The ccNUMA Memory Profiler

Alan L. Davis <ald@cs.utah.edu>
Uroš Prestor <uros@cs.utah.edu>

September 5, 2001

Abstract

This is abstract...

1 Introduction

Among scalable multiprocessor systems, cache-coherent nonuniform memory access (ccNUMA) designs are becoming increasingly popular. Compared to the traditional shared-bus (SMP) systems, they scale to much higher processor counts because their scalability is not limited by a single shared resource. Compared to other scalable multiprocessor designs, they are much easier to program. Like SMP systems, ccNUMA systems implement a globally coherent, shared address space in hardware. The applications written for SMP systems do not require any changes in order to execute on ccNUMA systems, an important consideration when existing codes are to be migrated on new architectures.

However, scalability does not come without cost. While the applications written for a SMP system run unmodified on a ccNUMA system, new factors need to be considered when tuning application performance. Just like other scalable multiprocessors, ccNUMA designs replace one shared resource (system bus) with a collection of distributed resources and any one of them can become a performance bottleneck. More importantly, the basic memory performance characteristics (latency and bandwidth) change depending on where the memory is allocated. On SMP systems, the cost of a memory access is constant for all processors and all memory locations in the system. On ccNUMA systems, the cost of a memory access depends on data placement and on the type of the coherency transaction. Even in the most aggressive implementations of a ccNUMA architecture, the cost of a remote memory access is more than twice the cost of a local memory access.

On SMP systems, the performance analysis of parallel programs needs to address (among other things) application cache behavior, load balancing, and the undesired artifacts of the cached, shared address space (e.g., false sharing). On ccNUMA systems, performance analysis also needs to address application memory behavior. While there are ccNUMA systems which offer operating system support for achieving memory locality, the automated features are not always sufficient. Applications typically need to carefuly tune their data placement in order to achieve good performance on ccNUMA architectures. The research on performance analysis for uniprocessor and SMP systems has produced a variety of methods and tools. When analyzing memory performance, these tools may distinguish between a cache hit or a miss, and they may even be aware of the cache hierarchy. However, very few tools focus on application memory behavior, especially in a nonuniform memory access environment. Furthermore, even though the implementations of the ccNUMA architecture have been available commercially for a number of years, there is no comprehensive study that evaluates the system performance in order to find the secondary effects of ccNUMA architectures on application performance.

We have developed a tool for analyzing application memory performance using the information provided by hardware event counters. A memory profiling tool, `snperf`, uses the information provided by the Origin hardware event counters to evaluate application memory behavior. The profiler continuously samples the event counters for the duration of the application execution and stores the samples into a trace file. The post-mortem analysis tool uses the high-precision timestamps in the trace files to correlate events in application threads, the memory system and the network interface. Combined thread, node and network metrics present a picture of the application resource usage, which may reveal potential performance problems.

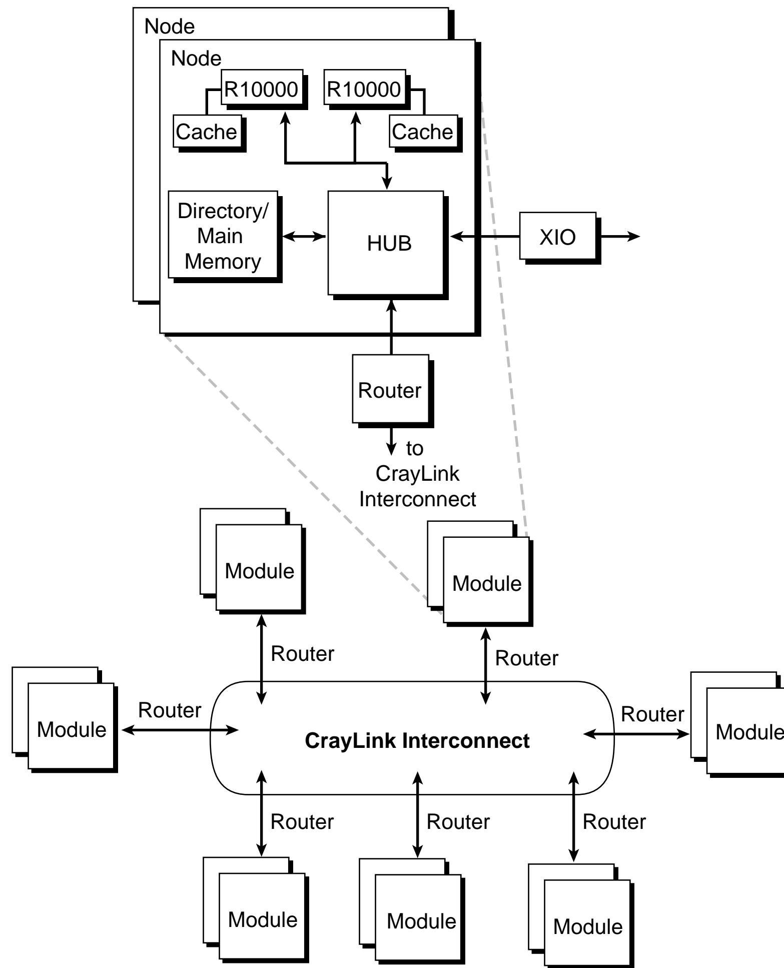

Figure 1: Logical organization of the Origin 2000 system

2 Background

Figure 1 shows the logical organization of an Origin 2000 system. It consists of one or more nodes connected with a fast scalable network. Each node contains two R10000 or R12000 processors, a portion of shared memory, and a directory for cache coherence. The processors are connected to the integrated memory and network controller (the Hub) via a multiplexed system bus (the SysAD bus). The Hub acts as an integrated memory, directory, and network controller and implements a fast I/O port (the XIO interface) connecting the node to the I/O subsystem (the Xbow ASIC). Two nodes access the local I/O subsystem through separate XIO interfaces. The scalable interconnect network is organized as a hypercube. The network is based on a six-ported Router ASIC with two nodes connected to a router and the remaining four ports used for links connected to other Router ASICs. Systems larger than 64 processors are organized as a fat hypercube with individual 32-processor hypercubes connected together with a metarouter.

2.1 Node Board

Figure 2 shows the block diagram of the node board. The processors communicate with the Hub over a shared SysAD bus. Each processor has a separate L2 integrated instruction and data cache. Rather than implementing a snoopy cluster, the two processors use the SysAD bus as a shared resource and do not perform snoops on it. This reduces the latency because the Hub does not have to wait for the result of a snoop before forwarding the request to memory; however, it is not possible to perform local cache-to-cache transfers. There are separate dual inline memory modules

Figure 2: Block diagram of the node board

(DIMMs) for main and directory memory which makes it possible for the Hub memory/directory interface performs data and directory lookups in parallel. The BX ASIC is used to multiplex the memory data path in order to save Hub pins.

The Hub is internally divided into four separate interfaces: memory/directory interface, processor interface, network interface and I/O interface. Internal messages are used to communicate between the Hub interfaces and each interface has a separate input and output queues. The central crossbar connects the four interfaces together.

The memory/directory (MD) interface is an integrated memory and directory controller: it processes memory requests generated by the processors and I/O controllers in the system and maintains directory information for the memory on the node board.

The Hub processor interface (PI) converts CPU transactions issued on the SysAD bus into protocol requests; additionally, it receives protocol intervention and invalidate messages and performs coherence operations in one or both processor's caches.

The Hub network interface (NI) converts internal Hub messages into CrayLink messages and sends them out on the CrayLink network. It receives messages from the CrayLink targeting the local PI, MD and IO interfaces. The network interface also keeps a local portion of the routing tables.

The I/O interface (II) contains the logic for translating protocol messages into the messages used in the XIO protocol. The II also implements two block transfer engines (BTE) which are able to do memory copy at near the peak of the node memory bandwidth.

The nodes in the Origin system are connected to a high-speed network, which is constructed around a set of fast routers in different-sized hypercubes. Two ports on each leaf router are connected to node boards; the other four ports connect router ports, either internally within a module or externally between different modules by means of external cables. Systems up to 64 processors (32 nodes) are organized in pure hypercubes; Systems larger than 64 processors are organized in a fat hypercube where two or more 32-processor hypercubes are connected together with a metarouter. Figure 3 shows the configuration of a 128-processor system. Larger (256- and 512-processor) Origin systems are built from 32-processor hypercubes by organizing the metarouter internally as a hypercube.

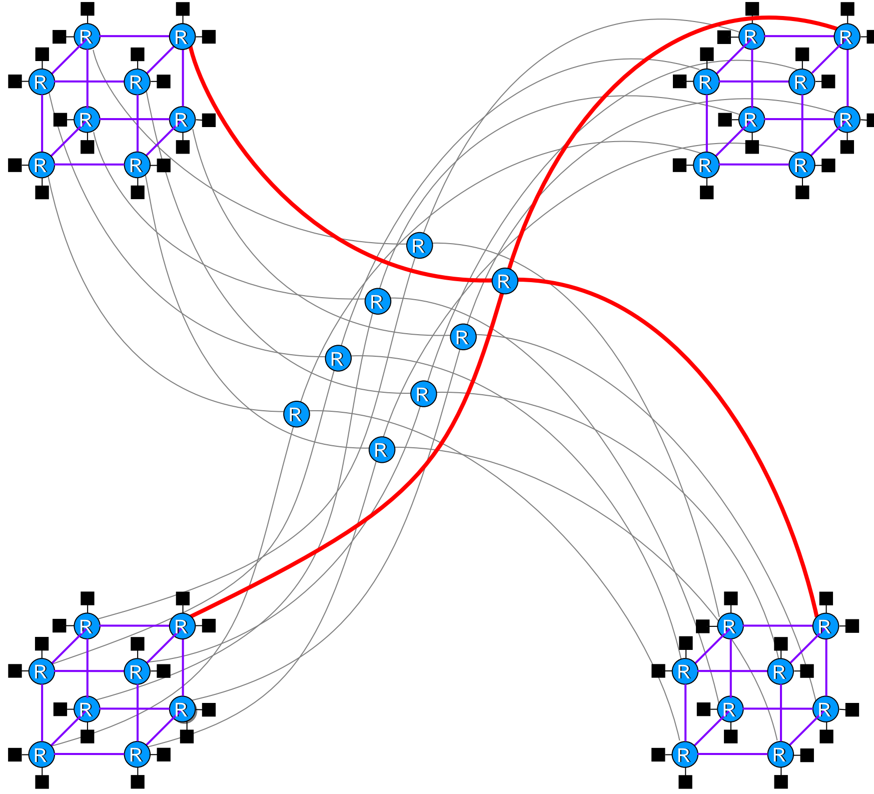

Figure 3: 128 processor Origin 2000 system topology

3 Memory Profiler

3.1 ccNUMA Performance Metrics

A multithreaded application uses a subset of the system resources. The threads execute on processors spread among several nodes. The data can be placed on pages allocated in different memory controllers, and the nodes are connected together with a subset of the interconnect links. Thread and memory placement can be left to the operating system. While the IRIX scheduler does its best to place threads close to the memory they access, the application tuned for the ccNUMA environment should use thread and data placement tools such as `dplace` [11] to pin threads to processors and to specify explicit memory placement. In this way, the application uses the resources on a subset of nodes in the Origin system.

The performance metrics can be divided into two groups: those that apply to each thread of execution and those that apply to shared system resources. The application developer is primarily interested in the behavior of each thread in the application: how it accesses the data items, what transactions it generates, and how it interacts with other threads. At the same time, all threads in the application use shared resources: the memory, interconnect, and I/O bandwidth. If the thread and system metrics are sampled while the application is executing, it is possible to correlate the data after the execution is finished to determine how the system resources were used. Such post-mortem analysis can uncover potential scalability bottlenecks. The accuracy of this approach depends on the “closed world” assumption: the application being profiled must not share the resources with other processes executing in the system. Since it is not possible to distinguish memory requests from different applications without extensive hardware support, the only ways to ensure accuracy are either to use the system in dedicated mode, or to divide the system into batch and interactive partitions and profile the applications submitted to the batch queue.

3.1.1 Thread Metrics

Thread metrics consist of a variety of events and derived quantities that can be associated with a single thread of execution. During its lifetime, a thread can move from one processor to another; while scheduling decisions could impact these indicators of performance, thread metrics do not depend on a specific system resource. The MIPS R10000 event counters are a good example of thread metrics. The operating system maintains a virtualized array of event counts for each thread of execution. Whenever a thread is rescheduled on a different processor, the OS updates virtual counters with event counts from the processor where the thread was executing, and clears event counters on the processor where the thread is about to resume execution. The operating system always returns virtualized event counts; in this way, scheduling decisions become irrelevant.

Several useful metrics help determine memory performance of an application on a ccNUMA system. Even with the aggressive interconnect fabric in the Origin 2000 systems, the latency to the nearest node is almost twice the local latency. The number of memory accesses that go to the local node would ideally be kept as close to the total number of memory references as possible. The *local memory access ratio*, defined as the number memory accesses satisfied locally over the total number of memory accesses is a good indicator of data placement. Beyond the local/remote ratio, one could further break down memory references based on the number of hops between the requestor and the home node.

Just as applications can be characterized by instruction mix (i.e., the number of integer, floating point, memory and branch instructions executed), the ccNUMA applications can be characterized by the *coherence transaction mix*. On the Origin, this mix is the number of unowned, invalidate, clean-exclusive, and dirty-exclusive transactions generated by each thread. Additionally, the invalidate transactions can be further classified by the number of sharers to be invalidated for each invalidating store. The transaction mix, combined with the breakdown of requests by distance to the home node, could be used as an estimate for the *average memory access latency*. Unfortunately, there are other factors that significantly impact transaction latency and bandwidth characteristics. A simple cycle counter embedded in the processor or the Hub processor interface section, triggered by a cache miss or a SysAD request, could yield a much more accurate latency.

A metric that estimates the fraction of the total bus bandwidth used by each thread would be useful for bandwidth-intensive applications. In the Origin system, the R10000 event counters can be used to estimate the secondary cache and SysAD bus traffic. Given the total cache and SysAD bus bandwidths, it is possible to compute the bus utilizations from the number of memory accesses generated by each thread.

The metrics described above are specific to each thread. They focus on the memory performance of the thread. Rather than attributing the metrics to the thread, it would be better if we were able to associate the metrics with the data structures the thread accesses. Linking memory references to data structures necessarily involves hardware and software support. In the absence of features that enable mapping memory requests to data structures, we can use periodic interrupts to record the thread program counter, and map it back into program source code. The data structures accessed at the time the thread was interrupted can usually be deduced from their location in the source code.

3.1.2 Node Metrics

Each node in the Origin system holds two processors, a portion of system memory, an I/O port and a network interface that connects the node to the rest of the system. All four interfaces have balanced bandwidth. The theoretical peak bandwidth of each port is 800 MB/s. The sustained bandwidths are around 570 MB/s for the SysAD interface and more than 620 MB/s for the memory controller. It is hard to measure the network link bandwidth, but it seems that the unidirectional bandwidth of a network link is higher than the total memory bandwidth. To determine potential resource bottlenecks we need to measure the utilization of each interface.

The most important node metric is *memory utilization*. This metric gives an estimate of the fraction of total memory bandwidth used on each node. Memory requests can be generated by either local or remote processors. Measured local and remote bandwidths suggest that the total memory bandwidth of a single node can be achieved just by two threads, one running on a local processor and another on the nearest node. A multithreaded application can be slowed down if several threads access data on a single node. Such imbalance can be spotted easily if we measure memory utilization on all the nodes where the application has allocated memory. If a single node shows high memory utilization while the other nodes are idle, chances are that the hot spot is limiting application performance. This scenario is not uncommon for “naive” multithreaded applications. The IRIX operating system will by default allocate pages on a first-touch policy; if the application initializes its data with a single thread, it is quite possible that all the data pages will be

placed on a single node. When the application spawns worker threads, they will all access data on a single node where the memory bandwidth can become a performance bottleneck.

Similar arguments apply to the network interface that connects the node to the nearest router: *network interface utilization* can be used to determine whether a node network interface is the bottleneck. In practice this seems very unlikely. Craylink ports have unidirectional bandwidth higher than the memory interface. The link could get saturated only if there is a significant amount of I/O activity in addition to the memory traffic.

The SysAD bus in the Origin systems has barely enough bandwidth to support a single processor. When two threads of a bandwidth-intensive application execute on a single node, the SysAD bus does not have enough capacity for both. The *bus utilization* metric gives the estimate of the fraction of the total SysAD bus bandwidth used by both processors on the node. If the bus utilization is sufficiently high, it could be worth placing one thread per node instead of two threads per node.

3.1.3 Network Metrics

The Origin interconnect routes are determined statically at system boot. While the bidirectional links have a higher bandwidth than the memory interfaces, the shared links that connect routers could become performance bottlenecks. The *link utilization* is an indicator of how much traffic is being routed over the link; high utilization rates can lead to performance bottlenecks. Another metric of interest is *link contention*. High contention rates contribute to network latency.

3.2 Implementation

The ccNUMA memory profiler uses the information from the Origin hardware event counters to compute various thread, memory and network metrics. The memory profiler is used in a manner similar to the SpeedShop tools[14, 12]. The application is launched by the profiler, which samples hardware event counters periodically and saves the output in trace files. When the application terminates, the data in trace files are analyzed with a post-mortem analysis tool that uses event counts from the trace file to derive ccNUMA performance metrics. There is no graphical user interface—tools such as `gnuplot` [15] or `jgraph` [10] can be used to visualize the ccNUMA metrics. It is also possible to interactively print node and network metrics by using the system activity monitor.

The existing IRIX interfaces to the hardware event counters are inadequate for accurate application profiling, mostly due to relatively slow update intervals (10 ms for the Hub counters and two seconds for the router utilization counters). Our first task was to design and implement a new, more flexible interface to the Hub and Router counters. The loadable kernel module (LKM) provides a fast and safe way for user programs to access the Hub and Router hardware event counters. The LKM is a simple device driver that establishes a connection between the user program and the hardware registers. It can map the Hub counters into user space, access the Router histogram registers, and perform a number of driver-specific device control calls. Higher-level operations, such as regular sampling, multiplexing, and virtualizing event counters, are implemented in user space.

The decision to implement the minimum amount of support in the kernel driver was based on a couple of observations. First, it is not possible to implement high-frequency sampling rates in the kernel without imposing an unwanted overhead on the rest of the system. The device driver should provide a means of accessing the hardware registers for processes which need it, but should otherwise stay out of the way. In this way, the sampling overhead is paid by the user-level process. Additionally, the sampling process can be pinned for execution on one processor. This has the benefit of minimizing the context-switch overhead (Irix will schedule the timer interrupt on the processor to which the sampling process is bound) and using the resources that are not used by application threads. Second, a sampling interface implemented in the kernel would be obsolete with a new generation of the Origin systems. An obsolete interface would have to stay in the kernel forever because of backwards compatibility.

The application launcher, `snrun`, is used to gather event counter traces for the duration of application execution. The `snrun` program combines the functionality of `snsar` with a shared library called the thread wrapper, which traces program execution and collects data about application scheduling and periodic samples of the R10000 event counters. The results of application profiling are two or more event trace files: one trace file contains the data from the Hub and Router counters, and the other trace files contain event traces for each application thread. The data in these files are viewed and analyzed with post-mortem analysis tools.

The application profiler outputs one or more event trace files. Each trace file contains information about the system where the trace was generated, the objects (threads, nodes, and network ports) whose hardware event counters were

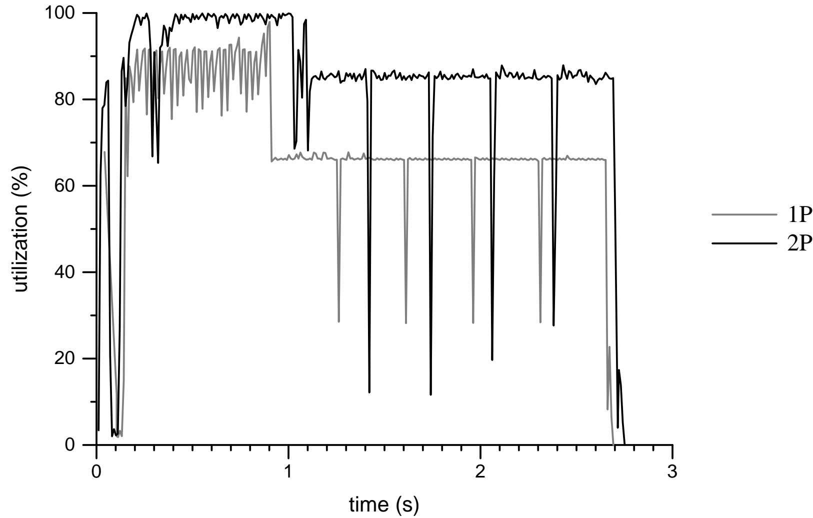

Figure 4: Memory utilization for 1- and 2-thread local reduction loop

being collected, and a collection of samples from each object. Each sample holds a high-precision time stamp, which is used to collate the samples from various trace files. There are two programs which can be used for the post-mortem analysis: the `sninfo` program prints information about trace files, and the `sndump` program prints full contents of the trace files.

4 Examples

4.1 Memory and Link Utilization

The utilization of the Hub memory/directory unit is a good indicator of the node memory pressure, i.e., how many memory requests are being serviced by a particular node. The Origin 2000 Hub ASIC is capable of measuring the number of cycles when the memory unit was idle, blocked, serving memory requests, or refreshing memory. The memory profiler described in Section 3 defines memory utilization as the fraction of time when the memory unit was not idle. Memory utilization computed in this way is not a linear function of the number of memory requests served per time unit. The nonlinear effect is due to the internal organization of the memory controller, which is highly pipelined. When the pipeline is full the memory utilization is 100%; however, when there are not enough requests to saturate the pipeline, the busy time is counted from the moment a request enters the pipeline until the response is sent out.

Figure 4 illustrates how memory utilization does not correspond to the amount of data returned by the memory unit. The two plots show the memory utilization profile of the `snbench` program, which ran five iterations of a simple array reduction loop. In the *1P* case, one thread was placed on the local node. The *2P* case shows the profile when two threads were placed on the local node. In both cases, the reduction loop operated on a unit-stride array of double-precision floating point values. The first part of each plot corresponds to the `snbench` initialization phase, during which the test array is modified and the cache flushed (this is necessary to place the cache lines in unowned state). The second part of the plot shows five iterations of the reduction loop, each of which is delimited by a short drop in utilization during which thread synchronization occurs.

In the single-thread case, `snbench` reported a reduction of 280 MB/s, which corresponds to $\approx 45\%$ of the total memory bandwidth (620 MB/s). However, the measured memory utilization is fixed at 66%. In the double-thread case, `snbench` reported a combined reduction of 412 MB/s, 66% of the total memory bandwidth. Again, the memory utilization rate is much higher—85%. Note how the bandwidth reduction for unit-stride arrays are much lower compared to the results of the `bw-read` experiments, which measure the bandwidth on double arrays with stride 16, and touch only one element in each 128-byte cache line. This reduction in bandwidth is due to the capacity of the L2 cache bus and to the delays in issuing loads because of arithmetic instructions.

Figure 5 compares memory and link utilizations. In this `snbench` experiment, the test memory was placed on

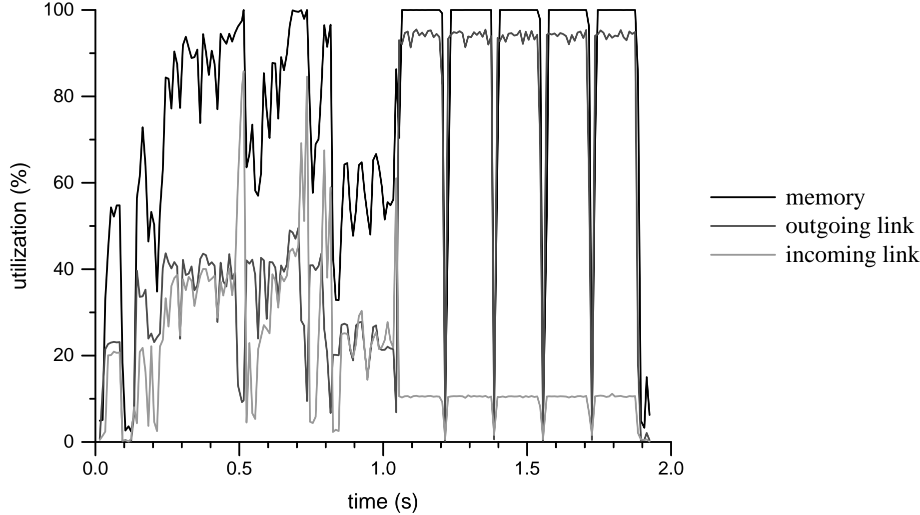

Figure 5: A comparison of memory and link utilizations

one node and three threads were placed on neighboring nodes (one per node to avoid the SysAD bottleneck). The threads executed five iterations of the `bwmp-read` experiment. The memory utilization is firmly pegged at 100%; the experiment reported cumulative memory bandwidth of 623 MB/s. Since no thread was placed on the remote node, all data was sent over the outgoing link. For each iteration of the experiment, the outgoing link utilization is $\approx 95\%$; this suggests a link bandwidth of 655 MB/s. However, the actual sustained link bandwidth is actually higher ($\approx$ 693 MB/s); since the link utilization counters are fairly accurate, we suspect that the difference is due to inaccuracies in `snbench` multithreaded memory bandwidth measurements. At the same time, the incoming link utilization was $\approx 10.5\%$, about one ninth of the outgoing link utilization. This suggests that the link utilization is proportional to the number of packets sent over the link. In each iteration of the experiment, memory requests are being sent to the home node over the incoming link, while the data responses are going out over the outgoing link. Since the experiment places data in the unowned mode, there is no additional traffic besides requests and replies. Each request consists of a single 128-bit packet; the reply consists of a header packet followed by eight data packets, which make up one 128-byte cache line.

4.2 SPLASH-2 FFT

We now turn to a simple application to illustrate how the memory profiler can be used to determine application resource usage and to evaluate the algorithmic trade-offs. We use the FFT kernel from the SPLASH-2 suite [16] to study the application memory requirements and to evaluate three alternatives for a matrix transpose algorithm.

The fast Fourier algorithm used in the SPLASH-2 FFT kernel is a complex 1-D version of the radix-$\sqrt{n}$ six-step algorithm described in [2], which is optimized to minimize interprocessor communication. The six steps in the algorithm are:

1. Transpose the data matrix.
2. Perform a 1-D FFT on each row of the data matrix.
3. Apply the complex roots of unity to the data matrix.
4. Transpose the data matrix.
5. Perform a 1-D FFT on each row of the data matrix.
6. Transpose the data matrix.

There is a barrier before the second and third matrix transpose (steps 4 and 6). The data set for the FFT consists of the n complex data points to be transformed (generated as a set of random values), and another n complex data

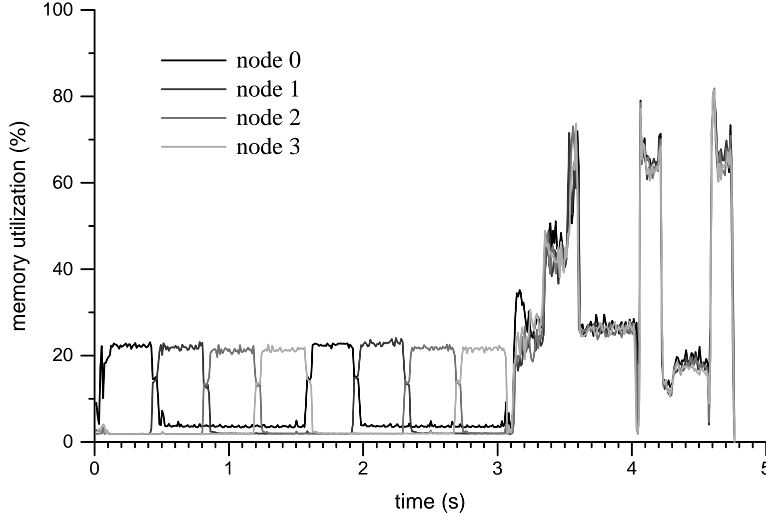

Figure 6: FFT memory utilization profile on four nodes

points referred to as the complex roots of unity. Both sets are organized as $\sqrt{n} \times \sqrt{n}$ matrices, and the matrices are partitioned so that every processor is assigned a continuous set of rows that are allocated in its local memory. Interprocessor communication is limited to the transpose steps.

Figure 6 shows the memory utilization profile on four nodes for the entire duration of a 8-processor FFT run. The first phase of the execution is the initialization of the roots of unity and the generation of random input data. The initialization is performed on the main thread, which runs on processor 0. The memory utilization profiles for each node show that the data set was distributed equally among the four nodes—the plot from start to ≈ 3 sec shows first utilization on node 0, then node 1, node 2 and node 3; the utilization has two peaks on each node, the first time when the application initializes the data for the roots of unity matrix and the second time when it generates the random data.

The second phase shows the progress of the FFT algorithm. The three peaks in the memory utilization plot correspond to the matrix transposes. These are the interprocessor communication phases, where every processor transposes a portion of the data matrix. The two valleys in between correspond to the 1-D FFT transformation on each (local) row and the application of the roots of unity. The barriers before the second and third transpose are visible as the sharp drops in memory utilization.

The transpose algorithm used by the SPLASH-2 FFT kernel works in two phases: first, each processor transposes a patch (contiguous submatrix) of size $\frac{\sqrt{n}}{p} \times \frac{\sqrt{n}}{p}$ from every other processor, and then transposes the patch locally. The transpose takes advantage of long cache lines by blocking. The original SPLASH-2 FFT uses staggering to communicate patches between processors: processor i first transposes a patch from processor $i+1$, then from processor $i+2$, etc., to prevent hotspotting. If the processors move in lockstep, no two processors read from the same processor's patch of memory at the same time. We will call this communication pattern the *basic stagger*. However, there are no barriers inside the SPLASH-2 FFT transpose algorithm. It is entirely possible that one or more processors fall behind the others, because it was preempted by system activity, for example. Since the processors transpose patches in a sequential manner, one delayed processor could cause a domino effect, and further delay other processors that follow it. To avoid this scenario, a second transpose algorithm uses a binary scrambling function to compute the next processor whose patch is to be transposed; this is the *optimized stagger* algorithm. Both staggered transposes are contrasted with the naive matrix transpose where each processor first replaces a patch from processor 0, then processor 1, and so on. This is the *unoptimized transpose* algorithm.

Figures 7–9 show high-resolution memory utilization profiles for unoptimized transpose, basic, and optimized staggering, respectively. All figures show the second transpose step in a 16-processor run for a data set size of 4M elements; each run assigned two threads to each node, allocating memory on 8 nodes. The memory utilization is shown for even-numbered nodes only.

Not surprisingly, the unoptimized transpose algorithm results in memory hotspots: as the processors transpose patches, they first overrun the memory capacity on node 0, then node 1 and so on. The basic stagger eliminates

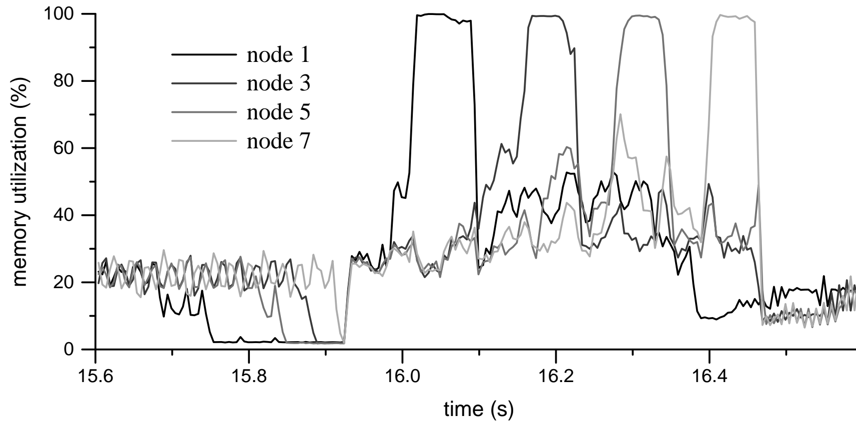

Figure 7: Unoptimized FFT matrix transpose without staggering

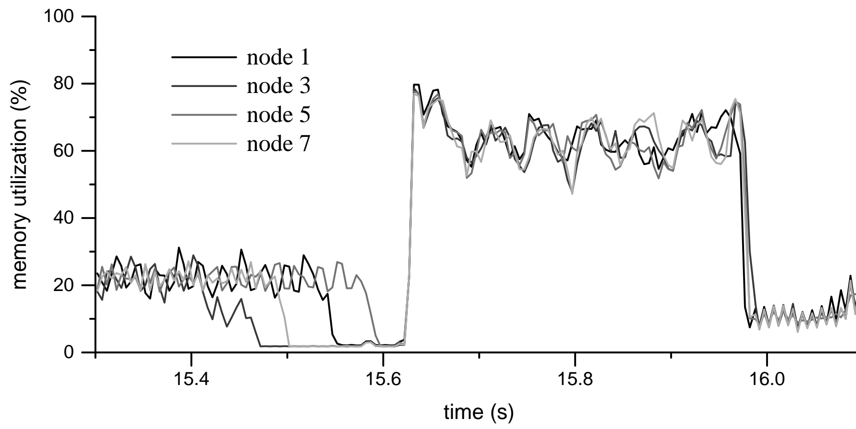

Figure 8: FFT matrix transpose with basic staggering

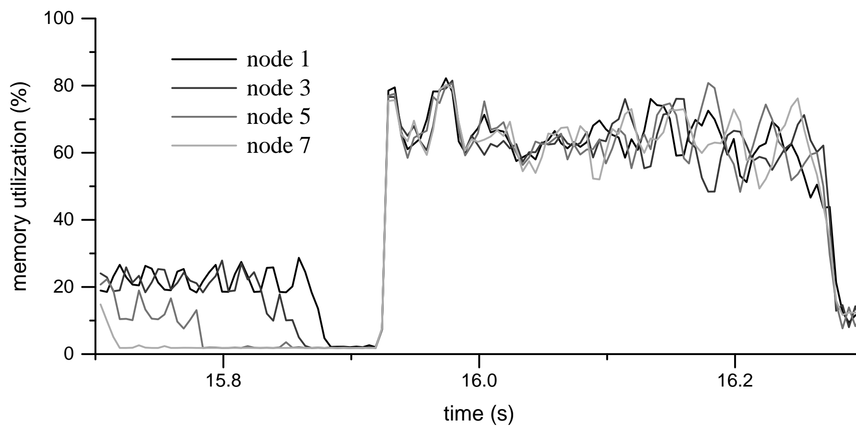

Figure 9: FFT matrix transpose with optimized staggering

memory hotspots: during the transpose phase, the memory on all nodes is utilized evenly. It seems that the basic staggered transpose leaves plenty of memory bandwidth because the node utilization hardly climbs above 80%; even with two processors pulling patches of the data matrix from a single node, the aggregate data rate from each node is comparable to the data rate achieved by a single local processor doing a sum over a unit-stride array (the 1P plot in Figure 4). The FFT transpose does not run at the full memory speed—one reason for this limitation could be the limited bandwidth of the SysAD bus which is shared between two processors doing the transpose. The other reason could be the decrease in remote memory bandwidth when threads access memory on remote nodes.

For runs with a relatively small number of processors, the optimized stagger algorithm does not seem to improve the performance. In the 16-processor case shown in Figure 9, it actually performs slower than the basic staggered transpose. Additionally, the memory load becomes much more uneven, especially at the end of the transpose phase. While the load becomes uneven in the basic stagger as well (most likely because some threads fall behind others), the effects are much more evident in the optimized stagger. It is not clear whether the optimized stagger will perform better at larger processor counts. Instead of simply scrambling the order in which other processor's patches are transposed, a better approach would be to copy patches in a topology-aware ordering. However, this is difficult to implement, because the Irix operating system does not provide detailed routing information to user programs.

Another reason for low memory bandwidth measured in our experiments could be the TLB miss penalty. The data set of 4M complex elements requires 64 MB for two arrays. With a page size of 16 KB which was used in our measurements, the TLB reach is only 2 MB (the R10000 processor has 64 TLB entries, each mapping two consecutive pages). McCalpin reported linear scaling of FFT transpose algorithm on Origin systems up to 64 processors. [8] His algorithm used barriers inside the transpose phase. At 64P, the transpose time was 1.6 times local copy time.

5 Related Work

With the increasing use of superscalar and out-of order instruction scheduling, it is even more difficult to determine how the program spends its time. However, it is increasingly common for the designers of microprocessors to include hardware support for counting various types of events or even profiling instructions during the execution in processor's pipelines. The MIPS R10000 provided a set of on-chip event counters [13], which can be used to count the number of of cache misses, memory coherence operations, branch mispredictions, TLB misses, and to determine instruction mix for both issued and retired instructions (R10000 uses speculative execution along predicted branch paths). Similar hardware performance counters have appeared in other processors: Cray vector processors [4], DEC Alpha [6], HP PA-8000 [7] and Intel Pentium processors.

DEC's Digital Continuous Profiling Infrastructure (DCPI) project [1] used hardware event counters to perform statistical profiling of the entire operating system, including the kernel, shared libraries, and all user-level applications. The data collection was designed for minimal system impact. The DCPI tools provided profile information at varying levels of granularity, from whole images (executables and shared libraries); down to procedures and basic blocks; down to detailed information about individual instructions, including information about dynamic behavior such as cache misses, branch mispredicts, and other stalls. The profiler ran on in-order DEC Alpha processors, which enabled the analysis tools to attribute stalls precisely to each instruction. Precise attribution of instruction stalls is no longer possible on an out-of-order processor. In order to achieve the same level of precision on an out-of-order processor, the DCPI authors designed a new form of hardware support for instruction-level information to be used with the DCPI tools [5]. They proposed an approach where the processor instruction fetch unit selects an instruction in the input stream at random and tags it with a special bit (the *ProfileMe* bit). As a tagged instruction moves through the procesessor pipeline, a detailed record of all interesting events and pipeline stage latencies is collected. This information is made available to the profiling software when the instruction is retired.

With the widespread availability of performance monitoring features in modern microprocessors there is a need to standardize the programming interfaces which are used to access these features. The PerfAPI [9] and PCL [3] projects aim to provide a portable library of functions and a standard set of performance monitoring events to be used by application writers who wish to instrument their codes in a portable way. Both projects support the majority of modern microprocessors and operating systems where the counting of hardware events is possible. They offer different language bindings (C, Fortran, Java) and they define a common set of event definitions; however, not all events may be implemented on all systems, which presents the fundamental problem for portability. The applications that need to be truly portable need to restrict the use of hardware events to the small group which is implemented on all systems (typically the number of CPU cycles and cache misses).

All projects described so far that use hardware event counters look at the application behavior from a processor-centric perspective. They all use performance-monitoring features that are implemented in the processor; while this offers plenty of data about processor-related events, all the information is lost when memory requests leaves the processor. At best, the processor event counters provide the latency of a memory operation and a glimpse into the cache coherence protocol for first- and second-level caches that are typically controlled by the processor. While this information is reflected in the program timings, the performance analyst typically cannot determine application memory behavior, especially in distributed systems with many independent resources (memory controllers, I/O units, network interconnect links) and complex cache coherence protocols. In such systems, distributed resources can become distributed bottlenecks.

Our memory profiling tool attempts to correlate processor events (thread metrics) with events in the memory subsystem on each node (node metrics) and the interconnect network (network metrics). The memory profiler does not use statistical sampling: to facilitate correlation between various sources of performance data the memory profiler continuously stores samples from processor, node and router counters into a set of output files; each sample is timestamped with a high-resolution clock. The post-mortem analysis tools correlate samples from different trace files and present all metrics on an unified timeline.

6 Conclusion

Cache-coherent nonuniform memory system architectures are becoming both economically feasible and commercially available as the basis for scalable multiprocessors. The shared address space programming paradigm, while similar to the conventional programming on symetric multiprocessor systems that use a snoopy bus-based cache coherence protocol, nonetheless introduces subtle differences, which are important for high-performance computing on ccNUMA systems. The nonuniform memory access times, which depend on the distance between the requestor and the home node, require that the user pays attention to data placement, a requirement not present in traditional SMP systems. Scalable multiprocessor systems replace the central bus with a set of distributed resources; while the individual bandwidth of each processor bus, memory port and interconnect link is smaller than its equivalent in a SMP system, the aggregate capacity far exceeds the capacity of the largest shared resource in a conventional SMP system. However, distributed resources introduce a possibility of distributed bottlenecks. Tools are needed which help the users determine application resource usage and detect potential bottlenecks.

Applications that generate significant amounts of memory traffic are very sensitive to the NUMA environment. We have developed a device driver which lets a program access the hardware event counters in various Origin ASICs. A separate program acts as a memory profiler: it samples the event counters and periodically stores them in a trace file. The profiler can use Origin Hub and Router event counters and the R10000/R12000 processor event counters. In this way, an application can be profiled after it finishes execution. The `snperf` memory profiler defines the notion of an object that corresponds to a particular system resource (a node or a network link) or a thread in the application program. In the post-mortem analysis, a number of different metrics can be computed from the event trace files; a metric can be specific to a node, a thread, a network link, or it can be a global metric which is derived from several other metrics. In addition to application profiling, the memory profiler can be used in a standalone mode to print interactive values of the hardware event counters.

We provide several examples to show the use of the memory profiler. First, we used `snbench` bandwidth kernels to evaluate the memory and link utilization metrics. We found that the link bandwidth is slightly higher than the memory bandwidth. We used node memory utilization to profile the execution of the SPLASH-2 FFT kernel. The utilization plot clearly identifies various application phases; we also found that the basic stagger algorithm used in the transpose phase performs well on systems up to 32 processors, while the “naive” implementation without staggering results in memory hot spots.

The `snperf` memory profiler is clearly a research tool. It requires an intimate knowledge of the Origin system architecture, and it lacks a user interface and a graphic display of its results. However, we believe that the fundamental approach is sound. In a system with distributed resources, the ability to determine the utilization of various system resources and to correlate per-thread and system-wide metrics is essential for performance analysis on large-scale systems. While additional hardware support for performance analysis is needed to increase the accuracy of the results, we believe that the data offered by the Origin event counters is a step in the right direction.

7 Acknowledgment

Figures 1, 2, and 3 were taken from the Origin technical manuals. Copyright © 2000 Silicon Graphics, Inc. Used by permission. All rights reserved.

References

[1] ANDERSON, J., BERC, L. M., DEAN, J., GHEMAWAY, S., HENZINGER, M. R., LEUNG, S.-T., SITES, R. L., VANDEVOORDE, M., WALDSPURGER, C. A., AND WEIHL, W. E. Continuous profiling: Where have all the cycles gone? In *ACM Transactions on Computer Systems* (November 1997), pp. 357–390.

[2] BAILEY, D. H. FFTs in external or hierarchical memory. *Journal of Supercomputing 4*, 1 (March 1990), 23–35.

[3] BERRENDORF, R., AND MOHR, B. ”PCL—the performance counter library: A common interface to access hardware performance counters on microprocessors”. `http://www.kfa-juelich.de/zam/PCL`.

[4] CRAY RESEARCH, INC. *UNICOS Performance Utilities Reference Manual*, January 1994. Cray Research Publication SR-2040.

[5] DEAN, J., HICKS, JAMEY WALDSPURGER, C. A., WEIHL, W. E., AND CHRYSOS, G. *ProfileMe*: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture* (December 1997).

[6] DIGITAL EQUIPMENT CORPORATION. *pfm(5) - the 21064 performance counter pseudo device*.

[7] HUNT, D. Advanced performance features of the 64-bit PA-8000. COMPCON’95, `http://www.convex.com/tech_cache/technical.html`, March 1995.

[8] MCCALPIN, J. D. Personal communication.

[9] MUCCI, P. J., KERR, C., BROWNE, S., AND HO, G. ”PerfAPI: Performance data standard and api”. `http://icl.cs.utk.edu/~mucci/pdsa`.

[10] PLANK, J. S. Jgraph — a filter for plotting graphs in PostScript. In *Proceedings of 1993 Winter USENIX* (San Diego, CA, January 25-29, 1993), pp. 63–68.

[11] SGI. `dplace(1)` - a NUMA memory placement tool. Man page.

[12] SGI. `SpeedShop(1)` - an integrated package of performance tools. Man page.

[13] SGI. *MIPS R10000 Microprocessor User’s Manual, Version 2.0*, October 1996.

[14] SGI. *SpeedShop User’s Guide*, April 1999. Document no. 007-3311-006.

[15] WILLIAMS, T., AND KELLEY, C. `gnuplot(1)` - an interactive plotting program. Man page.

[16] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The SPLASH-2 programs: Characterization and methodological considerationss. In *Proceedings of the 22nd International Symposium on Computer Architecture* (June 1995).