BwMTMemRd*, and *BwMTMemWr*. All programs were run on a single SMP node, with versions that used one, two, and four threads with the threads bound to different processors. We measured the rate at which each version was able to perform the various operations, as measured by the Intel performance counters.

Figure 6 shows the results. We measured the number of bytes per second processed by each program and normalized the operation bandwidth with respect to the single-thread case to show a speedup factor. Our measurements show that the performance does not scale with the number of processors on the bus. Contention for the local memory banks is the main reason for this problem. We expect this problem to last for some time because the planned increases in bus and memory bandwidth for SMPs do not match the projected increases in processor speed.

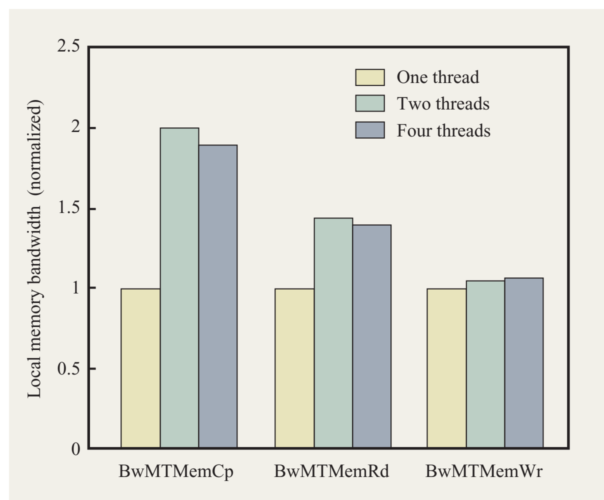

Figure 6

Local memory bandwidth for three programs.

Local vs. remote memory bandwidth

We also used the programs described in the preceding subsection to test the ratios of local versus remote memory bandwidth in order to measure the penalties associated with the additional hardware and coherence protocol in the four-node case. We used the four-thread versions in which the threads are bound to different processors in one node. We tested two memory-allocation configurations: local allocation and remote allocation. **Figure 7** shows the results when the operation bandwidth is normalized to the local-memory allocation policy. The measurements show that for these benchmarks, the operation bandwidth is reduced as much as 40% if memory is allocated remotely. For one program, BwMTMemWr, there was no noticeable difference between the two cases. Now, recall that this program saturates the local memory and does not allow for any scaling within one node. The same behavior dominates in this case, and even when memory is allocated remotely, the bandwidth is effectively dominated by the local bus contention for memory, and for the contention on the local buses of the remote-memory bank. This result shows that the lack of scalability within a node may often mask the performance penalty of remote-memory access. This calls for ccNUMA designs with a smaller number of processors per node and a switch capable of connecting a large number of nodes to obtain good scalability.

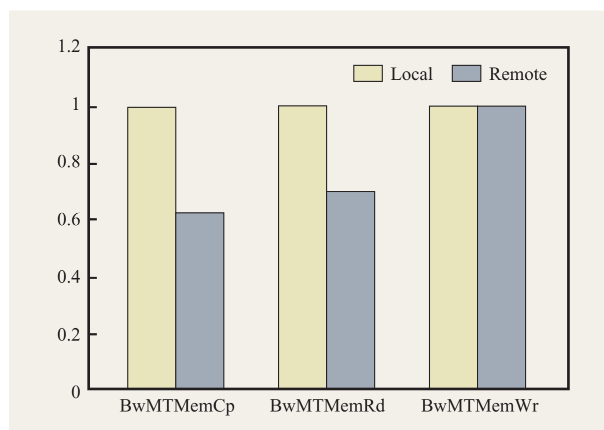

Figure 7

Local versus remote memory bandwidth for three programs.

The Splash-II benchmark

We conducted a detailed study of six applications, five of which are from the Splash-II benchmark suite [17]. The purpose of this study is to understand the effectiveness of the RSet implementation when it is used along with the information provided by the performance monitor card. The methodology examines the application to profile important data structures and uses the Opium card to determine how they are accessed by its threads. The

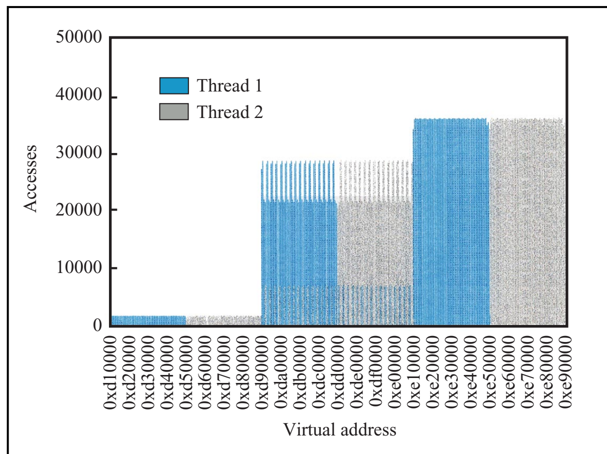

Figure 8

Access patterns for 3DFFT with 4K monitoring granularity.

information is analyzed, and an RSet is then used to enforce affinity policies that locate the threads near the data structures that they use most.

The application suite

The study presented here uses five applications from the Splash-II benchmark and a legacy implementation of the three-dimensional fast Fourier transform (3DFFT). The applications from the Splash-II benchmark are Water-nsquared, Water-spatial, Ocean-contiguous, Barnes-Hut, and FMM (a preliminary version of this study has been published elsewhere [16]). All applications were compiled using the Microsoft Visual C++** 6.0 compiler. A detailed description of the applications follows:

- *3DFFT:* This application implements a three-dimensional FFT. There are several large arrays that are shared by various threads and passed by reference. This program appears to have been written originally in Fortran and converted to C at a later date. Consequently, the sharing patterns and the program logic are obscure.
- *Ocean-contiguous:* Ocean studies large-scale ocean movements based on eddy and boundary currents. The contiguous version implements the grids as three-dimensional arrays. This data structure allows partitions to be allocated contiguously and entirely in the local memory of processors that "own" them, enabling data locality to be enhanced.
- *Water-nsquared:* Water-nsquared simulates molecular dynamics. The threads within this application allocate many small objects that are scattered throughout the address space and are linked with one another through complex data structures.
- *Water-spatial:* Water-spatial also simulates molecular dynamics. While the nsquared implementation uses an $O(n^2)$ algorithm, the spatial version imposes a three-dimensional grid on the problem domain, resulting in an $O(n)$ algorithm that is more efficient for larger numbers of molecules. From a performance-debugging perspective, the two versions use different data structures and are fundamentally different applications.
- *Barnes-Hut:* This application simulates the interaction of a system of bodies in three dimensions using a hierarchical N-body method. The main data structure is an Oct-tree representation of the computational domain. The leaves of the tree contain the bodies, while the internal nodes represent cells in space.
- *FMM:* The fast multipole method (FMM) simulates a system of bodies in three dimensions over a number of time steps. FMM uses the same major data structures as Barnes, but operates on them differently. In particular, the tree representing the computational domain is traversed only once up and down per time step, regardless of the number of bodies, and the accuracy is controlled by how accurately each interaction is modeled.

Performance-debugging methodology

Our generic performance-debugging methodology was to first set up the Opium card to collect information about the entire address space of the application. We programmed Opium to monitor all outgoing accesses from each node and refined this process on the basis of the collected information. We provide additional details on our methodology below.

For applications with array-based data structures (3DFFT and Ocean), we started with a fairly coarse monitoring granularity and then zoomed down to the level of cache lines when required. In general, this is the preferred mode for performance-debugging applications with regular access patterns. For instance, in 3DFFT, we programmed only one Opium filter to match all remote-memory references (read or write) starting from the beginning of the heap, with a granularity of one memory page. **Figure 8** shows the results from the 3DFFT initial phase, with two threads working on the program. (We show the pattern for two threads for simplicity and clarity. The discussion covers the cases of four, eight, and sixteen threads that we measured.) As can be seen in the figure, the data structures are laid out in a uniform manner, even though the program structure did not reveal this pattern easily.

The results suggest that the address range consists of six subranges, where the first and fifth are predominantly accessed by thread 1, and the second and sixth are predominantly accessed by thread 2. The third and fourth ranges are shared by the two threads and contain many

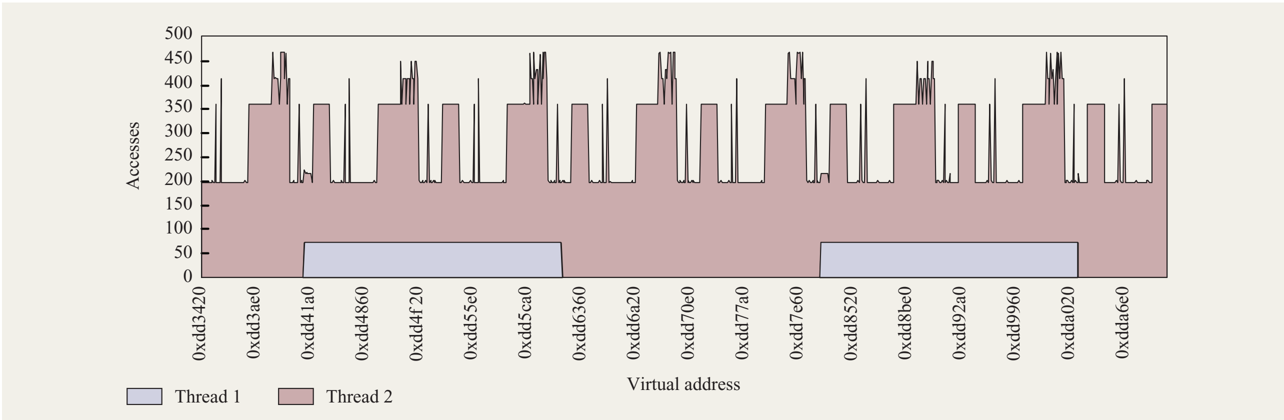

Figure 9

Refined monitoring for 3DFFT at 32-byte cache line size.

hot spots. Following the measurement methodology that we outlined earlier, we then zoomed on the contested area to identify the level of sharing that was taking place.

Figure 9 shows the results of the second phase of measurements, where the granularity was set to a 32-byte cache line size, with focus on the fourth area. The results identify a level of sharing in which thread 2 dominated the accesses. Combining the results of this phase with the initial results, the Opium card created the case for an allocation pattern that matched the application. Thus, instead of modifying the application, we modified the way memory was allocated to match the application's need. The only alternative was to substantially restructure the application, which was undesirable.

We were able to use a similar process to ascertain the data access patterns in Ocean. The contiguous version of Ocean contains a large number of arrays that are accessed by the different threads during execution. The Opium card showed that these data structures were divided among the threads in bands of rows and that interthread sharing occurred at the edges of the bands. With this information, we were able to redistribute the data structures so that each band was allocated with memory local to the node on which the corresponding thread was executing.

For applications with complex data structures (Water-spatial, FMM, Water-nsquared, and Barnes-Hut), our initial performance-debugging attempts revealed that the data structures were being uniformly accessed by all of the threads. To obtain further information, we therefore padded the structure elements to cache-line boundaries, permitting us to correlate the Opium information to specific data-structure elements. This method of performance debugging is well suited to applications with irregular access patterns. We illustrate our approach in greater detail using Water-spatial as an example.

In Water-spatial, the computational threads make a large number of small memory-allocation requests. This pattern results in fragmentation of the virtual (and physical) address space among the various threads in the application. By analyzing the data (not shown here because of space limitations), we quickly found that it was necessary to map the actual virtual addresses to the data structures being monitored in order to determine the best way of optimizing the applications. To do so, for each memory-allocation request, we recorded the calling thread identifier and the location in the program where it was made. This information was then used in addition to the Opium performance counter results to define the identities of the dynamic data structures allocated. The study quickly revealed that most of the water molecules would remain within the zone "assigned" to the thread that created it. Thus, by simply adjusting the memory-allocation policy to use the local allocation (i.e., the data comes from the node on which the calling thread is running), we were able to improve data-access locality. We used a similar technique for Water-nsquared.

In Barnes-Hut, using Opium, we were able to quickly discover a set of arrays that were being accessed by the threads in a partitioned manner, even though all of the arrays were allocated by the master thread during initialization. We partitioned these arrays among the threads, allocating each partition from the memory that was local to the executing thread. We used a similar technique for FMM. For Barnes-Hut, subsequent analysis using Opium revealed that accesses to the main particle data structure were distributed irregularly among the

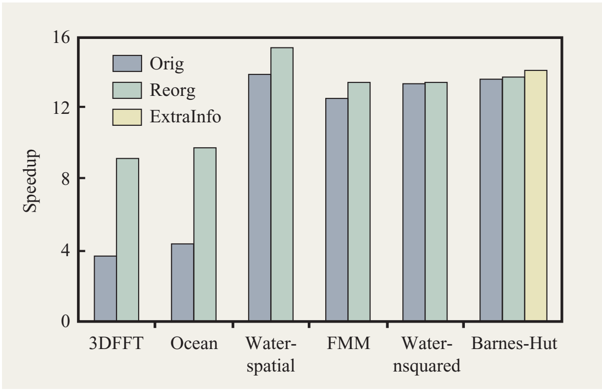

Figure 10

Application speedups before and after data reorganization.

various threads. In order to obtain more information, we padded the data-structure elements so that each element occupied a single cache line. Opium then revealed that the data-structure elements could be divided into two sets—a heavily accessed set and one for which there were substantially fewer accesses. At this point, we were unable to obtain further information without examining the program. However, we examined the program solely with respect to the "important" data-structure elements as identified by Opium. We were then able to ascertain a grouping for the structure elements that minimized the number of cache-line transfers.

Performance impact

In the section we discuss the performance impact of our data restructuring. **Table 1** shows the parameters for each program execution. The program parameters were chosen such that the total datasets would not fit into the L2 caches of all 16 processors.

Figure 10 shows the speedup of each application, before and after the data-structure reorganization derived using Opium. All speedup numbers were measured with respect to a single-threaded execution performed using the local memory-allocation policy. With the exception of Barnes-Hut, there are two bars for each application. The "orig" bar shows the speedup for a 16-thread execution with the striped memory-allocation policy with a stride of one page. With the lack of affinity information, this allocation is reasonable as a base case. The "reorg" bar also shows the speedup for a 16-thread execution, but with the data-structure allocation reorganized as recommended by the Opium analysis. For Barnes-Hut, we show an additional bar called "extra," in which we examined the program source armed with the Opium findings to regroup the program data-structure elements.

Table 1 Program parameters.

Application	*Parameters*	*Sequential execution time* (s)
3DFFT	128 × 128 × 128 array	170.58
Ocean-contiguous	514 × 514 grid	17.44
Water-spatial	32K molecules, 5 time steps	178.96
FMM	64K particles, 10 iterations	52.06
Water-nsquared	9261 particles, 5 time steps	1269.9
Barnes-Hut	128K particles, 4 time steps	160.82

The first point to note about the speedup figure is that for some applications, data restructuring has a radical impact on performance. For 3DFFT and Ocean, performance is more than doubled as a result of allocating memory locally. For FMM and Water-spatial, the performance improvement is much less dramatic, and the improvements are negligible for Barnes-Hut and Water-nsquared.

To explain the variation in performance improvement, we present in **Table 2** the distribution of memory accesses observed during program execution. The first two columns show the number of accesses made by the 16 processors (processor → L1 cache) and that made by all of the L2 caches in the system (L2 cache → memory). The next three columns show the number of request packets, response packets, and total packets in the network interconnecting the nodes. The last column presents the total amount of network traffic in bytes per second of application execution time.[2] This is the average interconnect bandwidth consumed by the application. The "orig" and "reorg" versions show these characteristics for the original and the reorganized version of the application. As discussed earlier, Barnes alone has an "extra" row.

The number of accesses made by the processors varies among the different versions of the applications. We implemented synchronization operations through user-level spin locks. The NT-provided synchronization constructs are highly inefficient and also make it difficult to understand application performance. Due to the suboptimal data-structure layout for the "orig" versions, the processors reach the barrier synchronization points in the program at different times. This causes the early processors that reach a barrier first to execute substantially more spin operations than the later ones. In the "reorg" versions of the application, the processors reach barriers at more or less the same time, leading to a small number of barrier spin operations. Spin-waiting at the barriers is particularly high in the case of Water-nsquared, where some threads take much longer to reach the barriers than others.

[2] Request packets take up 16 bytes, while response packets take up 48 bytes.

Table 2 Execution characteristics for all applications.

Application		*Memory accesses (in 10^6 transactions)*		*Network accesses (in 10^6 packets)*			*Network bandwidth usage* (MB/s)
		P → L1	*L2 → Mem*	*Req*	*Resp*	*Total*	
3DFFT	Orig	41967	408.46	497.89	308.42	806.31	494.11
	Reorg	31867	374.33	76.38	59.2	135.58	216.95
Ocean	Orig	6339	31.73	36.98	23.64	60.62	435.03
	Reorg	6016	28.11	3.58	1.33	4.91	67.85
Water-spatial	Orig	34952	29.96	28.16	20.60	48.77	112.11
	Reorg	35024	17.56	3.83	2.27	6.10	14.61
FMM	Orig	19916	18.96	19.83	13.86	33.69	237.07
	Reorg	19044	18.50	12.27	8.24	20.51	152.59
Water-nsquared	Orig	219836	31.88	26.79	19.50	46.29	14.36
	Reorg	219816	31.95	17.56	13.54	31.09	9.83
Barnes-Hut	Orig	29534	24.75	26.62	18.19	44.80	109.32
	Reorg	29488	24.92	21.25	14.70	35.95	86.90
	Extra	29195	22.40	17.87	12.04	29.92	75.99

Our NUMA system uses a directory-based cache-coherence protocol. In a four-node system such as ours, a memory transaction placed on the bus by a processor can give rise to multiple network packets. For example, when a memory location is invalidated by a processor that is not on its home node, up to five network packets could be generated. This is why, in some cases, the number of network packets exceeds the number of memory transactions placed on the buses by the 16 processors.

For all applications, the “reorg” version sends and receives fewer network packets than the “orig” version, illustrating the usefulness of the Opium performance monitor information. For 3DFFT, Ocean, and Water-spatial, the reduction is dramatic. The reduction is more modest for FMM and Barnes-Hut.

Note that a reduction in network traffic does not by itself result in a lower execution time. For 3DFFT and Ocean, the substantial network traffic reductions are matched by corresponding reductions in execution time for the “reorg” version. However, this is not the case for Water-spatial. We need to examine two factors in order to understand this behavior. First, we need to examine the impact of network traffic on the execution time; second, we need to examine the computation-to-communication ratio. It is reasonable to expect that an application with a small ratio of network traffic to memory transactions will not be much affected by a reduction in the (already small) network traffic component. Similarly, it is reasonable to expect that an application that performs a significant amount of computation per network packet will improve only modestly in performance when network traffic is reduced.

Table 3 illustrates these factors for the applications under study. The first column shows the number of network packets per memory-bus transaction. This metric captures the first factor outlined above. The closer this number is to zero for an original application, the less performance improvement we can expect from reducing network traffic. The next column shows the MFLOPs executed per second (as opposed to retired per second) by the application. The last two columns show the computation-to-communication ratio for each application. Since all of our applications are floating-point intensive, we use a FLOP as the basic unit of computation. We measured these quantities using the processor performance counters. Since the Pentium II processors in our system carry out speculative execution, fewer floating-point operations are retired than are executed by the processor.

Table 3 clearly explains the modest performance improvement for Water-spatial in spite of the dramatic reduction in network traffic. While the reduction in network traffic is several factors more than that for 3DFFT, and the number of network packets per memory transaction is comparable, Water-spatial executes substantially more floating-point operations per network packet and memory transaction. Thus, even though we were able to reduce the network traffic using the Opium performance monitor, the bottleneck for this application is the computation overhead, not the communication overhead. The lack of any performance improvement for Water-nsquared, FMM, and Barnes-Hut is also explained by the measured data. These applications execute such a large number of floating-point operations per memory transaction (or network packet) that the reduction in network traffic brings out no change in performance.

Table 3 Interconnect vs. memory bus traffic and computation vs. communication ratios.

Application		*Network packets per memory transaction*	*MFLOPs per second*	*MFLOPs per 10^6 memory transactions*	*MFLOPs per 10^6 network packets*
3DFFT	Orig	1.97	9.01	1.07	0.54
	Reorg	0.36	20.65	1.16	3.21
Ocean	Orig	1.91	4.49	0.75	0.39
	Reorg	0.17	7.01	0.84	4.81
Water-spatial	Orig	1.63	52.27	32.10	19.72
	Reorg	0.35	57.64	54.77	157.67
FMM	Orig	1.78	42.72	18.66	10.50
	Reorg	1.11	46.12	19.12	17.25
Water-nsquared	Orig	1.45	55.74	151.41	104.27
	Reorg	0.97	55.86	151.07	155.25
Barnes-Hut	Orig	1.81	19.96	11.37	6.28
	Reorg	1.44	20.17	11.29	7.83
	Extra	1.34	20.69	12.56	9.41

Effects of local-memory contention on application performance

We ran several applications with a limited number of threads (four) using different configurations. In one configuration, all threads are placed in the same node. In a second configuration, the threads are distributed over two nodes, such that each node has two threads. And in the third configuration, the threads are distributed over four nodes, such that each node has one thread running. For some applications, the local-memory bandwidth limited the performance of the versions where all threads were colocated in the same node. A detailed study of the experiments is available elsewhere [16]. The general conclusion that we drew from this study is that when the applications are modified to intelligently use RSets and the information collected from the performance card, remote-memory accesses on the ccNUMA configurations have less of an effect. Therefore, the individual threads benefit from having less contention for local memory in configurations where there are one or two threads per node. This result confirms our earlier conclusion about the need for having a smaller number of processors per node, with more nodes connected over the interconnecting switch.

Web-serving performance

We used an industry-standard benchmark to measure the performance of the ccNUMA prototype when it is operating as a Web server. The benchmark consists of a mix of static and dynamic data requests in a 7 to 3 ratio. During the tests, each SMP node was equipped with four 100Mb/s Ethernet network cards and four local 9GB SCSI disks. We used the software striping facility of Windows NT to provide better bandwidth for the benchmark logging. Logging was performed on a file system that was striped over two disks, while the benchmark data were stored in a file system that was striped over the other two disks. Throughout the test, the network load was observed, and in all measurements, it never reached saturation nor was the performance network bound.

The experimental setup consists of a 100Mb/s switched Ethernet network with 48 workstations serving as the clients for the purpose of the benchmark. Each client was equipped with a 400-MHz Intel Pentium II processor and 128 MB of main memory. All clients ran Windows NT 4.0, the Terminal Server Edition with Service Pack 3, build 1381. The Web-server benchmark used the Internet Information Server, Microsoft's built-in Web server. We incorporated all optimizations into the system's registry, as recommended by Microsoft.

Figure 11 shows the performance of the Web-serving benchmark, measured in number of HTTP connections per second. The results are shown for four points, representing one node (SMP basic performance), two nodes (eight processors and 2 GB of RAM only), three nodes (12 processors and 3 GB of RAM), and four nodes (the fully configured NUMA system with 16 processors and 4 GB of RAM). The results show that the scalability of the system with respect to the Web benchmark is adequate but not compelling. In particular, increasing the number of processors from 8 to 12 results in an increase of only 100 connections, and a similar increase in performance occurs when the number of processors rises from 12 to 16. We then analyzed the performance by running the same benchmark with different configurations in order to understand the causes of these problems.

Figure 12 shows the results of running the benchmark for the basic SMP but with different numbers of

processors. The results show that the performance of the basic SMP does not scale, and we attribute this problem to the poor local-memory bandwidth and the resulting contention that occurs when the number of processors increases on the same local bus. As can be seen from the figure, the basic configuration scales only from one to two processors. This is similar to the results from the synthetic benchmarks showing the scalability of the memory bandwidth of the basic architecture.

To confirm that the performance is limited by local-memory bandwidth, we conducted an experiment in which the number of processors per node is reduced. As expected, the reduction in local-memory contention improved performance for the same number of processors. That is, a ccNUMA architecture containing one processor per node (for a total of four processors) outperforms an SMP, despite having processors sharing data across the ccNUMA switch. We ran similar experiments for two ccNUMA systems containing eight processors, with all eight processors coming from two nodes in one configuration and from four nodes in the other. The results confirm the superiority of the configuration with fewer processors per node, despite the fact that more remote memory access occurs. These results are in line with those of the synthetic benchmarks and the Splash-II benchmarks. They certainly suggest a fresh look at how Intel-based ccNUMA systems should be constructed. Note that throughout the performance evaluation for the Web workloads, the main memory was never saturated, and in fact it never reached more than 300 MB per node in any configuration.

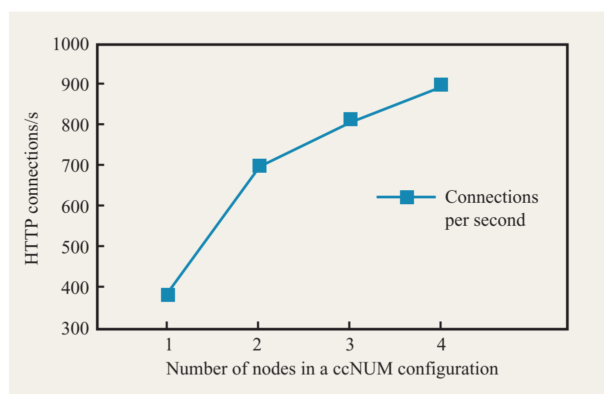

Figure 11

Web performance in HTTP connections per second.

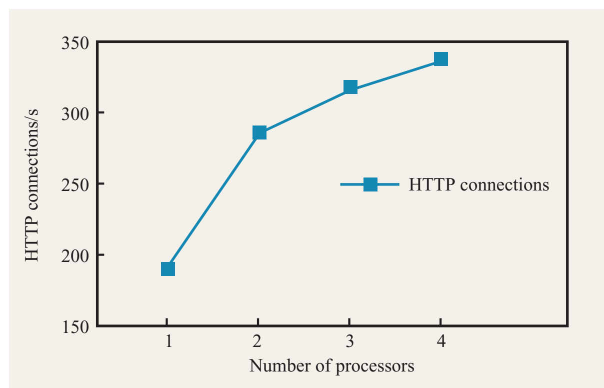

Figure 12

Web performance of the basic SMP.

7. Related work

Commercial interest in ccNUMA increased with the demonstration of the practicality of cache coherence through an interconnect, and the performance limitations of SMP systems. Several research projects contributed to this interest, including the Stanford DASH [18] and FLASH [19] machines, the MIT Alewife [20] machine, and the University of Toronto NUMAchine [21]. Chapter 8 of the famous text on computer architecture [3] by Hennessy and Patterson provides a general discussion of cache-coherent architecture, while the book by Pfister [1] is a survey of a variety of architectures including NUMA and various types of clusters. The recent text by Culler et al. [22] offers the most detailed textbook coverage of the topic of which we are aware, including not only the hardware and software aspects of SMP and ccNUMA systems but also the performance-evaluation techniques that are applied to them.

The research efforts have been accompanied by extensive commercialization of ccNUMA machines, including systems such as the Silicon Graphics Origin** [7], the IBM NUMA-Q* [5], the Data General Aviion** 2500 and NUMALiiNE** [4], HP nuSMP series, and the Unisys ES7000 Series [8]. Our hardware prototype is based on an extension of the Fujitsu Synfinity interconnect used in Fujitsu's Teamserver [9]. It is worth observing that our work began prior to IBM's acquisition of Sequent and, unlike the Sequent product line, is based on the extension of standard high-volume server hardware rather than the engineering of a NUMA-specific system.

Our hardware includes the Opium card in each node of our prototype. At a high level the measurement facilities of the Opium are similar to those of the Princeton SurfBoard, although the overall environments are quite different. In particular, the Princeton board was developed for the SHRIMP project, which implements an abstraction of distributed shared memory over a cluster system rather than a ccNUMA system [23, 24]. However, both share the overall goal of understanding the communication pattern

between the nodes of the system using data gathered during program execution by a noninvasive hardware probe.

Unlike most studies of ccNUMA machines, we use Windows NT 4 as one of the operating systems, and indeed, as the primary focus of our work. While UNIX is still the most commonly studied operating system on such machines, a number of recent systems, including the Sequent NUMA-Q and the Fujitsu system that our hardware extends, support Windows NT. However, Windows NT lacks some of the affinity support that we find to be necessary for good application scalability on a ccNUMA, and our work demonstrates the general need for affinity scheduling for processes and memory.

The programs that we study and whose performance we attempt to improve are taken from the Splash-II parallel processing benchmark [17], which is commonly used in both experimental and simulation studies. However, these are by no means the only types of applications of interest, and there have recently been a number of studies undertaken that use more commercially oriented benchmarks [25, 26] such as those from the Transaction Processing Performance Council [27].

This study uses the data collected using our instrumentation hardware to improve the overall locality of the programs being run through optimization of the source code. In this, it parallels some of the research on profile-based optimization. Our work relies on manually using the data to improve the source code of the application, in contrast to the Morph environment [28].

The synthetic benchmark study reported here uses a subset of a suite of programs developed as a derivative of the Hbench-OS suite of programs from Harvard, which were used in a decompositional study of system performance by Aaron Brown [29]. These, in turn, were based on a predecessor suite of programs, Lmbench [15]. However, all of the previous work was done on UNIX or UNIX-like systems, and all of it was restricted to single-stream and single-threaded tests. Our work includes a port to the native Win32 interfaces of Windows NT, the use of our affinity support, and the introduction of multi-stream tests to measure the effects of contention.

8. Conclusions

We have described our experience with building an Intel-based ccNUMA system using commodity hardware and software components, with a small add-on to scale a system that was built for an eight-way configuration to a 16-way ccNUMA one. Our results point out that the local-memory bandwidth often limits the scalability in performance; one would expect the effects of remote-memory accesses to dominate. According to our study, a good design point for future systems would be to reduce the number of processors connected directly by a bus within a node, but to increase the number of nodes connected by the switch.

Our design included an innovative hardware–software approach for application tuning, based on a hardware assist in the form of a performance-monitoring card, and a software library implementing a new abstraction called the RSet. The card provides a nonintrusive means for collecting performance data about applications, and we have shown that its use has been successful in optimizing several applications. The RSet abstraction, on the other hand, allows the programmer to specify affinity and express control over resource allocation in a hardware-independent manner. The implementation of the concept on top of Windows NT used a device-driver interface along with concealing the true amount of memory from the system. This combination allowed applications to directly control the data placement without having to modify the operating system source code. We have also implemented basic support for partitioning, both static and dynamic, and ported two other operating systems to the hardware.

In retrospect, the simplicity of the design and building on top of existing hardware and software components allowed a team of relatively small size to conduct the work described in this paper for a period of about two years. The resulting performance was not ideal, but it was more than adequate for several applications, including commercially oriented Web serving. Ultimately, such machines are desirable for the ease of system management, workload consolidation and isolation, and the ability to partition the hardware to fit installation needs. We believe that the approach used in this experiment should be considered as a serious option for situations in which time to market and costs are of paramount importance and performance requirements are not too stringent. This provides an interesting design point with respect to price–performance tradeoffs.

Acknowledgments

We are indebted to many individuals for their support and encouragement, including David Bradley, Steven Depp, Patricia Genovese, Peter Hortensius, Thomas Jeremiah, and Joseph McGovern from the IBM Personal Systems Institute; and Rick Harper, Richard Oehler, Basil Smith, and Marc Snir from IBM Research. Tim Biesecker, Willie Jimenez, Marisa Mace, and Bain Weinberger from the Austin Research Laboratory provided technical and administrative support. We also received valuable technical support from Jeff Smith in preparing our Windows NT port. Finally, we would like to thank HAL Computers (a unit of Fujitsu) for the MCU hardware and the excellent support that we received.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Microsoft Corporation, Intel Corporation, The Santa Cruz Operation, Inc., Fujitsu Ltd., The Open Group, Silicon Graphics, Inc., or Data General Corporation.

References

1. G. F. Pfister, *In Search of Clusters*, Second Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1998.
2. D. Solomon, *Inside Windows NT*, Second Edition, Microsoft Press, Redmond, WA, 1998.
3. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, San Francisco, 1996.
4. Data General, "Data General's NUMALiiNE Technology: The Foundation for the AV 25000 Server," *http://www.dg.com/numa*.
5. T. Lovett and R. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," *ISCA '96*, *Proceedings of the 23rd Annual IEEE International Symposium on Computer Architecture*, 1996, pp. 308–317.
6. J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *ISCA '97*, *Proceedings of the 24th Annual IEEE International Symposium on Computer Architecture*, 1997, pp. 241–251.
7. Silicon Graphics Corporation, "Origin 3000," *http://www.sgi.com/origin/3000*.
8. UNISYS Corporation, "ES7000 Series," *http://www.unisys.com/hw/servers/enterprise/7000/default.asp*.
9. 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," *ISCA '97*, *Proceedings of the 24th Annual IEEE International Symposium on Computer Architecture*, 1997, pp. 253–262.
10. J. B. Chen, "The Impact of Software Structure and Policy on CPU and Memory System Performance," Ph.D. Thesis; available as *Technical Report CMU-CS-94-145*, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1994.
11. J. K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud, "Bach: BYU Address Collection Hardware: the Collection of Complete Traces," *Proceedings of the 6th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Edinburgh, Scotland, September 1992, pp. 51–65.
12. B. Verghese, S. Devine, A. Gupta, and M. 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*, 1996, pp. 279–289.
13. J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News* **20,** No. 1, 5–44 (1992).
14. A. B. Brown and M. I. Seltzer, "Operating System Benchmarking in the Wake of lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," *Proceedings of the 1997 ACM SIGMETRICS International Conference on the Measurement and Modeling of Computer Systems*, June 1997.
15. L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," *Proceedings of the 1996 USENIX Technical Conference*, January 1996, pp. 279–294.
16. 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," *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999, pp. 61–72.
17. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *ISCA '95*, *Proceedings of the 22nd Annual IEEE International Symposium on Computer Architecture*, 1995, pp. 24–36.
18. D. E. Lenoski and W. Weber, *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann Publishers, San Francisco, 1995.
19. 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," *ISCA '94*, *Proceedings of the 21st Annual IEEE International Symposium on Computer Architecture*, 1994, pp. 302–313.
20. 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," *ISCA '95*, *Proceedings of the 22nd Annual IEEE International Symposium on Computer Architecture*, June 1995, pp. 2–13.
21. A. Grbic, S. Brown, S. Caranci, R. Grindley, M. Gusat, G. Lemieux, K. Loveless, N. Manjikian, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic, "Design and Implementation of the NUMAchine Multiprocessor," *Proceedings of the 35th IEEE Design Automation Conference*, June 1998, pp. 66–69.
22. D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, San Francisco, 1999.
23. Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems," *Oper. Syst. Rev.* **30,** 75–88 (1996).
24. S. C. Karlin, D. W. Clark, and M. Martonosi, "SurfBoard—A Hardware Performance Monitor for SHRIMP," *Technical Report TR-596-99*, Princeton University, Princeton, NJ, 1999.
25. K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker, "Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads," *ISCA '98*, *Proceedings of the 25th Annual IEEE International Symposium on Computer Architecture*, 1998, pp. 15–26.
26. P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, "Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors," *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 307–318.
27. Transaction Processing Performance Council, *http://www.tpc.org*.
28. X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System Support for Automatic Profiling and Optimization," *Oper. Syst. Rev.* **31,** 15–26 (1997).
29. A. B. Brown, "A Decompositional Approach to Computer Systems Performance Evaluation," B.A. Thesis, Harvard University, Cambridge, MA, April 1997.

Received September 11, 2000; accepted for publication April 3, 2001

Bishop C. Brock *IBM Research Division, Austin Research Laboratory, 11400 Burnet Road, Austin, Texas 78758 (bcbrock@us.ibm.com).* Mr. Brock is a Research Staff Member in the Tools and Technology Department of the IBM Austin Research Laboratory. He received an M.S. degree in computer science from the University of Texas at Austin in 1987 and joined IBM in 1997. Mr. Brock has co-authored numerous publications in the areas of automated reasoning, formal verification of hardware, and system design; he has filed several patents related to hardware performance monitoring, security, and power-management techniques for servers and embedded systems.

Gary D. Carpenter *IBM Research Division, Austin Research Laboratory, 11400 Burnet Road, Austin, Texas 78758 (carpentg@us.ibm.com).* Mr. Carpenter is a Research Staff Member at the IBM Austin Research Laboratory. He joined IBM in 1983 after receiving a B.S. degree in electrical engineering from the University of Kentucky. Since joining IBM Mr. Carpenter has worked in the areas of hardware system architecture, system design, and analog circuit and logic design. Currently his focus is research on energy-efficient circuits, processors, and systems.

Eli Chiprout *Intel Corporation, 5000 W. Chandler Boulevard, Chandler, Arizona 85226.* Dr. Chiprout is a Senior Staff CAD Engineer in the Desktop Products Group in Austin, Texas, and Chandler, Arizona. He has worked in the areas of computer-aided design for advanced circuits, numerical algorithms, microprocessor design, and operating systems. Dr. Chiprout has received an IBM Research Recognition Award; he is the author of a book and multiple journal and conference papers.

Mark E. Dean *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (deanm@us.ibm.com).* Dr. Dean is currently Vice President of Systems in IBM Research. He is responsible for the research and application of systems technologies from circuits to operating environments. Dr. Dean is a member of the National Academy of Engineering and an IBM Fellow. His most recent awards include the Black Engineer of the Year Award, the NSBE Distinguished Engineer Award, and induction into the National Inventors' Hall of Fame. He has more than 30 patents or patents pending. Dr. Dean received a B.S.E.E. degree from the University of Tennessee in 1979, an M.S.E.E. degree from Florida Atlantic University in 1982, and a Ph.D. in electrical engineering from Stanford University in 1992.

Philippe L. De Backer *IBM Research Division, Austin Research Laboratory, 11400 Burnet Road, Austin, Texas 78758 (pdb@austin.ibm.com).* Mr. De Backer is a member of the Research Staff of the IBM Austin Research Laboratory. He received an engineering degree at L'Ecole Centrale des Arts et Manufactures de Paris and joined IBM in France. Mr. De Backer has worked on AIX; he joined IBM Research in 2000.

Elmootazbellah N. Elnozahy *IBM Research Division, Austin Research Laboratory, 11400 Burnet Road, Austin, Texas 78758 (mootaz@us.ibm.com).* Dr. Elnozahy received his Ph.D. in computer science from Rice University in 1993. From 1993 to 1997 he served as an Assistant Professor at Carnegie Mellon University. In 1997, he joined the IBM Austin Research Laboratory, where he currently manages the System Software Department. His research interests include distributed computing, fault tolerance, networking, and operating systems. He holds three patents and has published more than 25 papers on distributed systems. His research in fault tolerance and reliable file systems has influenced several industrial products.

Hubertus Franke *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (frankeh@us.ibm.com).* Since 1993 Dr. Franke has been a Research Staff Member at the IBM Thomas J. Watson Research Center, where he currently manages the Enterprise Linux group. He contributed to the IBM SP2 system software and architecture, the K42 operating system, the development of Linux for highly scalable architectures, and the MXT Linux support. Dr. Franke has received multiple IBM Outstanding Technical Achievement and Outstanding Innovation Awards, and he has published more than 50 papers and 13 patents. He received a first-in-class Diplom. in Informatik from the Technical University of Karlsruhe, Germany, in 1987 and a Ph.D. degree in electrical engineering from Vanderbilt University in 1992. His research interests include architectures and operating systems for highly scalable systems, distributed systems, and system security. He is a member of the IEEE.

Mark E. Giampapa *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (giampapa@us.ibm.com).* Mr. Giampapa is a member of the Scalable Parallel Systems Department, where he holds a technical leadership position in the design and development of a parallel operating environment for a massively scalable parallel ultracomputer. After graduating from Columbia University with a bachelor's degree in computer science in 1984, he joined IBM Research to work in the areas of collaborative and distributed processing, and has focused his research on distributed-memory and shared-memory parallel architectures and operating environments. Mr. Giampapa has received three IBM Outstanding Technical Achievement Awards for his work in distributed processing, simulation, and parallel operating systems. He holds ten patents, with several pending, and has published ten papers on the subject.

David Glasco *Newisys, Inc., 11612 Bee Caves Road, Austin, Texas 78738 (david.glasco@newisys.com).* Dr. Glasco received his Ph.D. in electrical engineering from Stanford University in 1994. He has worked in the areas of highly scalable machine architectures, ccNUMA protocol development, microprocessor design, and system architecture. He is the author of multiple conference papers and holds several U.S. patents.

James L. Peterson *IBM Research Division, Austin Research Laboratory, 11400 Burnet Road, Austin, Texas 78758 (peterson@austin.ibm.com).* Dr. Peterson is a member of the Research Staff of the IBM Austin Research Laboratory. He received a Ph.D. from Stanford University in 1974 and then joined the Department of Computer Sciences of the University of Texas at Austin. Dr. Peterson joined IBM in 1989. He has published four books (including the popular *Operating Systems Principles* textbook) and a number of papers on computers, software, algorithms, and systems.

Ram Rajamony *IBM Research Division, Austin Research Laboratory, 11400 Burnet Road, Austin, Texas 78758 (rajamony@us.ibm.com).* Dr. Rajamony received his Ph.D. from Rice University in 1998 and his B. Tech. from the Indian Institute of Technology, Madras, in 1989. He is primarily interested in experimental systems, focusing in the areas of server power management, networking, operating systems, and web services. Dr. Rajamony has published more than ten papers at venues such as USENIX, ISCA, and SIGMETRICS. According to Citeseer, his publications have been cited more than 300 times. He also received the Best Student Paper Award at SIGMETRICS in 1997.

Rajan Ravindran *IBM Global Services India Pvt. Limited (rajancr@us.ibm.com).* Mr. Ravindran is a Software Engineer; he has been with IBM since 1998. He did his postgraduate work in computer science at Madurai Kamaraj University, India. His work has ranged from product evaluation to implementation.

Freeman L. Rawson *IBM Research Division, Austin Research Laboratory, 11400 Burnet Road, Austin, Texas 78758 (frawson@us.ibm.com).* Mr. Rawson joined IBM in 1973 and spent many years working on the development of operating systems and related software in San Jose, Boca Raton, and Austin before joining the Austin Research Laboratory in 1996. His interests are in systems architecture and software, systems management, and Internet-related technologies.

Ronald L. Rockhold *WhisperWire, Inc., 8240 N. MoPac Expressway, Suite 200, Austin, Texas 78759 (ron.rockhold@whisperwire.com).* Dr. Rockhold joined WhisperWire in 2000 as a Senior Architect. He was previously a Research Staff Member in the IBM Research Division in Austin, Texas, where he focused on operating system issues for NUMA systems. Dr. Rockhold is also an Adjunct Professor at the University of Texas at Austin, currently teaching courses in object-oriented programming and design. He received his Ph.D. in computer science from the Florida Institute of Technology in 1993.

Juan Rubio *The University of Texas at Austin, Department of Electrical and Computer Engineering, ENS 143, Austin, Texas 78712 (jrubio@ece.utexas.edu).* Mr. Rubio received his B.S. degree in electrical engineering from the Universidad Santa Maria, La Antigua, Panama, in 1997, and his M.S. degree in computer engineering from the University of Texas at Austin in 1999. He is currently a Ph.D. candidate at the University of Texas at Austin, where he is investigating hardware and software architectures to improve the performance of transaction-processing workloads. His research interests include the design of microprocessors and memory systems for high-performance concurrent architectures.