Evaluating the Memory Performance of a ccNUMA System

Uroš Prestor

September 5, 2001

Abstract

Scalable cache-coherent nonuniform memory access (ccNUMA) architectures are an important design segment for high-performance scalable multiprocessor systems. In order to write application programs that take advantage of such systems, or port application programs written for symmetric multiprocessor systems with uniform memory access times, it is important to understand the impact of nonuniform memory access times and the associated ccNUMA cache coherence protocols on aggregate application memory performance. This work presents a detailed memory performance analysis of a particular ccNUMA system (the SGI Origin 2000). The thesis presents a new memory profiling tool, called `snperf`, and a new set of microbenchmark codes, called `snbench`, which make such a fine-grained memory performance analysis possible. The analysis was performed on a wide variety of Origin 2000 system configurations and demonstrates that memory locality has a strong impact on application performance. More importantly, the results demonstrate a variety of second-order memory performance effects that are also substantial performance influences. Even though the specific implementation target for this thesis was the Origin 2000 architecture, the methods are applicable to other ccNUMA systems.

Contents

List of Figures

List of Tables

Chapter 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.

The SGI Origin 2000 system is an aggressive implementation of a ccNUMA architecture. The system scales from two to 512 processors, the operating system includes extensive support for NUMA programming and there is a rich set of tools for parallel program development. The system includes hardware support for performance analysis: both the processor and various system ASICs include a set of hardware event counters that can be used for application profiling. SGI offers a set of performance analysis tools that use processor event counters for application profiling [43, 51]. Even though the hardware event counters in the Hub and Router ASICs offer valuable information about the application memory behavior, the directory cache coherence transactions, and the interconnect network traffic, there are no tools that use this information to aid in performance analysis of parallel programs on the Origin. Furthermore, very little information exists about the memory system performance of large Origin systems. The existing publications [19, 50] focus on small systems, and they do not evaluate the impact of the directory protocol on memory performance.

The contribution of this thesis is twofold. First, it gives a detailed analysis of the memory performance on the Origin 2000, evaluating the system architecture, the directory cache coherence protocol, and the trade-offs that influenced their design. Second, it presents a way of analyzing application memory performance using the information provided by the Origin hardware event counters. A suite of microbenchmarks, `snbench`, was written to help evaluate Origin memory performance and the directory protocol. The microbenchmarks measure memory latency and bandwidth for different directory protocol transactions, and various combinations of thread and memory placements. The `snbench` suite was used to analyze a number of Origin systems, from the early 195 MHz R10000 systems to the latest 400 MHz R12000 systems, and ranging in size from dual-processor to 128-processor systems. 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. The Irix operating system does not provide an interface to the hardware event counters that supports high-resolution (sub 1 ms) sampling. Since one of the goals of the memory profiler was to capture the behavior in short application phases (e.g., the matrix transpose in the FFT kernel), a new interface to the hardware event counters was needed. The `snpc` loadable kernel module (LKM) exports the hardware event counters as a set of memory mapped files, enabling the profiler to be written as a regular user process, which increases the flexibility of the profiler and minimizes the impact on the operating system. Even though the microbenehmark suite and the memory profiler were written for the Origin 2000, we believe that our approach is applicable to other ccNUMA systems.

The thesis is organized as follows: Chapter 2 discusses related work. Chapter 3 introduces the basic concepts behind the directory-based cache coherence, and gives an overview of the Origin 2000 hardware implementation and the ccNUMA features of the Irix operating system. Chapter 4 presents the implementation of the `snbench` microbenchmark suite, and the results collected on a variety of Origin systems. The implementation of the memory profiler and the loadable kernel module is presented in Chapter 5, while Chapter 6 shows some examples of how the memory profiler can be used. Chapter 7 concludes the thesis.

Chapter 2

Related Work

2.1 Microbenchmarks

Microbenchmarks are small programs designed to measure a very specific aspect of an underlying system architecture. Microbenchmarks can be written in assembly or in a high-level language. In the later case, they evaluate not just the hardware performance but also the quality of the compiler and the supporting run-time libraries. The results from a collection of microbenchmarks can be used to predict performance of more complex applications.

One of the pioneering works in using micro benchmarks to evaluate system characteristics was done by R. Saavedra [33, 34, 35]. His collection of microbenchmarks evaluates both CPU and memory performance. The CPU microbenchmarks are defined in terms of an abstract machine that is essentially a Fortran execution environment. Most of the CPU metrics are based on arithmetic and trigonometric functions: integer and floating-point add, multiply and divide, complex arithmetic, intrinsic functions, logical operations, branch/switch operations, procedure calls, array indexing, and loop overhead. Saavedra also introduced a novel way of evaluating memory performance. He uses two computing kernels to stride through arrays of different sizes. The *read-use kernel* (RU) is used to measure memory read performance; this is a simple reduction sum. The *read-modify-write kernel* (RMW) combines read and write operations. For example, a RMW kernel for a given array size R and stride S is:

```
for (i=0; i < R; i += S)
    a[i] = a[i]*a[i] - CONST;
```

The P^3 diagram is constructed by varying the array size R and stride S. This process yields a number of curves, where each curve represents a fixed array size and the data points on each curve give timing results for a given array stride (increasing in powers of two). Figure 2.1 shows the P^3 diagram for a RMW kernel measured on a 250 MHz Origin. The array size varies from 16 KB to 32 MB.

The P^3 diagrams reveal various levels of the memory hierarchy and the latency characteristics of each memory level. Memory parameters can be deduced by observing when the plots move between different regimes. Saavedra [33] has shown how to use the P^3 diagrams to infer block size, cache size, associativity and even the fact that replacement strategy is random instead of LRU or FIFO. For example, Figure 2.1 reveals that the primary data cache size is 16K (the 16 KB curve

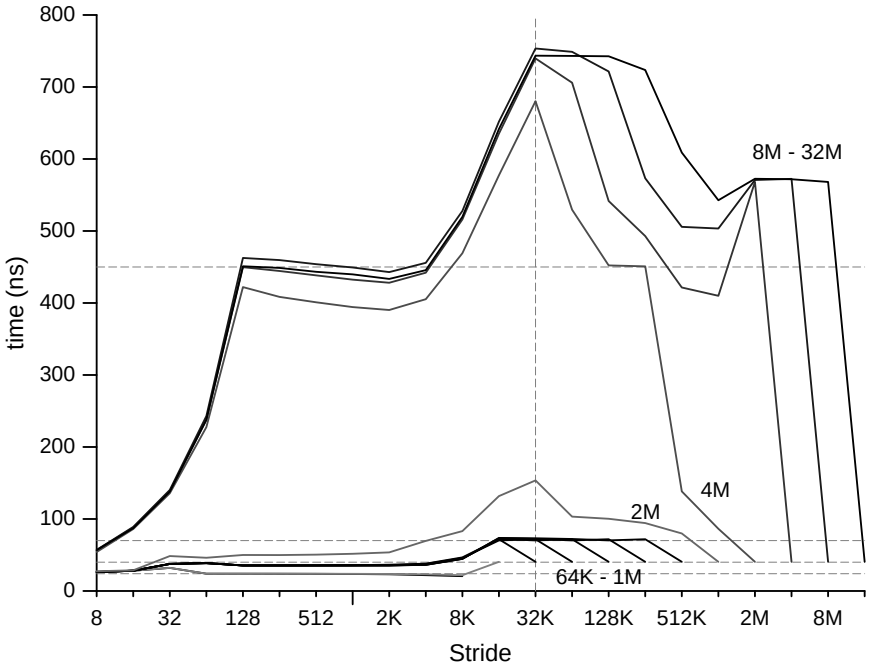


Figure 2.1: The P^3 diagram for a 250 MHz Origin system

fits entirely in L1). The secondary cache size is 4 MB two-way set-associative with a random replacement policy: all curves up to and including 1 MB fit in the L2; the 2 MB curve is slightly above others which implies that the set size is 2 MB with a random replacement algorithm (some lines are replaced even though the whole array fits in one cache set); there are two sets in the cache because the 4 MB curve is just slightly below the curves for larger array sizes where each miss goes to main memory (again, the curve is slightly lower due to random replacement). The secondary cache is not truly set-associative: it uses way prediction bits that are revealed by increased latency for 64 KB – 1 MB cirves at strides 16–512 KB. A miss to main memory combined with a writeback of a dirty line costs ≈ 450 ns; the secondary cache line size is 128 bytes (curves 4M–32M flatten out at stride 128). Each TLB entry maps two consecutive 16 KB pages and the TLB miss penalty is ≈ 300 ns (the jump for 4M+ curves when stride exceeds 32 KB).

While the P^3 diagrams reveal the basic memory parameters, the measured latencies do not accurately reflect the memory timing. There are several reasons for the inaccuracies. First, the RU and RMW kernels do not use dependent loads—an optimizing compiler will unroll the loop several times and issue the loads in parallel; the measured time of the kernel, divided by the number of loads therefore reflects the *pipelined load latency*, where multiple loads can overlap on a processor capable of processing more than one outstanding load. Second, the microbenchmark includes the loop overhead, and it makes no attempt to subtract it from the total time. Third, the loop is timed with standard Unix time functions; the precision of such timer facilities is usually around 10 ms, which requires a high number of repetitions for accurate results.

Another collection of microbenchmarks which evaluates memory parameters is lmbench [28] by McVoy and Staelin. Unlike Saavedra's microbenchmarks, which focus on the CPU performance, McVoy and Staelin's microbenchmarks were designed to evaluate operating system per-

formance. They include codes which measure the time for process context switching, interprocess communication, I/O, virtual memory system performance, and various Unix system calls (e.g., `fork`, `exec` and `getpid`). The lmbench suite of tests also includes two microbenchmarks that evaluate memory latency and bandwidth parameters. They recognize the deficiency of Saavedra's kernels to evaluate memory latency; instead of the pipelined latency they measure latency by timing the execution of a pointer-chasing loop (`p = *p`). (Early versions of this benchmark also assumed that the load instruction takes one clock cycle and thus subtracted one clock cycle from the measured latency). Similar to Saavedra's P^3 diagrams, their latency microbenchmark varies the size of the linked pointer list and the stride between the elements. The resulting latency plots are similar to Saavedra's P^3 diagrams.

The latency measurement in lmbench is quite accurate given that the microbenchmark is portable and written in a high-level language (C). McVoy and Staelin's method has some problems, though. For example, they assume that the cache has a LRU replacement policy—a walk through an array larger than the cache does result in a few cache hits on systems with a random replacement policy but the number of hits is comparatively small and does not significantly skew the results. Additionally, the pointer-chasing kernel measures the *back-to-back latency*, which is a pessimistic estimate of memory latency. On systems that return the critical data word first, the back-to-back latency includes the time when the processor is blocked due to a shared resource (e.g., the secondary cache interface on R10000). McVoy and Staelin argue that the optimistic latency measure (the time from the detection of a miss until the memory returns the critical word and the processor restarts the pipeline which they call latency in-isolation) is too optimistic and not likely to be relevant for actual commercial programs.

Accurately estimating the lower bound on load latency is a nontrivial task, especially when the microbenchmark is intended to be portable. Hristea et al. [19] have written another collection of microbenchmarks that focuses entirely on memory performance. They distinguish between the back-to-back latency (pointer-chasing kernel, similar to the one used by lmbench) and *restart latency* (i.e., latency in isolation). They estimate the restart latency by adding an increasing amount of work instructions following the load, where the work instructions do not generate any memory requests and are dependent on the result of the load. When the time needed to execute the work instructions exceeds the time when the shared resources are busy, the work time is subtracted from the total time, which yields an estimate of the restart latency. They have compared results on the Origin 2000 and Sun UE 10000. The latency results show a significant difference between the back-to-back and restart times for the R10000. Hristea et al. also recognized that different cache-coherence transactions incur different costs. They use remote threads whose task is to place the cache lines in the desired state. For the Origin, results are presented for unowned, clean-exclusive and dirty-exclusive lines. Finally, Hristea et al. used the microbenchmarks to evaluate NUMA latencies and bandwidths—they present results on the Origin 2000 where the requestor is both at the home node and one hop away.

A popular microbenchmark which evaluates pipelined memory bandwidth is STREAM [27]. The kernels in this microbenchmark use four typical operations used by scientific codes written in Fortran: copying an array to another array, multiplying all elements of an array by a fixed value, computing a sum of two arrays, and computing a sum of two arrays where the second array is multiplied by a scalar. The results are computed on arrays that do not fit in the processor's caches. In all kernels the results are written to a separate array. STREAM does not evaluate pure memory read bandwidth. Both lmbench and Hristea et al. use a reduction loop to evaluate memory

read bandwidth. Additionally, lmbench also evaluates memory write and copy bandwidths by measuring the performance of manually unrolled loops and the library `bcopy` function.

Our collection of microbenchmarks, called snbench, focuses on back-to-back and restart latencies and memory bandwidth. The primary goal was accuracy instead of portability. The latency kernels were written in assembly to have full control over instruction execution. Another goal of our mucrobenchmarks was the evaluation of ccNUMA coherence protocol transactions: `snbench` has the ability to place threads and memories anywhere in the system, and to place cache lines in any coherence state before executing the timed kernel. We also use front-end scripts to determine the topology of the Origin system and generate a number of invocations of the `snbench` executable with different combinations of thread placement. These experiment groups are designed to evaluate the characteristics of several important classes of Origin coherence protocol transactions. We use a pointer-chasing kernel similar to the one used in lmbench to evaluate the back-to-back latency. We have developed a new algorithm to automatically evaluate the restart latency. The basic approach is similar to the one used by Hristea et al., except that our algorithm uses automatically generated kernels in assembly language to get around compiler optimizations. Finally, we use a different algorithm for computing restart latency.

2.2 Performance Analysis Tools

Many tools have been used to help users to speed up applications. There are many performance analysis tools and many ways to evaluate program behavior. The most basic information about program execution is the number of times each instruction, statement, basic block, or function has been executed. This information can be collected by instrumenting the executable at either compile or run time. While counts give information about relative frequency of execution for each part of the program, these frequencies do not necessarily have a strong relationship with the time spent in each section. The simplest approach in collecting timing information is to use statistical profiling: the program is interrupted at specified intervals, and the value of the program counter is used to index into an array of counters that can be mapped back into program source code. On Unix systems, the `prof` command [4] uses this approach.

Statistical profiling can be combined with a static program call graphs to attribute parts of the execution time for each function to the other functions it calls (function's descendants in the call graph). The `gprof` execution profiler [15] uses instrumentation added by the compiler to obtain runtime information about function call counts. The run-time library also constructs the histogram of the location of the program counter with the help of the operating system. Function call counts and a program counter histogram are combined in a postprocessing step that attributes the time for each function to the functions that call it. Each parent in the call graph receives some fraction of a child's time. With `gprof`, timing attribution is based on the number of calls from parent to child: if a parent called a child function p times and the child was called n times in all, a fraction of p/n of the total time spent in the child will be attributed to the parent. This does not necessarily reflect true run-time behavior: if calls from function A cause function C to return immediately while calls from function B account for the majority of time spent in C, the attribution of C's time to A and B based only on the number of calls can be inaccurate.

Statistical profiling answers the question about how much time was spent in each part of the program, but it does not give any answers about the cause. The operating system usually collects

some statistics during program execution, such as the number of page faults, floating point exceptions and process memory usage. An integrated performance analysis framework such as SGI's SpeedShop [51] uses this information to help the user characterize program behavior: whether the process is CPU-bound, I/O-bound, or memory-bound. It can also help in program debugging by instrumenting calls to memory allocation functions and matching memory request and release calls. Performance analysis with SpeedShop is done by first profiling the application with one or more experiments and then analyzing the profiles with a separate GUI interface. SpeedShop experiments include both traditional (flat) and call-graph based statistical profiling; in the latter, SpeedShop performs proper attribution of the time spent in the callee to the callers by unwinding the stack on each clock tick to obtain the complete run-time call chain.

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 [48], which can be used to count the number 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 [10], DEC Alpha [12], HP PA-8000 [20] and Intel Pentium processors.

These counters were primarily used by hardware developers and a few performance analysts. To be used effectively, hardware event counters require some operating system support and tools which use them. Cray Research traditionally offered strong support for hardware performance monitors. SGI was the first workstation and server vendor to include OS support and tools to use hardware performance counters. The Irix operating system supports virtualized performance counters, which are a part of each kernel-visible thread context [44]. Even though the R10000 processor only includes two physical counters, the operating system removes this limitation by multiplexing several events on one hardware counter. No special instrumentation is required to profile a program with hardware event counters. Users can measure cumulative event counts for the whole duration of the process with the `perfex` tool [43]. SpeedShop uses hardware event counters for statistical profiling by using performance counter overflow as the interrupt source. This makes it possible to correlate processor events (e.g., cache misses) with the locations in program source code.

DEC's Digital Continuous Profiling Infrastructure (DCPI) project [2] 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 [11]. 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 [29] and PCL [5] 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.

Chapter 3

Background

Multiprocessor systems have been used for a long time to construct computers with a much higher aggregate performance than the fastest uniprocessor system. In order to leverage the fast design cycle of modern microprocessors, the most common way to harness the power of multiple commodity microprocessors is a design based on a common bus that connects the microprocessors, the system memory, and one or more I/O bridges. Such systems use bus snooping to enforce cache coherence; they are commonly referred to as *symmetric multiprocessor systems* (SMPs). This acronym is used in operating system literature to describe multiprocessor kernels where all processors are functionally equivalent, and it is also used by computer architects to describe systems that implement the *shared memory programming* model. In such systems, all processors share the same (physical) address space and the communication is based on memory operations such as loads and stores. The SMP systems typically have support for atomic memory operations, which are the basis of higher level synchronization primitives such as locks, semaphores and barriers. The terms shared memory programming and *uniform memory access* (UMA) are often used interchangeably, although the terms are not strictly equivalent.

SMP systems are widely used as servers and are being used as high-end workstations. They are relatively easy to build and relatively easy to program: the shared address space programming model is well understood. The snoopy bus that is the basis of cache coherence protocols is at the same time the fundamental limitation to their scalability. There are several mechanisms which facilitate a higher bus throughput and thus a higher processor count: multiple interleaved split-transaction busses combined with relaxed memory models and a lot of clever engineering have pushed the number of microprocessors in a single SMP system up to 64. However, such large processor counts are pushing the scalability limits of SMP systems.

Scalable multiprocessors replace a shared bus with a scalable interconnect. This change opens up a vast design space that covers a range of systems from tightly coupled multiprocessor systems to networks of workstations. The shared address space that makes SMP systems so attractive is typically the first casualty of scalability. Scalable multiprocessors replace it with some form of message passing, either explicitly through communication libraries (such as PVM or MPI) or implicitly with parallel language constructs which are translated into messages by the compiler (e.g., High Performance Fortran or OpenMP). The programming model is suitable for relatively coarse-grained parallelism, depending on how tightly the processor is integrated with the network. Clusters of relatively inexpensive computers connected with a fast general-purpose network (Myrinet, 100 MB/s Ethernet) are gaining popularity in scientific processing space due to their

price/performance characteristics. When used with commercial applications, they are typically found in high-availability cluster setups.

However, eliminating the shared bus does not preclude a shared address space. Scalable cache coherence can be implemented with a protocol which keeps track of the state of each cache line by exchanging messages among the participants (processors and memories). Scalable cache coherent systems consist of a number of nodes, each with one or more processors, a portion of system memory and a communication interface; the nodes are linked together with a scalable interconnect. Such systems deliver the convenience of programming in a shared address space and a promise of scalability to high processor counts.

The SGI Origin 2000, although not the first commercially available scalable multiprocessor system, was nonetheless one of the most ambitious designs when it was introduced in the market. Its design came out of two research efforts at Stanford University, the DASH multiprocessor [23] and its follow-on, the FLASH project [22]. Both projects explored the limits of a particular multiprocessor design, the cache-coherent nonuniform memory architecture. The Origin 2000 family of multiprocessor systems was introduced by SGI in 1996. The modular design allows for scalability from two to 512 processors. The hardware cache coherence is achieved by means of a memory-based directory cache coherence protocol. It features an aggressive communication mechanisms and hardware and software support for NUMA features such as page migration and replication, support for explicit thread and data placement and a novel way of enforcing TLB coherence (directory poisoning).

This chapter describes the basics of scalable shared-memory multiprocessing and the design and implementation details of the Origin 2000. Section 3.1 gives an overview of directory-based cache coherence and some of the design trade-offs. In Section 3.2, we describe the choices made by the Origin designers. In particular, we describe the hardware organization, the network topology, and the design of the directory cache coherence protocol. The ccNUMA hardware features need software support. Section 3.3 gives an overview of the operating system interfaces and commands that let user applications take advantage of various Origin hardware features.

3.1 Directory-Based Cache Coherence

The cache coherence in SMP systems is typically enforced with a snoopy bus protocol such as MESI (Illinois) [30]. The unit of coherence is a small cache line (usually between 32 and 128 bytes). Each processor maintains a state of each cache line in its local cache, In the MESI protocol, the state is either invalid, shared, exclusive or modified. Several processors may have a read-only copy of a shared cache line in their local caches simultaneously. A processor can obtain an exclusive copy, which requires that the other processors invalidate that cache line in their local caches. A processor can modify the contents of a cache line only after obtaining exclusive ownership; at that time, the state is changed from exclusive to modified. When the line is in the modified state, the current owner will supply the data when another processor requests a copy. The data for exclusive lines can be supplied either by the owner or by the memory controller. The cache controller performs state transitions by processing memory requests from the processor and by snooping on the requests on the shared bus. The simplicity of this scheme relies on the shared bus. All memory requests are broadcast on the bus and all processors (and memory controllers) listen in on the traffic.

A scalable multiprocessor decouples the system into a set of nodes linked together with a scalable interconnection network. Each node consists of one or more processors with corresponding caches, a portion of system memory, and a communication assist which connects the node to the rest of the system. There are several approaches to achieving cache coherence in scalable multiprocessors. The broadcast and snooping mechanism can be extended with a hierarchy of broadcast media such as busses or rings. Another approach is based on the concept of a directory: the global state of each cache line is maintained in a known location—the directory—where the requests can go and look it up. Each cache line is assigned a home where the directory state is kept. Typically, each node in the system is home to a portion of the system memory. Yet another approach uses a two-level protocol hierarchy where the inner protocol keeps the data coherent within a node and the outer protocol maintains global coherence. The inner and outer protocols need not be the same. A common organization is for the outer protocol to be a directory protocol and the inner one to be a snooping protocol [23, 25]. Other combinations, such as snooping-snooping [13] and directory-directory [9] are also possible.

On a machine with physically distributed memory, nonlocal data may be replicated either in the processor's caches or in the local main memory. The systems that replicate data only in processor caches and keep the data coherent in hardware at the granularity of cache lines similar to the bus-based systems are called *cache-coherent nonuniform memory access (ccNUMA)* architectures. More generally, systems that provide a shared address space programming model with physically distributed memory and coherent replication (either in caches or in local memory) are called *distributed shared memory (DSM)* systems. The ccNUMA approach is an example of the DSM architecture. Other DSM system architectures include COMA (cache-only memory architecture) [13, 18], Simple COMA [36, 32], and software DSM approaches [24, 6, 21]. The design space for DSM architectures is very broad; however, performance issues favor designs which implement cache coherence in hardware. Among hardware solutions, directory-based ccNUMA systems appear to be the most popular choice for scalable shared-memory multiprocessing.

3.1.1 Protocol Operation

The directory protocol is invoked if the processor makes an access that its cache cannot satisfy by itself. For example, an access to a cache line which is not present in processor's cache or a store to a read-only copy of the cache line. Unlike the snoopy cache coherence where the processor simply places the request on the shared bus and waits for a response from either memory or another processor, the directory protocol has to communicate with other participants in the system in order to satisfy the request. First, the protocol needs to find out enough information about the state of the cache line to determine what action to take. If there are other copies cached in the system, the protocol must be able to locate them in case it needs to invalidate them. Finally, the protocol must provide a way of communicating with the other copies. The directory protocol finds the information about the state of cache lines by looking up the directory through network transactions. The location of copies is also found from the directory and the communication with other copies is done by network transactions in an arbitrary interconnection network.

In a ccNUMA system, each cache line is assigned a *home node*. The home keeps the global state of the cache line in addition to the data. Each node in the system is home to a portion of total system memory; the home node is determined from the globally unique address of the cache line. The nodes are connected in a network. In addition to the memory controler, each node also has a

communication assist which is responsible for communicating with other nodes. The *local node*, or *requesting node* is the node containing the processor that issues a request for the cache line. The communication assist in the local node sends a network message to the home node asking for a copy of the cache line. Depending on the directory state of the cache line, the protocol returns a copy immediately if there are no other copies cached in the system. If there are read-only copies cached by other nodes in the system and the local node requests an exclusive copy, the directory needs to locate all *sharer nodes* and invalidate their copies before granting the local node exclusive access. Finally, if the only valid copy of the cache line is located in a remote processor's cache, the directory needs to locate the *dirty node*, invalidate the dirty copy in remote processor's cache and send the modified data to the local node (SGI literature refers to this as an *intervention*). The *owner node* is the node that currently holds a valid copy of the cache line and must supply data when needed; in directory protocols, the owner is either the home node or the dirty node.

When a cache miss occurs, the local node sends a request to the home node where the directory information for the cache block is located. On a read miss, the directory indicates from which node the data may be obtained; on a write miss, the directory identifies the copies of the block and the protocol needs to invalidate the copies by sending invalidate messages to each individual copy. Each invalidation requires a separate acknowledgment after the node has invalidated the cache line in its processor's caches; in systems which implement strict memory consistency (such as Origin 2000), the local node can proceed with a write only after the acknowledgments from all sharer nodes have been received.

3.1.2 Directory Organization

The natural place for directory information is to include it with the main memory in the home node. At the very least, the directory information needs to include cache line state and enough information to locate the owner or the list of sharers. This scheme is known as flat directory scheme because the source of directory information is determined by the address of the cache line. On a cache miss, a single network transaction is sent directly to the home node to look up the directory regardless of how far the home node is. An alternative to the flat memory scheme is to organize the directory as a hierarchical data structure (a tree). In the *hierarchical directory schemes*, the source of directory information is not known a priori: upon a miss, the directory information is found by traversing up the hierarchy by network transactions until a node is reached which indicates that its subtree has the desired information. The latency and bandwidth characteristics of hierarchical directory schemes tend to be much worse than that of the flat schemes; hierarchical organizations are therefore not popular in modern multiprocessor designs.

In flat directory schemes, the list of sharers can be included in the directory along with the cache line state. If the directory needs to invalidate a list of sharers the information is available when the original request reaches the home node. Since all relevant directory information is kept in memory at the home node, this approach is known as the *flat memory-based scheme*. Origin 2000 uses this approach; similar designs include the Stanford DASH and Flash machines. There are several possibilities on how to keep the list of sharers. The simplest case is to keep a vector of presence bits where each sharer is represented by a single bit—if the bit is set the corresponding node has a cached copy of the cache line. If the system contains many nodes, the directory information can consume a significant portion of system memory. An alternative to the bit vector scheme is to keep the sharers organized as a linked list (limited pointer directories) or to keep a coarse bit

vector where a single bit represents several nodes. Another alternative is to organize the directory itself as a cache. For example, the Origin systems use two directory formats to keep the metadata overhead low: systems up to 32 processors (16 nodes) use a 32-bit directory entry for each 128-byte cache line (a 3% memory overhead); larger systems use a 96-bit directory entry with a 64 presence bits (a 9% overhead) with the protocol reverting to a coarse-grained scheme where each bit represents eight nodes for systems larger than 128 processors (64 nodes). Regardless of the directory organization, the memory-based scheme keeps all information about the sharers at the home node.

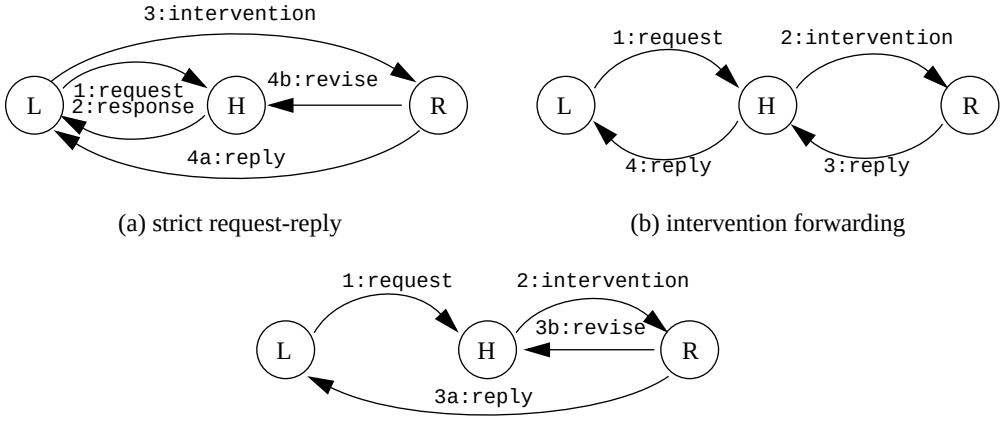
An alternative to the memory-based scheme is for the directory entry to contain only a pointer to the node that holds a copy of the cache line. When there are multiple sharers of the cache line, each node holding a copy keeps a pointer to the next sharer. The sharer list is usually organized as a doubly-linked list to reduce the number of transactions required when one of the sharers removes a copy of the cache line. Instead of the main memory, the directory information is distributed along with the copies of the cache line; this approach is known as the *flat cache-based scheme*. The distributed maintenance of the sharers is considerably more difficult than manipulating the presence bits or a linked list stored at the home node. The complexity issues of this design have been alleviated by the publication of a standard for cache-based directory organization and protocol, the IEEE Scalable Coherent Interface (SCI) [17]. Several commercial systems use the SCI protocol (e.g., Sequent NUMA-Q [25], Convex Exemplar [9], and Data General [8]). While the cache-based schemes reduce the memory used for directory data, the additional transactions required to maintain distributed directory state increase the critical path latency. For systems of moderate size, it is not clear whether the reduction in memory offsets the decrease in memory performance.

3.1.3 Performance and Correctness Issues

The directory protocol must maintain the correct states of cache lines while preserving serialization and ordering relationships required by the coherence and consistency requirements. Additionally, the protocol must be free from deadlock, livelock and, if possible, starvation. At the same time, the directory protocol should satisfy two basic performance goals—to reduce the number of network transactions generated for memory requests and to reduce the number of network transactions on the critical path. This section describes some trade-offs for improving protocol performance for flat memory-based directory protocols.

Figure 3.1 shows three different protocol transactions that could be used to move the ownership of a cache line from the current owner R to the requestor L; H is the home node. All three transactions start with the local node sending a request for ownership transfer to the home node. In a *strict request/reply protocol*, the home node responds to the requestor with the identity of the current owner. The local node then sends a separate request to the remote node, which replies to the requestor with the cache line contents and informs the home of the new owner's identity with a revision message. This transaction requires five network messages, four of which are on the critical path.

The number of network messages can be reduced by one when the strict request/reply is replaced by *intervention forwarding*. Instead of the requestor sending the message to the owner, the home node forwards the intervention; the owner responds to the home node, which in turn sends the reply to the requestor. The latency is not reduced as there are still four messages on the criit-ical path. Intervention forwarding has a disadvantage that the home node needs to keep track of



(a) strict request-reply

(b) intervention forwarding

(c) reply forwarding

Figure 3.1: Reducing protocol latency through forwarding

outstanding requests instead of the requestor. This means that the buffering resources at the home node have to be increased substantially: if there are P processors in the system, each having up to n outstanding requests, the home node would have to keep track of nP requests in the worst case instead of only n required when the request tracking is done by the requestor. This requirement makes intervention forwarding an unlikely protocol choice.

The number of messages on the critical path can be reduced to three by the use of *reply forwarding*. This transaction is similar to intervention forwarding—the home node forwards the intervention request to the remote owner; however, the remote node sends a reply to the requestor directly and informs the home node with a revision reply. The critical path now consists of the original request message, followed by the intervention and reply; the revision message is sent in parallel with the final reply. This scheme has the best latency and bandwidth characteristics. It does have a disadvantage of increasing the message dependency length from two (request/reply) to three (request/intervention/reply), which has potential implications for deadlock avoidance.

All distributed systems have to consider the possibility of a deadlock. Typical deadlock scenarios in a request/response protocol involve buffer overflow: in order to complete a transaction, a reply message has to be sent to the requestor; if the output queue is full the transaction cannot be completed and the system could end up in a deadlock. One solution for this problem is to provide enough buffer capacity either by anticipating the worst-case scenario or by providing extra buffering in hardware or in main memory (the approach used in the MIT Alewife system [1]). In large systems, providing adequate buffering space could result in underutilization of system resources or could adversely impact system performance. Another solution is to send a negative acknowledge (NACK) whenever there is not enough output buffer space. The third solution is to provide a separate request and reply networks, either by physical network separation or by multiplexing (e.g., virtual channels) with separate buffering space for each channel.

Both the Stanford DASH and the Origin 2000 systems use separate request and reply networks for deadlock avoidance. This solution is adequate for strict request/reply protocols. However, both systems use reply forwarding to minimize request latency. As noted above, reply forwarding extends the dependency chain to request/intervention/reply; in order to prevent a deadlock the

system would have to provide three separate networks. Since the third network would mostly be underutilized, deadlock can be prevented with only two separate networks by detecting when the system could deadlock (e.g., no output buffer space) and either NACK-ing the incoming request (the DASH solution) or reverting to a strict request/reply protocol (Origin 2000). The use of NACKs can lead to the livelock problem: the messages are being exchanged but no forward progress is achieved. In this respect, reverting to strict request/reply protocol is preferable because it does not suffer from livelock problems.

The final issue that the coherence protocol has to consider is starvation. When NACK messages are used there is a possibility some requestors always receive a negative acknowledge while other requestors are accessing a cache line. In reality, starvation is a result of extremely pathological timing. The designers could discard starvation as a remote possibility (e.g., the DASH protocol does nothing to prevent livelock and starvation). A relatively inexpensive solution is to assign priorities to the requests. When a request is NACK-ed the requestor increases the priority and reissues the request. When the home node receives a request it cannot satisfy immediately it updates the priority level which is stored in the directory entry; all requests with priority less than the directory priority are NACK-ed. Eventually, the processor that has been denied access to the cache line will eventually increase priority high enough to override all other processors and obtain the ownership of the cache line.

3.2 Origin 2000 Hardware Design

The Origin 2000 architecture is an outgrowth of the Stanford DASH and Flash research projects. The systems debuted in 1996 with machine sizes ranging from two to 64 processors; subsequent versions can scale up to 512 processors. The system is based on MIPS R10000 processors and uses a tightly integrated memory and network controller to implement hardware cache coherence with a flat memory-based directory protocol.

Figure 3.2 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 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.

3.2.1 Cache Coherence Protocol

The Origin 2000 cache coherence protocol is based on the standard three-state invalidation-based directory protocol with several optimizations. It is a nonblocking protocol and it uses reply forwarding with dynamic back-off to prevent deadlocks. The protocol is independent of network

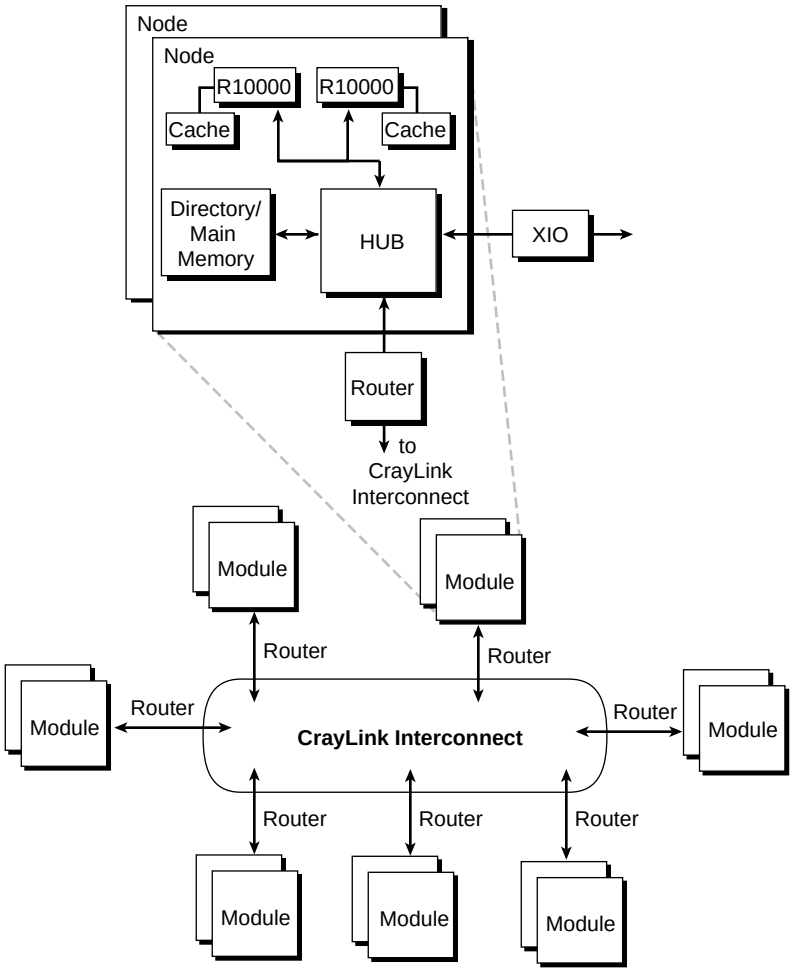


Figure 3.2: Logical organization of the Origin 2000 system

ordering. In addition to processor requests, the protocol supports coherent DMA requests and an innovative way to achieve lazy TLB shootdown.

Each cache line has an unique home determined by the physical address of the cache line. Physical addresses are 40 bits wide: the upper 8 bits select the home node while the lower 32 bits represent the local address within the node. This mode (called M-mode) allows for 256 nodes (512 processors) with a limit of 4 GB of memory per node. An alternative configuration (N-mode) uses 9 bits to select the node and 31 bits for local offset for a total of 512 nodes with 2 GB of memory per node; this mode is not used in existing Origin systems. An in-depth description of global addressing in the Origin systems is given in [49, pp. 3–23].

The directory protocol is nonblocking: the home node will not buffer a request if the response cannot be generated immediately (e.g., because the line is owned by another processor). Rather than buffering the request at home, the directory state is changed to a transient (busy) state while the current owner is being tracked down. The current owner responds to the intervention by updating the directory with a revision message, which causes the transient state to change to one of the permanent states (unowned, shared or exclusive). Requests to cache lines in busy state are

acknowledged immediately with a negative response, which causes the requestor to reissue the request later. Starvation is prevented by assigning requests a priority value, which is a function of the number of times the request was denied.

The protocol does not rely on network ordering: messages could take different routes through the interconnect network and arrive at the destination out of order. The out-of-order message delivery can result in several race conditions. Rather than imposing strict network ordering on the interconnect, the Origin directory protocol handles race conditions explicitly. The only messages which require point-to-point network ordering are certain uncached operations used to implement programmed I/O (PIO).

Directory States

The directory protocol is based on states given in Table 3.1. Four of the states are stable: in the unowned state, the memory keeps the only copy of the cache line; in the shared state, the cache line may be replicated read-only in one or more processor's caches (the identity of sharers is kept in a bit vector stored in the directory); in the exclusive state, one processor or I/O unit keeps an exclusive copy of the cache line that is either clean or dirty (again, the identity of the owner is stored in the directory); in the poisoned state, the cache line has migrated to another node during a page migration and the access will cause a trap if a processor attempts to access this line through a stale TLB mapping. Transient states are used in the intermediate stages of protocol transactions: busy shared and exclusive states indicate that the directory should not handle a new request for the cache line until the previous transaction has completed; the wait state is used to prevent a writeback race condition.

state	*description*
unowned (`UOWN`)	No cached copies in the system
poisoned (`POIS`)	Not cached, access returns bus error
shared (`SHRD`)	Read-only copies cached by one or more nodes
exclusive (`EXCL`)	Cached in exclusive state by one processor or I/O
busy shared (`BUSYS`)	Directory is busy tracking down a shared copy
busy exclusive (`BUSYE`)	Directory is busy tracking down an exclusive copy
waiting (`WAIT`)	Directory is waiting for a revision message

Table 3.1: Directory states in the Origin directory protocol

The directory information is kept separate from the main memory so that memory and directory operations can be overlapped. In addition to the cache line state, the directory keeps the identity of the owner for exclusive lines and a list of sharers for shared lines. The sharers are kept in a bit vector where each bit represents a node; invalidation messages thus target both processors on the node. The decision to keep sharers on a per-node basis allows a 64-bit presence bit vector to cover all nodes in a 128-processor system. Larger systems use a different mode to keep track of the sharers. If all sharers are located in the same 64-node *octant* the presence bits again represent one node in the octant; invalidations are sent to individual nodes. When the sharers are spread across octants, the directory uses a coarse bit vector scheme where each bit represents eight nodes.

Invalidations are sent to all eight nodes, even if some of these nodes may not have a cached copy of the cache line.

Protocol Optimizations

The first optimization that aims to improve protocol performance is the use of *speculative replies*. For all requests that require a data response, the directory controller always performs directory lookup and memory read in parallel. In most cases, the data are sent as part of the response (e.g., read requests on unowned or shared lines). When the home node is not the current owner of the cache line, the data are sent to the requestor in a speculative reply. The speculative data are used if the current owner has a clean copy of the cache line. By using speculative replies, the protocol assumes that it is not likely that the owner has modified its copy. If this is not the case the updated data have to be returned with the response from the current owner while the speculative data are discarded.

The Origin protocol supports the full MESI protocol used by the R10000 processor to manage the L2 cache, including the clean-exclusive state. In exclusive state, the processor is the exclusive owner of the cache line but it has not modified it yet. The processor can invalidate an exclusive line from its cache at any time; this does not present a problem on a snoopy bus because snoop request will miss in owner's cache when another processor requests a line. This does present a problem in a directory protocol—while the processor has silently dropped the cache line the home node still keeps the processor as the owner. One solution is to extend the protocol with replacement hints: whenever the processor drops a clean line it sends a revision message to the home node. The disadvantage of this approach is that it uses extra network traffic and wastes directory bandwidth. The Origin protocol allows a processor to drop a clean line from its cache without notifying the home node (*silent clean-exclusive replacement*). If the same processor later requests the cache line again the directory assumes that the processor has dropped the cache line and immediately returns a fresh copy. On the other hand, if another processor requests a copy the home node forwards the intervention request to the owner as if it still had a copy in its cache. The intervention request will fail. However, the requesting processor can use data returned in the speculative reply. In this case, the extra cost is the additional intervention request, which would not be necessary if the protocol used replacement hints. On the other hand, silent dropping of exclusive lines is beneficial to uniprocessor applications because directory bandwidth is not wasted with replacement hints.

In order to support application execution in ccNUMA environment, the Origin designers decided to implement page migration to improve application performance [7]. Efficient page migration requires a fast data copy mechanism and the ability to globally purge stale TLB entries. Implementing a global TLB shootdown algorithm without hardware support can be very costly. Origin directory protocol allows for an efficient global TLB purge by supporting *directory poisoning*. A cache line can be "poisoned" during the data-copy phase by the copy engine. The poison read request invalidates all cached copies of the line and sets its directory state to poisoned. When a processor attempts to read a poisoned cache line via a stale TLB mapping the directory returns an error. This in turn invokes the kernel exception handler, which recognizes the special error and invalidates the stale TLB entry. No expensive global synchronization operations are necessary: processors invalidate stale TLB entries on demand.

Protocol Transactions

The directory protocol supports coherent processor read and write operations, coherent I/O requests which are used by the DMA engines, and a number of noncoherent operations which are used for processor-initiated programmed I/O. The processor can generate three types of read requests to coherent memory. The *read-shared request* (RDSH) asks for a read-only copy of the data; this request is used for instruction fetches. The *read request* (READ) returns an exclusive copy if there are no shared copies cached in the system; an exclusive copy is downgraded to a shared state with both the requestor and the current owner in the list of sharers; if the line is already shared the requestor is simply added to the list of sharers. This request is generated when a load instruction misses in the cache. The *read-exclusive request* (RDEX) asks for an exclusive copy of the cache line; it is generated when a store instruction misses in the cache.

Depending on the state of the cache line, read requests trigger different types of directory transactions. If the line is in the unowned state, the home node sends the reply immediately and updates the directory to reflect the new owner. The *unowned transaction* requires only two messages, a request and a reply. If the line is shared read-only among several nodes in the system, and a processor requests an exclusive copy of the cache line, the shared copies have to be invalidated before the requestor can assume exclusive ownership. The resulting *invalidate transaction* is a three-step transaction: the requestor sends a request to the home node, the home node forwards invalidations to the sharers, and the sharers reply with acknowledgment messages to the requestor. Finally, if a processor requests a copy of a cache line whose directory state indicates that another processor holds a valid copy, the remote processor needs to relinquish the ownership and possibly send a locally modified copy to the requestor. There are several variants of this scenario which involve three participants: the requestor, the home node and the remote owner. The *intervention transacton* is a broad term, covering all variants of this scenario. The Origin directory protocol supports two more transaction types, variants of the intervention and invalidate transactions, which are used when the system detects a possibility of a deadlock. In this case, the directory protocol reverts to a strict request/reply protocol. The resulting *backoff intervention* and *backoff invalidate* transactions are four-step transactions where the local node first sends a request to the home node, the home responds with a backoff response to the requestor, the local node then sends the intervention or invalidate requests to the remote nodes and waits for the response.

3.2.2 Node Board

The node board is the basic building block of the Origin system. It holds two R10000/R12000 processors and a portion of system memory and the associated directory memory. Two bidirectional interfaces connect the node board to the rest of the system. The communication with other nodes goes through the CrayLink port which connects the node to a router board. The local I/O system is accessible through the XIO port. At the center of the node board is the Hub ASIC, which connects the processors, memory, I/O and network interfaces.

Figure 3.3 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

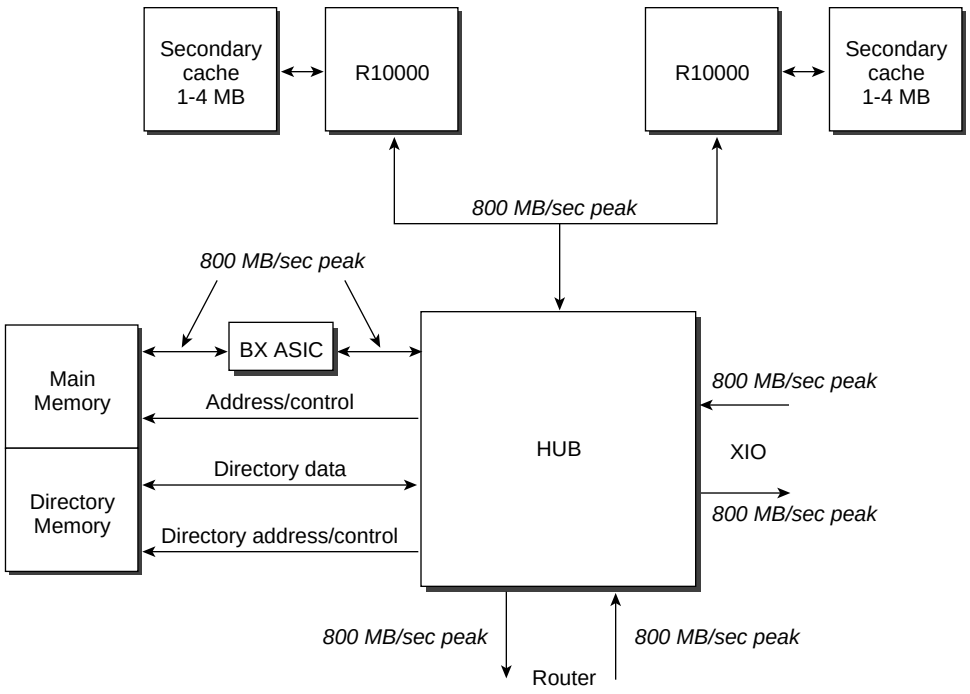


Figure 3.3: Block diagram of the node board

possible to perform local cache-to-cache transfers. There are separate dual inline memory modules (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. 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. Figure 3.4 shows the block diagram of the Hub ASIC. The external connections (CrayLink and XIO ports) both use a similar protocol; the only difference is in the messages used by the communication protocol. The CrayLink messages are tailored for the needs of directory-based cache coherence whereas the messages in the XIO protocol are used to control the I/O devices (storage, network and graphics) which have different requirements. The original version of the Origin featured R10000 processors running at 195 MHz; in these systems, the Hub core operates at 97.5 MHz. Newer systems with 250 MHz R10000 processors and 300 or 400 MHz R12000 processors have the Hub running at 100 MHz. All Origin systems have the interconnect network and the XIO interface running at 400 MHz. To accommodate the variations in Hub frequency, both the network and the XIO interface have to provide an asynchronous interface.

One of the Hubs in the system can be designated as the source of a global clock. The clock information is propagated by the routers to all other nodes in the system. This hardware feature is used to implement a global, synchronous high-resolution (800 ns) cycle counter. Applications can map a page of Hub physical space into their address space to access the cycle counter directly, which results in an extremely low access time (300 ns).

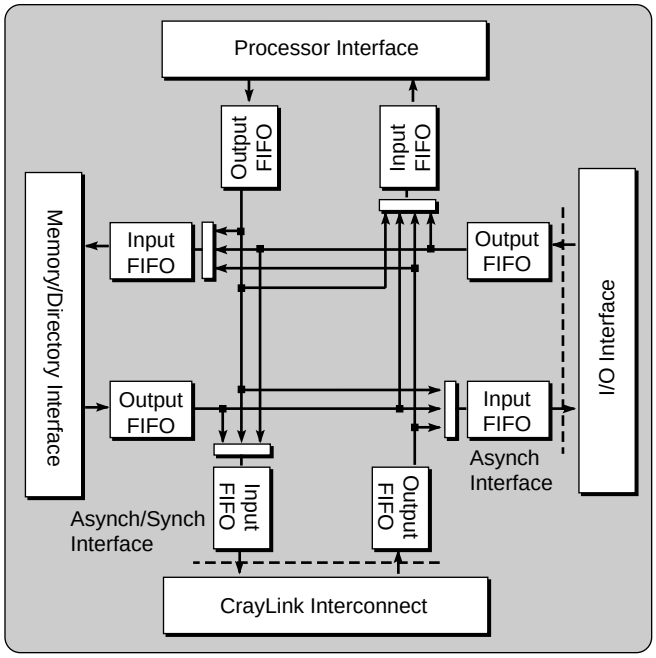


Figure 3.4: Block diagram of the Hub ASIC

Memory/Directory Interface

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.

In addition to the directory state and the sharer list, the Hub provides a set of reference counters for every 4 KB page of main memory. The number of counters in the set depends on the size of the system: for systems with less than 64 nodes (128 processors), there is one counter for every node in the system; in larger systems there is one counter for every 8 nodes. Whenever the MD interface receives a memory request it increments a counter for the node where the request was issued. This information may be used by the operating system to dynamically migrate pages and by the applications to obtain remote access counts for each 4 KB data block. The operating system can set up an interrupt delivery whenever the number of remote requests less the number of local requests passes a specified threshold. This indicates that the page is a good candidate for migration and the kernel may choose to migrate it closer to the requestors. Additionally, the operating system can allocate 64-bit virtual counters that are updated regularly with the physical reference counters. The virtualized page reference counters are accessible through a kernel API. An application can access a set of virtual reference counters corresponding to a 4 KB memory block that stores a portion of the application data, thus providing an estimate of remote vs. local reference patterns; the Mutt project [16] provides a library interface that associates virtual reference counters with portions of application address space. The virtual page reference counters have a major drawback: the kernel reserves a large portion of main memory for in-core copy of the reference counters; by default, Irix disables this interface.

In addition to the processing of memory requests, the MD interface implements the *at-memory*

fetch-and-op synchronization mechanism. This mechanism uses uncached memory references to Hub memory special space (Mspec) [49, p. 23]. The uncached loads and stores to Mspec space operate on 32-bit words or 64-bit doublewords. The address of the cache line also encodes the atomic operation to be performed. In addition to a simple read of the cache line, load operations can also perform atomic increment, decrement and clear. Store operations can perform atomic bitwise and/or, increment decrement (the stored data is not used in this case). The Hub maintains a two-entry cache of recently used fetch-and-op cache lines; both load and store operations can explicitly flush the entry from the fetch-and-op cache. The use of fetch-and-op synchronization primitives is described in [40].

Processor Interface

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 state of pending requests for each processor is maintained in a special data structure, the *coherency request buffer* (CRB). Each processor has its own CRB table. The CRB is divided into read request buffers (RRB), which track outstanding processor read requests; write request buffers (WRB), which track pending stores; and intervention request buffers (IRB), which keep the information for pending interventions and invalidates. The RRB, WRB and IRB buffers for each processor are kept coherent to detect possible race conditions. However, the CRBs for each processor are not kept coherent with respect to each other. Requests generated by the PI which target the peer processor on the same node have to be converted into protocol messages and sent through the crossbar in the Hub back into the PI's incoming request queue.

Network Interface

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. Additionally, the NI is responsible for taking a compact internal invalidation message where the list of sharers is kept in a bit vector and generating the multiple unicast invalidate messages required by the protocol.

I/O Interface

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 BTEs are used in page migration and page poisoning; the Irix kernel also uses the BTE internally to zero-out memory pages. The II keeps track of the outstanding IO requests via a set of I/O request buffers (IRB). The IRB tracks both full and partial cache line DMA requests by the I/O devices as well as full cache line requests by the BTE.

3.2.3 Interconnect Network

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 (the signalling technology allows a maximum cable length of 5 meters). Systems up to 64 processors (32 nodes) are organized in pure hypercubes; Figure 3.5 shows Origin topologies up to 64 processors. The largest configuration uses all router ports for the construction of the hypercube. In smaller systems, unused router ports can be used as *express links* (shown in Figure 3.5 as dashed lines) to shorten the longest distance between nodes. 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 3.6 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.

System topology and routing tables are set statically at boot time. The design is based on static routing tables that determine packet routing in a pipelined fashion. The header of the packet contains the exit port when the packet enters the router. The router sends the packet on the selected exit port and updates the header information with the new exit port for the next hop according to its local routing tables. This design results in a fast wormhole routing with minimum router delay (the pin-to-pin packet latency is 41 ns [14]). While the hardware design has a built-in degree of fault tolerance, the system is not fault-tolerant in practice. When one of the links goes down during regular operation the system has to be rebooted to determine new routing information. In addition to the fast wormhole routing based on distributed routing tables, the Router ASIC supports low-performance source routing where each packet contains absolute routing path in its header. Source routing is used to access Router registers and for the initial discovery of the system topology.

The Router ASIC

The block diagram of the Router ASIC is shown in Figure 3.7. Each of the six full-duplex ports guarantees a reliable message delivery by separate physical, data link and message layers implemented in hardware.

The physical transmission layer is based on a pair of Source Synchronous Drivers and Receivers (SSD/SSR), which transmit 20 data bits and a data framing signal. The 200 MHz differential clock signal is sampled at both edges resulting in the effective data transmission rate of 400 MBaud. The Router core operates at 100 MHz; at each clock edge, the core provides 80 bits of data which the SSD serializes into a 20 bit stream over four 400 MHz clocks. Three router ports use SGI Transistor Logic (STL) signalling while the other three ports use differential signalling; the STL ports are used internally inside a module to connect the router board to two node boards and to another router board while the differential ports are connecting the router boards in separate modules with external cables.

The data link layer (LLP) guarantees a reliable transmission using CCITT-CRC code with a go-back-n sliding window retry mechanism [52, pp.137–144]. The basic unit of delivery in LLP is a micropacket which consists of 128 bits of data and 8 bits of sideband information. The sideband is used by the message layer to implement virtual channel tagging and message flow information. The theoretical peak link bandwidth is 800 MB/sec per direction per link.

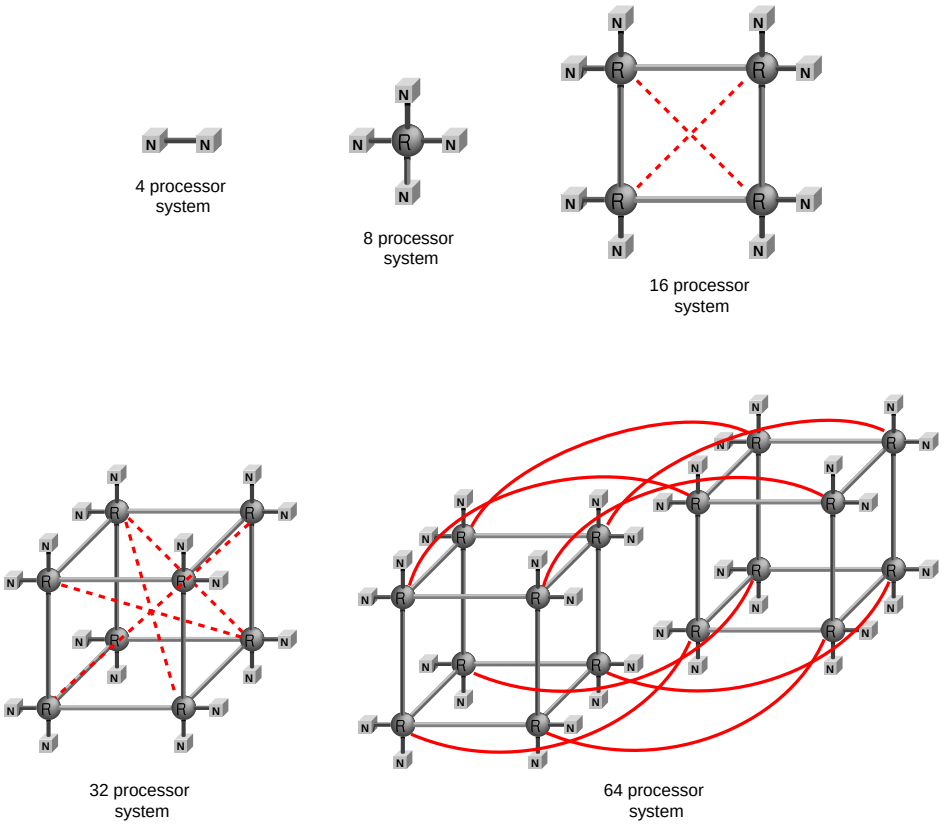


Figure 3.5: Origin 2000 topologies from 4 to 64 processors

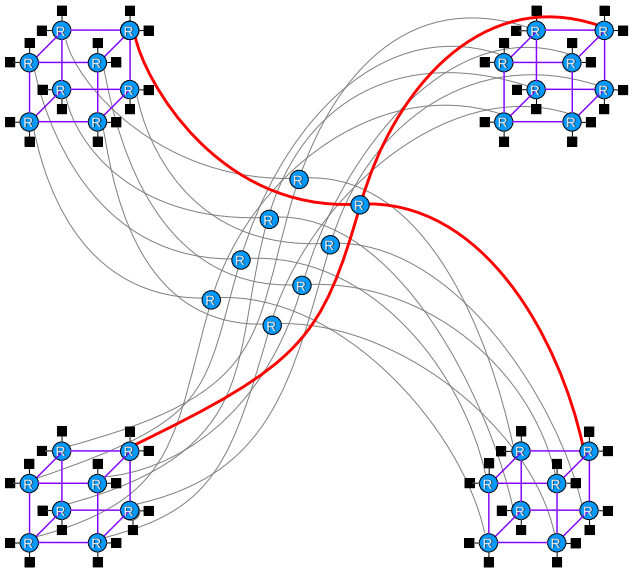


Figure 3.6: 128 processor Origin 2000 system topology

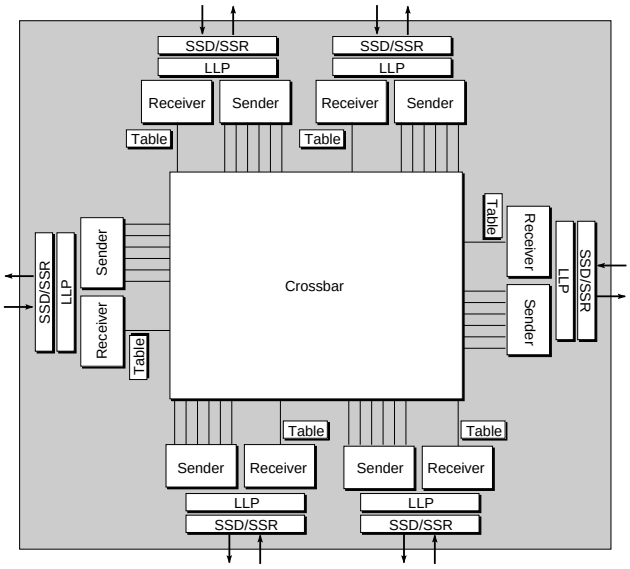


Figure 3.7: Block diagram of the Router ASIC

The message layer (Sender/Receiver) defines four virtual channels and a credit-based flow control scheme that supports arbitrary message lengths. Each message consists of a header micropacket which specifies message destination, priority and congestion control options. A header micropacket is followed by zero or more data micropackets with a stop bit designating the last micropacket in the message. Each virtual channel provides independent flow control and buffering space; the Router ASIC implements a 256-byte receive buffer for each virtual channel on each port. The buffers are organized as a set of linked lists; there is one linked list for each possible output port and each virtual channel. The linked lists are implemented as a dynamically allocated message queue (DAMQ). Each router port has a local routing table which stores a portion of the global routing information sufficient to route the incoming messages to the next router hop.

At the core of the Router ASIC is the central crossbar connecting the DAMQs in the receiver block with the inputs in each of the other five sender blocks. In each clock cycle, the crossbar arbitration mechanism chooses up to six winners by using a variant of the wavefront arbiter [53]. Each virtual channel from each port can request arbitration for every possible destination, providing up to 120 arbitration candidates on each cycle. To improve the routing latency in a lightly loaded system, the Router implements a fast message path. When there are no micropackets in the DAMQ for a given virtual channel, the micropacket may bypass the DAMQ structure entirely provided that there is no conflict for the output port in the crossbar. The bypass latency is 41 ns; when the bypass is not granted the latency increases by two router cycles to 61 ns.

3.2.4 Physical System Organization

The Origin system is physically organized as one or more modules. Several modules are connected together with external CrayLink cables to construct systems up to 64 processors. A deskside Origin 2000 system consists of a single module in a chassis. It scales from two to eight processors. Two modules may be placed in a rack and several racks may be combined together. Figure 3.8

shows a fully configured 32-processor system with four modules in two racks: the express links use external router ports to shorten the longest distance between nodes. A 64-processor system adds another four modules. The router ports used for express links in 32-processor system are used to connect modules in separate 32-processor cubes. Systems larger than 64-processors use an additional rack that holds the metarouter. As with smaller configurations, the metarouter ports are connected with CrayLink cables to other router ports. Figure 3.9 shows the physical configuration of a 128-processor system which consists of four 32P hypercubes connected with a metarouter.

Each module holds up to four nodes and two routers. The midplane board is at the center of the module: the four node boards are inserted from the front and the router boards are inserted from the back. The midplane provides a standard system clock for both the XIO and CrayLink interconnection, the STL CrayLinks between the nodes and the local routers, the XIO links for all four nodes, power distribution, system control signals and SCSI connections. The midplane holds two XBow ASICs: two adjacent node boards share a single XBow with two separate XIO ports. The XBow is the gateway to the local I/O system. The BaseIO board, present in every module, includes two SCSI interfaces, an Ethernet port and two serial interfaces; other XIO boards include a PCI bridge, ATM, FibreChannel and graphics interfaces.

3.3 The Irix Operating System

The operating system running on the Origin systems is Irix, Silicon Graphics' version of the AT&T System V Release 4 (SVR4) Unix system. The Irix kernel is a 64-bit executable. User processes can use any of the three supported application binary interfaces (ABI): backwards compatibility is maintained by the old 32-bit ABI while the new 32-bit ABI (`-n32`) and the 64-bit ABI (`-64`) use advanced features of the MIPS architecture. Irix supports symmetric multiprocessing and multitasking: internally, kernel threads are used for lightweight execution; user processes have a choice of using user-level threads created with the `sproc` system call or POSIX threads which are scheduled entirely in user space. Heavyweight processes can use any of the standard Unix interprocess communication interfaces: System V IPC primitives, POSIX semaphores, or shared memory. The MIPSpro compilers include support for automatic parallelization and OpenMP programming models. Irix also supports many of the advanced Unix features: kernel threads, journalled filesystem (XFS), asynchronous I/O and advanced scheduler features such as currency-based scheduling, gang and frame scheduling and soft realtime processes. The same Irix kernel runs on the whole range of systems manufactured by SGI, from the entry-level O_2 desktop to the largest Origin 2000 supercomputer. This section describes some of the Irix features which were introduced to support application execution in the ccNUMA environment.

3.3.1 Hardware Graph

A single Origin system can scale to hundreds of processors, routers and I/O units. Rather than using the old device interface in the `/dev` directory, Irix introduced a new, hierarchical way of organizing devices: the *hardware graph* and its associated hwgraph file system, which represents the collection of all significant hardware in the system. The hardware graph is a directed graph where each vertex represents a hardware object: processor, router, disk drive, disk partition, network controller, etc. There are some additional vertices which represent a collection of objects (e.g., all

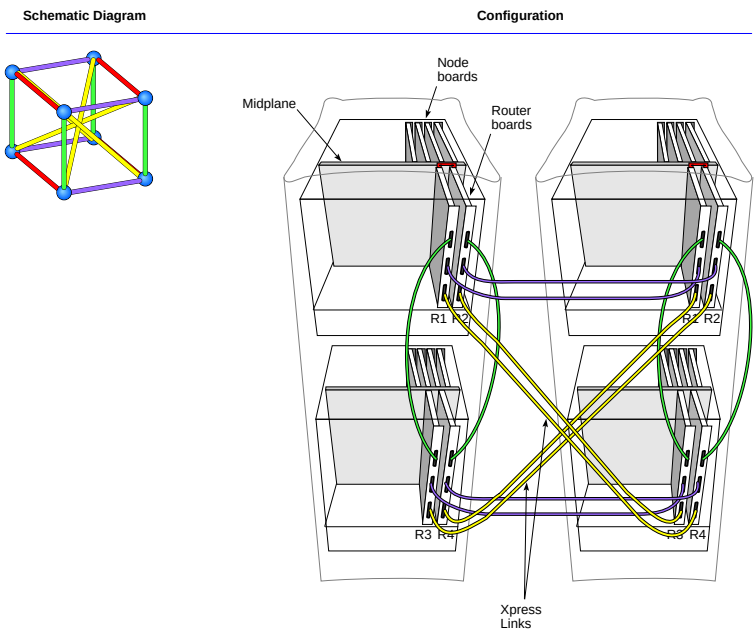


Figure 3.8: Physical configuration of a 32-processor Origin system

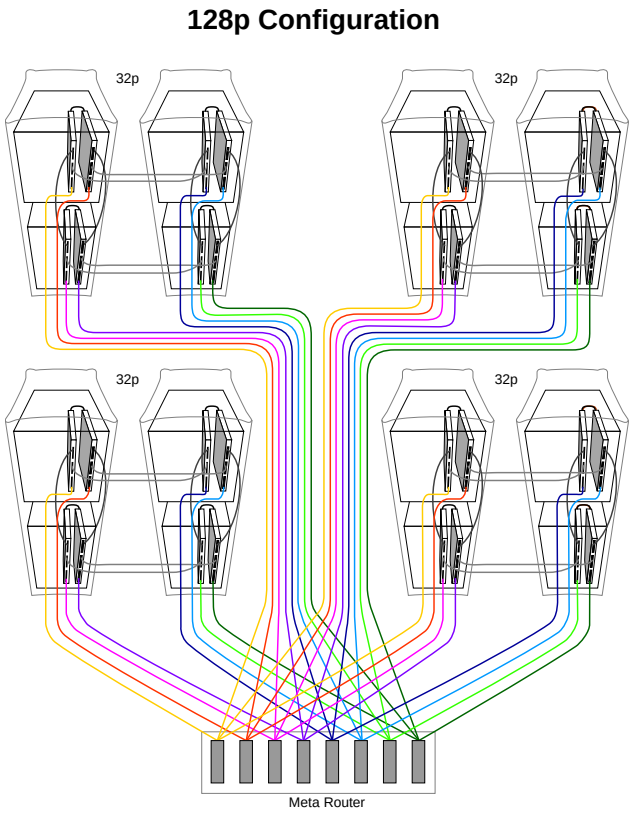


Figure 3.9: Physical configuration of a 128-processor Origin system

disks) or a grouping of hardware (e.g., a node). Labeled edges are used to connect graph vertices in a way that shows some relationship between the underlying hardware. For example, an edge labelled `memory` points from a vertex representing a node to a vertex representing the memory on that node. The hardware graph also supports arbitrarily-labeled information associated with each vertex. This allows device drivers to associate useful data structures and other information with a particular device. For example, hardware inventory labels are used by the hardware inventory program (`hinv`) to print object-specific information about objects, such as processor speed, cache size, etc.

The internal hardware graph is exported to user-level processes through a pseudo-file system. The hardware graph file system, mounted on `/hw`, represents the collection of all hardware in the system as a tree of files and directories. While the internal graph may contain cycles, these cycles are broken externally—the hwgfs file system imposes an artificial hierarchy on the hardware graph and it uses symbolic links that point "up" to higher-level directories. Users cannot create files in the hardware graph file system; rather, the creation of files and directories is allowed only to kernel drivers, which export device files by extending the hardware graph by adding new edges and vertices.

The hardware graph provides the main name space for nodes and routers in the Origin system. A name is a string of characters in the form of a path that both identifies a node and defines its location relative to the overall hardware. For example, eight nodes in a 16-processor Origin system are located in the hardware graph as follows:

```
$ find /hw -name node -print
/hw/module/1/slot/n1/node
/hw/module/1/slot/n2/node
/hw/module/1/slot/n3/node
/hw/module/1/slot/n4/node
/hw/module/2/slot/n1/node
/hw/module/2/slot/n2/node
/hw/module/2/slot/n3/node
/hw/module/2/slot/n4/node
```

The naming reflects physical organization of the system: there are four nodes in each module.

Another node name space is the *compact node identifier* space. All available nodes in the system are enumerated with small integer numbers starting at 0. This enumeration may change across reboots if the nodes are removed from the system or placed in another module. The current mapping from the compact node number (the *cnodeid*) to the full path in the `/hw` file system is defined by the hardware graph directory `/hw/nodenum`:

```
$ ls -o /hw/nodenum
total 0
lrw-------   1  26 Jul 10 13:36 0 -> /hw/module/1/slot/n1/node
lrw-------   1  26 Jul 10 13:36 1 -> /hw/module/1/slot/n2/node
lrw-------   1  26 Jul 10 13:36 2 -> /hw/module/1/slot/n3/node
lrw-------   1  26 Jul 10 13:36 3 -> /hw/module/1/slot/n4/node
lrw-------   1  26 Jul 10 13:36 4 -> /hw/module/2/slot/n1/node
lrw-------   1  26 Jul 10 13:36 5 -> /hw/module/2/slot/n2/node
lrw-------   1  26 Jul 10 13:36 6 -> /hw/module/2/slot/n3/node
lrw-------   1  26 Jul 10 13:36 7 -> /hw/module/2/slot/n4/node
```

The processors have their own name space. As with nodes, the primary name space is provided by the hardware file system. Each path name also identifies the node to which the processor is connected. Just like the compact name space for nodes, there is a corresponding *compact processor identifier* space where each processor is identified by a small integer number. The enumeration starts at 0 and the mapping from the *cpuid* to the full path is given by the hardware graph directory `/hw/cpunum`. The list of all processors in the system can be obtained as follows:

```
$ find /hw -name "[ab]" -print
/hw/module/1/slot/n1/node/cpu/a
/hw/module/1/slot/n1/node/cpu/b
/hw/module/1/slot/n2/node/cpu/a
/hw/module/1/slot/n2/node/cpu/b
/hw/module/1/slot/n3/node/cpu/a
...
```

Unlike nodes and processors, there is no compact enumeration of the routers in the system—each router can be identified only by a full path in the hardware graph:

```
$ find /hw -name router -print
/hw/module/1/slot/r1/router
/hw/module/1/slot/r2/router
/hw/module/2/slot/r1/router
/hw/module/2/slot/r2/router
```

The topology information can be deduced from the hardware graph as well. Each node has an edge called `link` that points to the router to which the node is connected. All routers store the connectivity information in the `router` directory: edges are labelled as small integers 1–6 where each integer represents an active router port. Command `topology` is a simple shell script which parses the contents of the `/hw` file system and prints all connectivity information:

```
$ topology
...
The topology is defined by:
/hw/module/1/slot/n1/node/link -> /hw/module/1/slot/r1/router
/hw/module/1/slot/n2/node/link -> /hw/module/1/slot/r1/router
/hw/module/1/slot/n3/node/link -> /hw/module/1/slot/r2/router
/hw/module/1/slot/n4/node/link -> /hw/module/1/slot/r2/router
...
/hw/module/1/slot/r1/router/1 -> /hw/module/2/slot/r1/router
/hw/module/1/slot/r1/router/4 -> /hw/module/1/slot/n2/node
/hw/module/1/slot/r1/router/5 -> /hw/module/1/slot/n1/node
/hw/module/1/slot/r1/router/6 -> /hw/module/1/slot/r2/router
/hw/module/1/slot/r2/router/1 -> /hw/module/2/slot/r2/router
...
```

Externally, the hardware graph looks like an ordinary file system; no special system calls are necessary to access files and traverse the topology. Internally, the hardware graph is accessible through a kernel API described in [39].

3.3.2 Distributed Memory Management

The distributed memory in Origin nodes places a new set of requirements on the operating system, the compiler, and the users writing applications for the ccNUMA environment. Since the memory

performance degrades if the program references memory on remote nodes, memory locality is an important goal. The virtual memory management system in Irix was extensively redesigned to support the requirements of nonuniform memory. Irix provides a rich set of features for managing memory locality, both automatically and manually. Automatic memory locality management is based on the concept of adaptability while the manual tools work based on hints provided by users, compilers, or special high level memory placement tools.

Automatic Memory Locality Management

Automatic memory locality management in Irix is based on memory replication, dynamic memory migration, and an initial placement policy based on a first touch placement algorithm. Read-only data can be replicated on multiple nodes to improve locality. By default, the executable code for the Irix kernel is replicated on every node in the system. User read-only pages are replicated according to a dynamic coverage radius algorithm. Every memory object has an associated coverage radius that defines a neighborhood of nodes that are considered to be close to a physical page associated with the memory object. When a thread incurs a page fault, the page fault handler looks for a page that contains the data needed in the memory object that covers a portion of the virtual address space where the faulted page is located. If the missing page is already in memory on a node within the coverage radius, this page is used; otherwise, a new page frame is allocated on the node where the page fault occured and the data are copied from the other page. Kernel parameters that control memory replication are described in [45].

Dynamic page migration is a mechanism that provides adaptive memory locality for applications running on the Origin systems. The Origin hardware implements a page migration algorithm based on comparing remote memory access counters to a local memory access counter. When the difference between the numbers of remote and local accesses goes beyond a preset threshold, an interrupt is generated to inform the operating system that a physical memory page is experiencing excessive remote accesses. The kernel interrupt handler decides whether a page should be migrated closer to the node where most of the remote accesses were generated, based on a number of parameters grouped in the page migration policy. Page migration is controlled either through global kernel page migration parameters or through the memory management control interface. Page migration is described in more detail in [41]. In practice, page migration is not widely used in the Origin systems. Early versions of the Irix operating system have assigned an incorrect page color to the migrated pages, substantially degrading application performance. Even though this issue was fixed in never versions of Irix, the applications written for the Origin system typically perform explicit memory locality management.

When a process touches a page of virtual memory for the first time, the operating system has to decide where to allocate the associated page frame. Irix supports several different policies for initial page allocation: the pages can be allocated on a node where the process is executing, on a fixed node, or on one of the nodes in a group using a round-robin algorithm. The vast majority of programs are single-threaded; a sensible default page allocation strategy is to place page frames on a node where the process is executing. This is the first touch allocation policy; combined with memory affinity scheduling, where the kernel attempts to schedule a process on the same node where it was running previously, this policy results in good memory behavior for uniprocessor applications.

User-Driven Memory Locality Management

The default memory allocation policies are generally sufficient for uniprocessor applications and some parallel applications using either compiler extensions (OpenMP) or parallel programming models (MPI). However, "naive" parallel applications which assume a SMP memory model may run into trouble. For example, the default first-touch page allocation policy can have a side effect of allocating all application memory on a single node, if the parallel application initializes its data in a single thread. Irix provides a *memory management control interface* [42] to allow users explicit control over memory system behavior. This interface covers both ccNUMA and generic memory system control. For ccNUMA, the interface provides control over placement, migration and replication policies; for generic memory management, the interface provides control over page size and paging algorithms. The memory management control interface can be used directly, via OpenMP compiler directives, or via high-level placement tools (`dplace`, `dlook` and `dprof`).

The memory management control interface (MMCI) is based on the specification of different kinds of policies for different kinds of operations executed by the virtual memory management system. Users are allowed to select a policy from a set of available policies for each one of these VM operations. Any portion of a virtual address space, down to the level of a page, may be connected to a specific policy via a *policy module*. Policy modules control initial allocation, dynamic relocation and paging behavior of the VM system. For initial allocation, the MMCI interface can specify what policy to use for placement, page size and what should be the fallback method. Dynamic relocation controls the level of replication and the conditions under which a page should be dynamically migrated. The paging policy controls the aggressiveness and domain of memory paging.

The *placement policy* defines the algorithm used by the memory allocator to decide what node to use to allocate a page in a multinode ccNUMA machine. The goal of this algorithm is to place memory in such a way that local accesses are maximized. All placement policies are based on two abstractions of physical memory. A *memory locality domain* (MLD) with center c and radius r is a source of physical memory composed of all memory nodes within a router hop distance r of a center node c. Normally, MLDs have radius 0, representing one single node. Several MLDs can be placed into a *memory locality domain set*; MLDsets address the issue of placement topology and device affinity. When initially created, memory locality domains are not associated with a particular node; when one or more MLDs are combined into a MLDset, the placement of the MLDset is performed based on the user-specified topology information and resource affinity. The topology can be determined by the system or specified explicitly. The resource affinity can be used to place the MLDset to a particular hardware device (e.g., a disk or a graphics pipe). After the creation of MLDs and placement of MLDsets, the application provides hints to the scheduler to run threads close to where the memory is allocated.

Figure 3.10 shows an example of how the memory management control interface is used to place processes and memories. The application on the left consists of four processes that share a single address space. Four processes require placement on at least two nodes. The example also assumes that each thread accesses 90% of the data exclusively while the remaining 10% is shared with the neighboring threads. The application also uses a graphics pipe, and it wants the threads to be placed into an one-dimensional cube. Two memory locality domains are created to control memory allocation for the two nodes. The MLDs are then grouped into a MLDset. The operating system could decide to place the MLDset anywhere in the system. However, when placing the

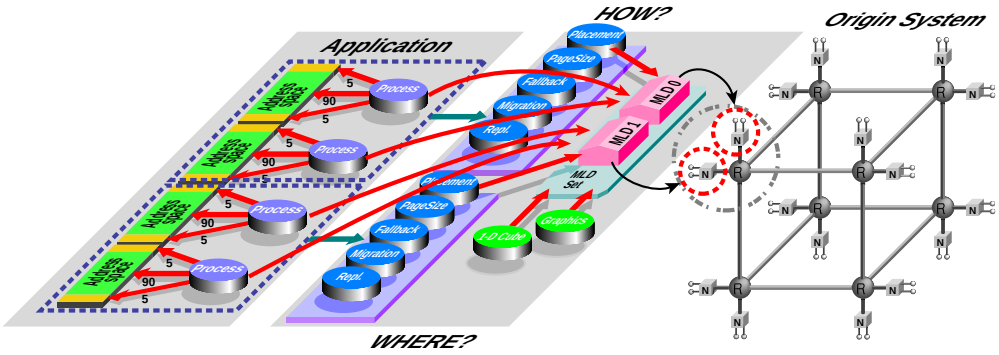


Figure 3.10: Irix memory locality management example

MLDset the application requests proximity to a graphics pipe. The operating system chooses two nodes in the upper left corner of the 32-processor hypercube and associates the MLDs with two nodes there. Finally, the application provides hints to the scheduler by using the first (now placed) MLD for the first two threads and the second MLD for the other two threads. The scheduler will run threads on the two processors on each of the two nodes while the VM system will allocate memory on local nodes. The policy modules associated with portions of the virtual address space will also control other aspects of memory management, such as whether to give preference to locality or desired page sizes.

Chapter 4

Architectural Evaluation of the Origin 2000

Microbenchmarks are used to evaluate basic system parameters such as the speed of various processor operations, memory latencies and bandwidths. On an SMP system, there are only a few basic numbers that characterize the system: memory latency, processor, memory, and bus bandwidth; these parameters are independent of the processor and memory locations in the system. On a ccNUMA system, these values are dependent not only on the locations of the CPU and memory, but also the type of coherency transaction. For example, accessing an unowned cache line will have a much lower cost than pulling the cache line out of a remote processor's cache.

Various microbenchmarks measure memory latencies and bandwidths: the STREAM benchmark [27], the memory latency kernel in the lmbench suite [28], and the microbenchmark used by Hristea et al. [19]. However, there are several problems when trying to use these codes to systematically evaluate a large Origin 2000 system. First, these are all separate executables with different modes of invocation; integrating them in a common test harness would be hard, if not impossible. Second, with the exception of Hristea's restart benchmark, the other benchmark codes were not written with multiprocessor systems in mind. The STREAM benchmark can be modified to use OpenMP parallel loop constructs. However, placing threads and memory into the desired configuration is quite a challenge. Third, none of the existing benchmark codes are capable of generating all coherency transactions.

The snbench benchmark suite was written from scratch to help evaluate ccNUMA latencies and bandwidths. It had to satisfy several goals: first, it should be capable of measuring latencies and bandwidths of all Origin 2000 coherency transactions generated by the processors. Second, snbench should be capable of placing threads and memory on arbitrary nodes in the system. Third, it should be possible to use the benchmark as a part of a larger script that would first evaluate system configuration and then generate a list of experiments which aim to measure the complete set of ccNUMA memory characteristics. Fourth, accuracy was considered more important than portability. The development of snbench revealed several not-so-obvious aspects of the Origin directory cache coherence protocol, design tradeoffs and the idiosyncracies of the Irix operating system.

This chapter is organized as follows: Section 4.1 describes the interaction of cache and directory coherence protocols, and how processor actions translate into directory requests. Section 4.2 gives an overview of the snbench implementation. Finally, Section 4.3 presents data gathered on a variety of Origin systems.

4.1 Protocol Transactions and Coherence States

4.1.1 Composite Cache Coherence State

The basic unit of coherence in the Origin system is a 128-byte cache line. Inside the processor, the cache line is subdivided into 64-byte blocks in the primary instruction cache and 32-byte blocks in the primary data cache. The unified secondary cache operates on 128-byte memory blocks. The processor uses a four-state MESI protocol [30] to manage the cache coherence of the secondary cache. Each block can be in different MESI states in different processor's caches. Additionally, the home node keeps the directory state of all blocks representing a portion of the total system memory which resides on that node. Conceptually, the composite cache state of each 128-byte block is a vector containing its directory state (maintained by the home node) and its MESI state (maintained by all processors in the system). For a system with n processors, the composite cache state vector for an uncached memory block is $\{\texttt{UOWN}, \texttt{I}^n\}$. A read-only copy of a memory block replicated in one or more processor's caches is described with a composite vector of $\{\texttt{SHRD}, (\texttt{S}|\texttt{I})^n\}$. A modified memory block present in the first processor's cache is described as $\{\texttt{EXCL}, \texttt{MI}^{n-1}\}$. In practice, however, only a small subset of the total system memory is present in processor's caches. Clearly, the set of valid composite cache state vectors is only a small set of all possible permutations.

A processor generates a memory request when it encounters an access fault. A *read-access fault* happens when the requestor does not have a valid copy of the cache line in its cache (i.e., its MESI state in requestor's cache is invalid). Both load and store instructions can trigger a read-access fault. A *write-access fault* happens when the requestor attempts to retire a store to a cache line whose MESI state in requestor's cache is shared. In this case, the requestor needs to obtain the exclusive ownership of the cache line (by means of an upgrade transaction) before it can retire the store. At the time of an access fault, the only possible MESI states in requestor's cache can be I or S. Read- and write-access faults trigger transactions on the SysAD bus. The only other MESI state transition that results in a SysAD bus transaction happens when the processor writes back a modified cache line that was displaced by another cache line. However, this transaction is a side-effect of a read-access fault encountered by the processor. Writebacks can be avoided by limiting the size of the test area to fit in the processor's secondary cache. Writebacks can be forced on every read-access fault by modifying all lines in the secondary cache beforehand. Other MESI transitions are not visible externally.

The composite cache coherence states are used by snbench to describe a combination of the directory state of a cache line, the vector of its MESI states in remote processor's caches, and (in some cases) the resulting directory protocol transaction type, *as seen by the local processor when it encounters an access fault*. Table 4.1 shows composite cache coherence states used by snbench. The *local* and *remote* columns list possible MESI states in local (requestor) and remote processor's caches. The *directory* column is the directory state of the cache line. The last column gives the resulting directory transaction (discussed in Section 4.1.3).

The UOWN composite state represents two cases: when the home node keeps the only copy of the cache line (i.e., the line is not cached anywhere in the system), or when the local processor requests a cache line after it has silently dropped the line from its cache due to a conflict miss. The Origin directory protocol allows a processor to silently drop an E line from its cache without notifying the home node. The home node keeps the identity of the current owner as part of the cache

state	*local*	*directory*	*remote*	*directory transaction*
UOWN	I	UOWN|$\texttt{EXCL}_{me}$	I	unowned
SHRD	I|S	SHRD	S	unowned, invalidate
CEXH	I	$\texttt{EXCL}_{other}$	E	intervention (clean-exclusive hit)
CEXM	I	$\texttt{EXCL}_{other}$	I	intervention (clean-exclusive miss)
DEXD	I	$\texttt{EXCL}_{other}$	M	intervention (dirty-exclusive downgrade)
DEXT	I	$\texttt{EXCL}_{other}$	M	intervention (dirty-exclusive transfer)

Table 4.1: Snbench composite cache coherence states

line directory state. The line can be returned immediately if the dropped line is requested again by the current owner. (The Origin directory cache coherence protocol uses the $\texttt{EXCL}_{me}$ pseudo-state to indicate this.) Since the directory protocol transactions are the same for both unowned and silently-dropped lines, a single snbench composite state covers both cases.

The cache line is in the SHRD state when one or more processors keep a read-only copy of the cache line in their caches. Remote processors (if any) keep the line in the S state. In the the local processor's cache, the line is in the I state on read-access faults, and in the S state on write-access faults.

The directory protocol uses the $\texttt{EXCL}_{other}$ pseudo-state to describe a case when a local processor requests a copy of a cache line whose current owner (at the directory level) is a remote processor. In the remote processor's cache, the cache line could be in either E, I, or M state. When the cache line is in the E state, the remote processor keeps a clean-exclusive copy in its cache; the CEXH composite state is used to describe this scenario. When the cache line is in the I state, the remote processor has silently dropped the line from its cache; the CEXM state is is used in this case. In both cases, the cache line has not been modified and the home node can supply a valid copy to the requestor. Two different clean-exclusive states were introduced to evaluate the impact of the R10000/R12000 secondary cache controller, which is located on-chip, while the L2 tags and data are located off-chip and they are accessed via a separate L2 cache bus.

When the remote processor has a modified copy of the cache line, the directory state of that line is $\texttt{EXCL}_{other}$ and the MESI state in remote processor's cache is M. The home node does not have a valid copy of the cache line. The directory protocol changes the ownership of the cache line from the remote to the local processor. If the local processor requests an exclusive copy, the remote processor sends its modified copy directly to the local processor and purges the line from its cache. In this case, the home node does not have to be updated because the local processor becomes a new owner (this requires that the local processor keep the line in M state). This transaction is represented with the DEXT composite state. If the local processor requests a shared copy, the remote processor downgrades the line to S. However, the cache line is now shared at the directory level and the home node needs to have an updated copy of the cache line as well. Therefore, the remote processor needs to send two messages carrying the modified cache line, one to the requestor, and another to the home node. This transaction downgrades the state of the cache line from dirty-exclusive to shared and it is represented with the DEXD composite state. The distinction between these two states was made because the latency and bandwidth results (presented later in this chapter) indicate a substantial difference between dirty-exclusive downgrades and dirty-exclusive ownership transfers.

4.1.2 Processor Actions and Protocol Requests

The MIPS processors use load and store instructions to transfer data between registers and memory. Additional prefetch instructions may be used to place data in the cache before it is needed to hide memory latency. When the processor encounters an access fault, an external SysAD request is issued by the processor. This request is translated by the Hub processor interface (PI) into a directory protocol request. When the directory protocol transaction completes, the PI generates a reply on the SysAD bus. Only then can the processor-initiated action complete. Table 4.2 shows how various processor actions translate into directory protocol requests.

action	*request*	*result*
load miss	READ	get a shared or exclusive copy
store miss	RDEX	get an exclusive copy
upgrade	UPGRD	invalidate other sharers
writeback	WB	write a modified line to memory
read prefetch	RDSH	get a shared copy
write prefetch	RDEX	get an exclusive copy
instruction fetch	RDSH	get a shared copy

Table 4.2: Processor actions and protocol requests

Load instructions which miss in the secondary cache are translated into READ directory protocol requests. The home node returns an exclusive copy of the cache line when the line is in the UOWN state. If the line is in the SHRD state, the requestor is simply added to the list of sharers. If the line is owned by a remote processor, its state is downgraded to SHRD. The READ semantics is intended to speed up the performance of uniprocessor applications. By returning an exclusive (instead of a shared) copy, the processor can modify the cache line without having to obtain exclusive ownership first.

There are two cases when a store instruction cannot be immediately retired. If the line is not present in the secondary cache, the processor needs to obtain an exclusive copy. The Hub translates a store that missed in the cache into a RDEX request. If the store hits in the secondary cache but the processor has a read-only (shared) copy, an upgrade request is placed on the SysAD bus. SysAD upgrade requests are converted into UPGRD directory protocol requests. The difference between a RDEX and an UPGRD request is that in the later case, the processor already holds a valid copy of the cache line in its cache. The RDEX reply carries the cache line data but the UPGRD reply does not.

The R10000 processor supports four flavors of prefetch instruction. The MIPSPro compiler uses read prefetch instructions for right-hand side operands and write prefetch instructions for left-hand side operands. In other words, a read prefetch is used for an operand that requires a load, whereas a write prefetch is used for an operand which is the target of a store. The SysAD bus requests can distinguish between read and write prefetch requests. A read prefetch is translated into a RDSH directory protocol request while a write prefetch is translated into a RDEX request.[1]

[1] A side effect of write-prefetching a cache line on the R10000 is that the processor immediately places the cache line into the modified state. [56]

The other two flavors of the prefetch instruction place a cache line in a particular set of the two-way set-associative secondary cache and they are not used by the compiler.

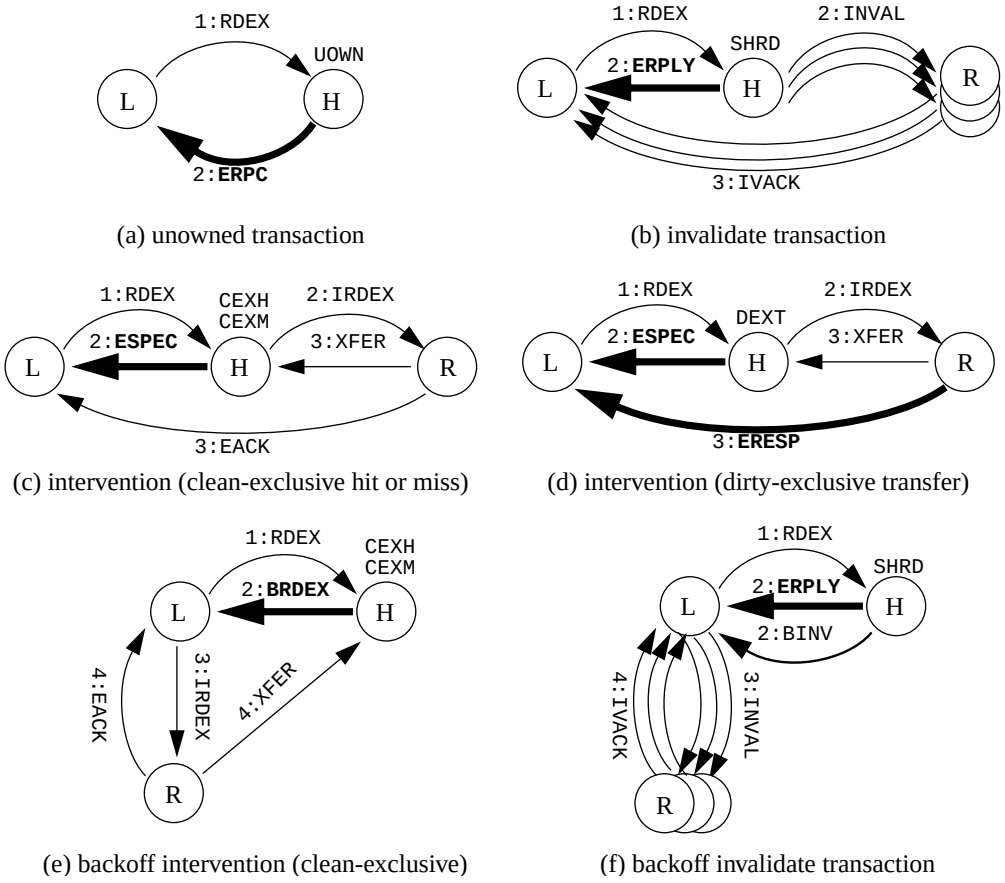
The semantics of load and of read prefetch are subtly different. Load requests are translated into READ requests whereas read prefetches are translated into RDSH requests. When the requested line is in the UOWN state, the READ request returns an exclusive copy while the RDSH request returns a shared copy. If the processor tries to write to a read-prefetched cache line, it encounters a write-access fault and the cache line needs to be upgraded from the shared to the exclusive state. On the other hand, an exclusive line obtained with a load can be upgraded from the exclusive to the modified state immediately, without incuring a penalty of an external upgrade request. Additionally, if the read-prefetched line is silently dropped from the cache and then becomes a target of a store instruction, the resulting RDEX request finds the read-prefetched line in the SHRD directory state. However, the directory maintains a list of sharers on a per-node basis; read-exclusive requests cause an invalidation to be sent to the other processor on the node (and thus increase latency). This anomaly impacts uniprocessor applications that use large arrays first as source and then as destination (the STREAM benchmark uses arrays in such a fashion). The difference between load and prefetch instruction semantics is discussed further in Section 6.2.

4.1.3 Directory Protocol Transactions

In the Origin cache coherence directory protocol, the same request can trigger different protocol transactions, depending on the directory state of the requested cache line, its MESI state in remote processor's caches, and the saturation of various message queues. For example, Figure 4.1 shows six (out of seven) directory protocol transactions triggered by a read-exclusive request. The local processor is marked as L, the home node as H, and the remote processor as R. The snbench composite state of the cache line is shown above the home node. Protocol messages are represented with arrows of different thickness to indicate the size of the message in 128-bit packets, the basic unit of transfer over the Origin interconnect network. Thin lines are requests and replies that require only one 128-bit packet. Thick lines carry the contents of the cache line in addition to the protocol message and consist of nine 128-bit packets.

The first transaction diagram is the *unowned transaction*: it consists of the RDEX request message followed by the ERPC reply from the home node. This is the most basic transaction; its name is derived from the unowned directory state (hits on unowned lines will always result in a two-step transaction). Other combinations of a request and directory state that also result in a two-step transaction include a READ request on a line in the SHRD or EXCL_{me} state, or a WB request on a line in EXCL_{me} state. This transaction involves only the requestor and the home node; in the absence of resource contention, the performance of the unowned transaction depends only on the distance (in network hops) between the requestor and the home node.

When several processors have a read-only copy of the cache line and one of them wants to obtain exclusive ownership, the read-only copies in remote processor's caches have to be invalidated before the requestor can retire the store instruction which triggered the write-access fault. Case (b) shows the resulting *invalidate transaction*. This is a three-step transaction involving the local processor, the home node, and one or more sharer nodes. The Origin directory protocol keeps track of sharers with a presence bit vector on a per-node basis. The home node sends INVAL messages to all sharer nodes whose presence bit is set. The home node also supplies the up-to-date copy of the cache line to the requestor with the ERPLY message. When the Hub on a sharer node receives the



(a) unowned transaction

(b) invalidate transaction

(c) intervention (clean-exclusive hit or miss)

(d) intervention (dirty-exclusive transfer)

(e) backoff intervention (clean-exclusive)

(f) backoff invalidate transaction

Figure 4.1: Directory protocol transactions generated by a RDEX request

INVAL message, the processor interface generates an external invalidate transaction on the SysAD bus for both processors on that node, even though only one processor may hold a cached copy of the line. When both processors respond to the external invalidate request, the Hub sends an invalidation acknowledge (IVACK) to the requestor. The R10000 processor relies on strict memory consistency; the requestor needs to collect IVACKs from all sharer nodes before the store can be retired. The Origin interconnect network does not have a broadcast ability. The unicast INVAL messages are generated sequentially in the network interface at the home node. Once they are injected in the network, the INVAL messages traverse the network and are acted upon by the sharer nodes in parallel. The performance of the invalidate transaction depends on the distance between the requestor and the home node, the longest distance between the home node and the sharers, the longest distance between the sharers and the requestor, and the number of sharers that need to be invalidated.

When the local processor requests a copy of a cache line which is owned by a remote processor, the directory protocol executes one of the several types of *intervention transactions*. In Figure 4.1, diagrams (c) and (d) show two possible intervention scenarios. In both cases, when the RDEX request reaches the home node, the directory state of the cache line is EXCL and the owner of the cache line is the remote processor (this is denoted by the EXCL_{other} directory pseudo-state). The home node sends a speculative reply (ESPEC) to the requestor, and an intervention-exclusive

request (IRDEX) to the remote processor. The home node also changes the directory state of the cache line from EXCL to BUSYE, temporarily locking the cache line until the transaction is completed. The scenarios differ depending on the MESI state of the cache line in remote processor's cache.

If the remote processor has a clean copy, or if the processor has silently dropped the cache line, the local processor can use the data it received in a speculative reply. The remote processor sends the EACK acknowledgement to the requestor, and the XFER directory revision message to the home node. The requestor completes the transaction when it receives the acknowledgement. The home node changes the cache line state back to EXCL (with the local processor as the new owner) when it receives the XFER message. Diagram (c) illustrates such a *clean-exclusive intervention* transaction. Even though snbench uses two different composite cache states, CEXH and CEXM, for clean-exclusive transactions, the directory protocol transaction is the same in both cases. The only difference is that the MESI state of the cache line in remote processor's cache is E in the former case, and I in the later. The performance of clean-exclusive intervention transactions depends on the distance between the participating nodes, and the SysAD timing when the local and remote processors reside on the same node.

If the remote processor has a modified copy of the cache line, the cache line data sent with the speculative reply is invalid. The remote processor sends an up-to-date copy to the requestor with the ERESP response and the requestor discards the speculative data. The remote processor also sends the XFER directory revision message to the home node, which concludes the transaction. This scenario, illustrated in diagram (d) in Figure 4.1, is an example of the *dirty-exclusive ownership transfer*. If the local processor requested a shared copy of the cache line (by using the READ or RDSH directory protocol request), the resulting *dirty-exclusive downgrade* (not shown in Figure 4.1) is similar to the ownership transfer, except that the final state of the cache line at the directory level is SHRD instead of EXCL, and the remote processor sends the SHWB (sharing write-back) instead of the XFER directory revision message to the home node. However, the SHWB message carries cache line data whereas the XFER message does not. In snbench experiments, the DEXT composite state indicates dirty-exclusive ownership transfers while the DEXD state indicates dirty-exclusive ownership downgrades. Just like clean exclusive transactions, the performance of dirty-exclusive transactions depends primarily on the distances between the three participants. However, the speculative reply, discarded by the local processor, wastes network bandwidth and possibly local processor's SysAD cycles. Also, the fact that the remote processor needs to send two copies of the cache line instead of one has an impact on the performance of the dirty-exclusive downgrade, especially when the local and remote processors are located on the same node.

The unowned, intervention, and invalidate transactions are the basic transaction types in the Origin directory protocol. The protocol supports two additional transaction types, variants of the intervention and invalidate transactions. When the home node detects the possibility of a deadlock, the Origin directory protocol reverts to a strict request/reply protocol. Instead of forwarding the intervention or invalidate requests, the home node sends a reply to the requestor instructing it to perform the intervention or invalidate by itself. In Figure 4.1, an invalidate transaction (b) becomes a *backoff invalidate* transaction (f). In addition to the cache line data sent with the ERPLY message, the home node sends the backoff-invalidate (BINV) message to the local node which carries the sharer presence bit vector. The local node then sends out individual INVAL messages to the sharers and waits until it receives all the IVACK replies. In the same manner, a clean-exclusive interven-

tion transaction (c) becomes a *backoff clean-exclusive intervention* transaction (e). The home node sends the BRDEX instead of the ESPEC message, which includes the identity of the current owner, as well as the speculative copy of the cache line. The local node then sends the IRDEX request to the remote node and waits until the remote responds with the EACK. If the remote processor has a modified copy of the cache line, it would respond with the ERESP message, resulting in a *backoff dirty-exclusive transfer*. Similarly, if a shared copy of the cache line is requested, the transaction would become a *backoff dirty-exclusive downgrade*. The backoff interventions and invalidations are four-step transactions. Their performance is worse than their non-backoff counterparts. Backoff transactions are hard to generate reliably because they require that the output queue at the home node is almost full. The results from Chapter 6 indicate that it requires at least seven or more processors continuously invalidating cache lines on a single home node to detect a small percentage of backoff invalidate transactions. Backoff interventions are even harder to generate reliably. While snbench can be used to generate backoff transactions, a separate tool is required to measure the fraction of backoff transactions. Section 6.3 presents some experimental results.

4.2 Snbench Implementation Overview

The snbench suite consists of a C program and a set of Perl scripts. The snbench executable takes care of thread and memory placement and executes experiments that measure memory latencies and bandwidths. The snbench executable can be used either standalone or in combination with a front-end Perl script. This script determines system topology and generates a shell script combining several snbench invocations with different parameters designed to evaluate local memory performance, remote latencies and bandwidths, and the characteristics of intervention and invalidate transactions. When this shell script is executed, the output is captured in a file; another Perl script parses the output files, extracts data, and prints desired values in tabular form.

The coherence transactions discussed in previous sections involve three participants: the local processor that issues requests, the home node, and zero or more remote processors. The snbench executable allocates a pool of memory on the home node and then spawns one or more local and remote threads. The local threads allocate the memory for latency and bandwidth experiments from this pool. The remote threads touch or modify the allocated memory to place the cache lines in the desired state; the local thread executes one of the latency or bandwidth measurement kernels. The main thread waits until all local threads complete before printing results.

Local threads execute one or more iterations of the microbenchmark and measure the time it takes for each iteraction. Local threads are sometimes called requestor threads because they are the originators of protocol requests: cache misses and writebacks caused by requestor threads are translated into protocol requests directed at the home node. Each local thread may use one or more remote threads to place cache lines in the desired cache state on the remote processor before each iteration of the test. At the end of execution the main thread computes and prints minimum, maximum, median, average and standard deviation for test results generated by each local thread. All local threads are synchronized to start executing simultaneously. Local and remote threads are bound to a particular processor specified by the command line parameter. Threads can be bound to a node as well; in this case, the operating system will schedule the thread on any processor on the designated node.

4.2.1 Measuring Memory Bandwidth

The snbench suite is capable of measuring several types of memory bandwidth. There are five different bandwidth experiments, each designed to generate a different stream of protocol requests. All experiments are based on two kernels, a simple reduction loop and a fill loop, shown in Figure 4.2.

A fixed stride of 16 double words (128 bytes) was chosen to reduce the loop overhead and to eliminate reuse of the data in secondary cache lines; each pipelined access will hit on a different cache line. The kernels are actually written in Fortran instead of C. They are compiled with the highest level of compiler optimization; the compiler uses software pipelining and unrolls the loops to achieve the best memory performance. The reduce loop actually comes in two versions, one using prefetch instructions and the other using loads. The prefetched reduce loop generates RDSH requests, and the nonprefetched loop generates READs. Table 4.3 summarizes the bandwidth experiments.

experiment	*kernel*	*request*
bw-read	sum	READ
bw-rdsh	sum	RDSH
bw-rdex	fill	RDEX
bw-zero	fill	RDEX, WB
bw-upgrd	fill	UPGRD
bwmp-read	sum	READ

Table 4.3: Memory bandwidth experiments

The bw-read and bw-rdsh experiments use the reduce loops to evaluate memory bandwidth. The bw-rdsh kernel is compiled with prefetch instructions enabled; the compiler will schedule read prefetch instructions before the loads that compute the sum; this results in lines requested through RDSH directory transactions. The bw-read kernel is compiled with prefetch instructions disabled; the loads issued by the processor will be translated into READ requests.

The bw-rdex and bw-zero experiments are similar: both use the fill kernel to modify cache lines in the test array. The store instructions are translated into RDEX directory requests; however, when the array size exceeds the size of the L2 cache, each store will displace a dirty line and generate a WB request. The bw-rdex experiment limits the size of the test array to no more than the size of processor's L2 cache; this guarantees that no writebacks are generated—dirty lines stay in the cache after the end of the experiment. The bw-zero experiment uses array sizes larger than the L2 cache size. Additionally, the experiment modifies a part of the array before it starts measuring elapsed time. The size of the initially modified array is the same as the L2 cache size; this step ensures that all L2 cache lines are dirty. When the timed section of the loop is executed, each store will displace a dirty cache line, and generate one WB and one RDEX request.

Unlike the bandwidth experiments described so far which operate on cache lines in any state, the bw-upgrd experiment can operate only on lines in SHRD state. The local thread places a read-only copy of the test array in its L2 cache, as do sharer threads running on other nodes. When the local thread executes the fill loop, the store instructions hit on shared lines in the processor's L2 cache, resulting in UPGRD protocol requests. The test array is limited to the size of a local

processor's L2 cache; this ensures that all lines are placed in the local processor's cache and the subsequent store instructions will be sent out as UPGRD requests. We expect that the resulting bandwidth should be higher than either bw-rdex or bw-zero because the upgrade transactions do not carry cache line data.

All bandwidth experiments can have more than one local thread timing the benchmark kernel. However, the local threads are synchronized only at the start. While they will start the clock at the same time, they may end at different times. This poses a problem when we want to measure the total bandwidth of a common system resource, such as the SysAD bus, the memory controller, or a shared link. Initially, all threads participate in saturating the resource. If the bandwidth of the shared resource is not evenly distributed across threads, they will take different amounts of time to read or modify cache lines in their portion of the test array (all threads operate on arrays of equal size). The bwmp-read experiment is used to solve the problem of local threads terminating at different times. It is similar to the bw-read experiment; however, the first thread that finishes processing its portion of the test array signals all other threads to stop the execution. The shared resource is evenly utilized, and its effective bandwidth can be computed by adding the bandwidths reported by individual threads. The bwmp-read experiment was used to measure effective bandwidth of the SysAD bus, the memory interface and the interconnect links.

4.2.2 Measuring Back-to-Back Latency

The first set of latency experiments is similar to the memory latency benchmark in McVoy's lmbench suite [28]. There are three back-to-back latency experiments, all based on the idea of chasing a linked list of pointers. A simplified version of the back-to-back measurement code is shown in Figure 4.3. Function init constructs a circular list of n elements, where each element is separated by $stride$ bytes. Functions walk and modify traverse the circular list and generate READ or RDEX requests, respectively. The back-to-back latency is computed as the time to traverse the entire list divided by the number of elements in the list.

Table 4.4 lists back-to-back latency experiments. The lmbench-read experiment uses the walk function to traverse the list. Each load is dependent on the result of the previous load, which guarantees that no two memory accesses overlap. Experiments lmbench-rdex and lmbench-upgrd traverse the list with the modify function. This function is similar to walk; however, the first access to the cache line is a store, not a load. Instead of a READ, the Hub will issue a RDEX or an UPGRD request, depending on the state of the cache line in local processor's cache.

experiment	*kernel*	*request*
lmbench-read	walk	READ
lmbench-rdex	modify	RDEX
lmbench-upgrd	modify	UPGRD

Table 4.4: Back-to-back memory latency experiments

Just like the bw-upgrd experiment, the lmbench-upgrd experiment is meaningful only for lines in the SHRD state. The local and sharer threads first touch all cache lines where the circular list is stored, bringing them into the L2 cache in shared state. The local thread then measures the

```
    double sum(double a[], int n)
    {
        double s = 0.0;

        for (i=0; i < n; i += 16)
            s += a[i];

        return s;
    }

    void fill(double a[], int n)
    {
        for (i=0; i < n; i += 16)
            a[i] = 0.0;
    }
```

Figure 4.2: Kernels used in bandwidth experiments

```
struct line {
    struct line* next;
    double       dummy;
};

void init(char* array, int stride, int n)
{
    struct line* p = array;

    while (--n > 0) {
        p->next = (char*) p + stride;
        p = p->next;
    }
    p->next = array;
}

void walk(struct line* p, int steps)
{
    while (--steps >= 0)
        p = p->next;
}

void modify(struct line* p, int steps)
{
    while (--steps >= 0) {
        p->dummy = 0.0;
        p = p->next;
    }
}
```

Figure 4.3: Kernels used in back-to-back latency experiments

execution of the `modify` kernel. Since the first access to the line is a store, the line needs to be upgraded to the exclusive state with the `UPGRD` request. In the local thread, the size of the circular list in the experiment is limited to the size of the L2 cache to avoid writing dirty lines back to memory.

Two techniques are used to eliminate cache line reuse. First, each element is visited at most once, even though the list is circular. Both the R10000 and R12000 use a random replacement strategy for the L2 cache; it is possible for a data item to stay in the two-way set-associative cache if the array does not fit in the cache. Second, the cache contents are invalidated between iterations of the experiment. We used R10000 hardware performance counters to verify that the kernels generate the expected number of cache misses.

The code fragments in Figure 4.3 are written in pseudo-C. The functions are actually manually unrolled 32 times and written in MIPS assembly, not just to ensure the highest possible accuracy but also to avoid compiler optimizations. For example, the `modify` kernel written in C would never result in the desired assembly code because the compiler would promote the load before the store.

4.2.3 Measuring Restart Latency

The back-to-back latency discussed in the previous section gives a worst-case latency estimate. If there are instructions following a load that do not miss in the cache or do not generate memory accesses, the transfer of the remainder of the cache line could effectively be hidden from the application. The restart latency gives the best-case latency estimate. If there is no contention for shared resources, the pipeline can be restarted much faster.

Conceptually, the restart latency is computed by creating a series of `walk` functions similar to the one shown in Figure 4.3. Each function in the series adds more work instructions following the load (i.e., the pointer dereference). These work instructions must be dependent on the data returned by the load, but they should not generate any memory traffic. The algorithm works by increasing the amount of processor-only work on each iteration until it hits a threshold where the dummy work following the load takes longer than the transfer of the rest of the cache line into the L2 cache. After the threshold is found, the algorithm computes the time it takes to execute only the work instructions (done by constructing a one element circular list and having the function spin in the cache). The restart latency is computed by subtracting pure work time from the back-to-back time. To ensure accuracy of the restart benchmark, the increase in the amount of work on each iteration should be as small as possible. It is very hard to achieve this goal when the kernel is written in a high-level language such as C. The code becomes dependent on the version of the compiler and the optimization levels; even then, it is not possible to use loops for idle work because all loops need branches and branches are not predictable. The work steps must be added manually and the loops unrolled by hand. We solved this problem by writing a Perl script that generates a series of `walk` and `modify` kernels in assembly language. Each successive kernel has an additional add instruction following the load; each add increases the work by one processor cycle. The back-to-back experiments use the first pair of functions in this series where no work instructions are following the loads.

Figure 4.4 illustrates the difference between the back-to-back and restart latency. The x-axis indicates the number of dependent work instructions following the load. The y-axis is the time (in processor cycles) required for each step. The samples were taken on a system with 250 MHz

R10000 processors. The 4 MB secondary cache bus was synchronous with processor core.[2] The SysAD bus was running at 100 MHz. The plots labelled L1 and L2 are both linearly increasing with L2 slightly above L1. The L1 plot starts at two core clock cycles for loads only; each add instruction increases the timing by one clock cycle. The L2 plot starts at 8 cycles and increases at one core cycle per add. This confirms our expectation that a load satisfied in the secondary cache takes six cycles. The back-to-back latency plot starts with a flat line which covers the section where the back-to-back latency dominates the total work time; all add instructions following the load are executed before the L2 bus completes transferring the cache line from SysAD into L2. After the first spike (which is due to L2/SysAD timing), the back-to-back plot monotonically increases in pronounced jumps. The jumps reflect the timing needed to synchronize processor core and SysAD bus timings; each jump is five core clocks (two SysAD bus cycles). The restart plots the difference between back-to-back and L1 plots.

The Impact of L2 Cache Bus Speed

The processor core and its secondary cache bus do not have to run at the same frequency. In fact, the 250 MHz R10000 systems are the only ones with synchronized core and L2 cache frequencies. All other Origin systems use a $3:2$ core to cache bus clock ratio.

When the core and cache bus run at different speeds, the data from a cache miss satisfied in the secondary cache will not always be returned in the same number of processor cycles. For example, on a 195 MHz R10000 with the 130 MHz secondary cache bus speed, the data will be returned in either 8, 9 or 10 processor cycles. Figure 4.5 shows latency plots for such system. Instead of a smooth line in Figure 4.4, the L2 hit curve is now a step function that increments by three cycles every three work steps. The back-to-back plot shows the effects of L2 cache bus superimposed on the SysAD bus timing. At 25 work steps, the back-to-back latency jumps from 93 to 97 processor cycles, drops back to 93 cycles at 26 instructions and climbs back to 97 cycles at 27–30 instructions. From then on, we can consistently observe two SysAD cycle latency increase every additional six instructions, with a drop to the previous level on the second sample in the six instruction cycle.

The Impact of R12000

The difference between back-to-back and restart latencies is quite significant. Hristea et al. [19] report 140 ns or about 30% of the back-to-back-latency. Recall that the increased latency of back-to-back loads is due to the L2 bus being occupied while the rest of the cache line is transferred from SysAD to L2. While the L2 bus is busy, the processor cannot access L2 tags to determine whether the next load missed in L2 cache and thus cannot issue the next request on the SysAD. To minimize this gap, the designers of R12000 changed the cache line transfer algorithm by transferring the 128 byte cache line in four 32 byte blocks; after each block, the R12K can preempt cache line transfer to perform a L2 tag check.

This modification reduced the latency of back-to-back loads. It also changes the latency plots. Figure 4.6 shows latency plots for a 300 MHz R12000 system. The secondary cache bus runs at 200 MHz and the SysAD at 100 MHz. The L1 plot is the same as before: it starts at two cycles and

[2]This combination of processor type, core frequency, secondary cache speed and size will be abbreviated as R10K 250/250/4.

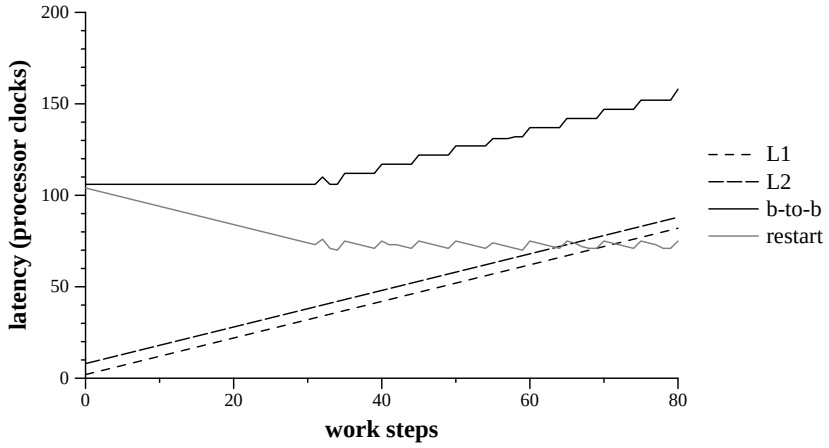


Figure 4.4: Back-to-back and restart latencies on a R10K 250/250/4 system

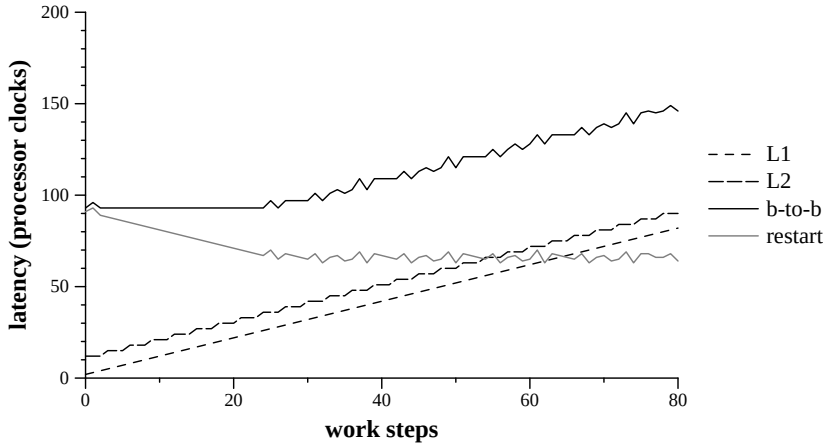


Figure 4.5: Back-to-back and restart latencies on a R10K 195/130/4 system

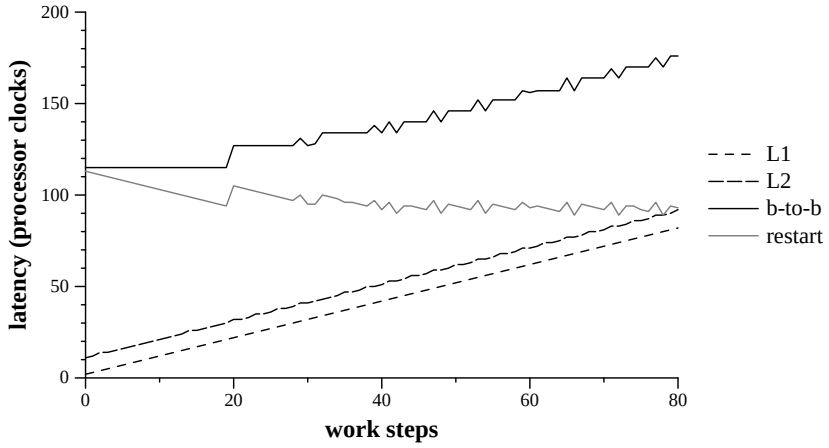


Figure 4.6: Back-to-back and restart latencies on a R12K 300/200/8 system

increases one processor cycle for each add instruction. Just like R10000 running at $3:2$ processor to cache speed ratio, the L2 plot is a step function; the data is returned from L2 in either 9 or 10 cycles. The back-to-back plot reveals the change in cache line transfer algorithm: there are two large steps following the initial flat portion of the plot; the plot then changes into a series of smaller steps superimposing the L2 cache over the SysAD bus timing similar to the one in Figure 4.5.

Restart Latency Algorithm

The algorithm used in [19] computed the restart latency by progressively increasing the amount of work following the load until the back-to-back time started to increase. With different processor behaviors dependent on different processor generations and different cache clock speeds, detecting the knee of the back-to-back latency curve and computing the restart latency at that point does not report correct results. A different restart algorithm is needed to handle different situations.

The restart plots in Figures 4.4–4.6 are computed as the difference between back-to-back and L1 latencies. They all start with a steep descent and end in a horizontal saw-like pattern that reflects the SysAD timing. Instead of looking for the knee of the back-to-back curve, the restart algorithm in snbench computes restart latency as the difference between the back-to-back and L1 times. On each iteration, the algorithm adds one additional work step. The algorithm terminates when the average restart latency computed in a moving window of specified length stays within a boundary of the moving average for a sufficient number of work steps. Default values for the moving window, bounds and stable sequence are 13 iterations, ± 5 ns and 19 iterations, respectively. The defaults were chosen to work sufficiently well for the whole range of systems where snbench was tested; they can be changed through command line options. The search will also be terminated when the algorithm runs out of work functions.

When the average restart latency stabilizes, the restart plot looks like a series of more or less broken saw tooths. As mentioned before, this mode reflects the combined SysAD and cache timing. Which restart value should be reported as the restart latency, the moving average or the (absolute) minimum? The average value reflects the cost of "the average" load in isolation; i.e., if there is no contention for the L2 bus, the average time takes into account the fact that the load miss timing depends on L2 and SysAD timing. However, it is not clear that the load misses will not suffer from L2 bus contention. Since the whole idea behind the restart latency is to give an estimate of the best-case latency, it seems more appropriate to chose minimum restart time as the (reported) restart latency. The snbench will report the minimum latency observed in the stable window as the restart latency; it can also report window median, average and maximum times.

4.3 Results

This section presents snbench results gathered on a variety of Origin systems. Section 4.3.1 gives local latency and bandwidth results. These values give a baseline system performance, where all ccNUMA effects are ignored. If the application placed its data such that there were no remote accesses, it would observe the local latencies and bandwidths. This section also presents improvements due to the increase of processor speed and changes in its microarchitecture, which were introduced in several generations of the Origin 2000 systems. Section 4.3.2 analyzes the impact of accessing data in remote nodes, the most important aspect of the ccNUMA architecture. The

distance between the requestor and the home node is not the only parameter that affects the memory performance. Another important parameter is the directory transaction type. Section 4.3.3 evaluates intervention transactions, where cache line ownership is transferred from one processor to another. Section 4.3.4 evaluates invalidation transactions, where a processor invalidates shared copies before it can assume an exclusive ownership of a cache line.

Table 4.5 presents the systems that were used to collect microbenchmark data. System `bitmap` is an example of the first generation of the Origin2000 system based on 195 MHz R10000 processors; the Hub runs at 97.5 MHz to accomodate the $2:1$ processor/SysAD clock ratio. The follow-on system, `bootleg`, uses a linear shrink of the R10000 processor running at 250 MHz; in addition to faster processors, it also increased the Hub frequency to 100 MHz for a $5:2$ clock ratio. The 250 MHz systems have synchronized processor and L2 cache interface speeds; all other systems run the cache interface at a $3:2$ ratio. The third generation of Origin systems shipped with the R12000 processor featuring enhancements in the microarchitecture and a higher clock frequency (300 MHz). The last member of the Origin2000 family is based on a shrink of the R12000 processor running at 400 MHz. All systems based on the R12000 use a 100 MHz Hub frequency.

system	*CPUs*		*Hub*	*comment*
`bitmap`	16	R10K 195/130/4	97.5 MHz	first generation
`bootleg`	32	R10K 250/250/4	100 MHz	$1:1$ L2 cache speed
`arctic`	64	R12K 300/200/8	100 MHz	R12000 CPU
`bwana`	16	R12K 400/266/8	100 MHz	last generation
`stinger`	128	R12K 300/200/8	100 MHz	128P system

Table 4.5: Origin 2000 systems used in experiments

Contrary to the incremental changes in processor and Hub frequency, the system interconnect has not changed through the life of the Origin systems. The router core runs at 100 MHz and the link signalling runs at four times the router speed. The backplane connecting the node to its nearest router runs at 390 MHz. All the systems up to 64 processors use regular routers. As shown in Figure 3.6, systems with more than 64 processors connect 32P hypercubes with a metarouter; `stinger` is an example of such an Origin system.

The Origin system scales from 2 to 512 processors. Up to 128P, the directory cache coherence protocol uses a 64-bit vector to keep track of the sharers of the cache line. Beyond 128P, the directory employs a coarse-grained vector where each bit represents 8 nodes. Also, the metarouters for 256P and 512P systems are different from the metarouter for the 128P system. Unfortunately, we did not have access to systems larger than 128P.

4.3.1 Local Transactions

This section presents latency and bandwidth results for four generations of Origin 2000 systems. We start with local results obtained on the first generation of the Origin systems, similar to the one used by Hristea et al. [19] (a R10K 195/130/4 system). We continue with Origin generations that featured either a processor speed bump (R10K 250/250/4 and R12K 400/266/8), or

tweaks in the processor microarchitecture (R12K 300/200/8). The local results are comparable to a dual-processor SMP systems—all communication is limited to a single node. However, the cache coherence between the two peer processors on each node still involves the directory protocol transactions. The latency and bandwidth results are presented for each composite directory state and the protocol request issued.

Table 4.6 shows local results measured on a 195 MHz R10000 system. The back-to-back and restart latency results were computed as the minimum value over ten iterations of the experiment, whereas the bandwidth results were computed as the maximum value over ten iterations of the experiment. All experiments were timed with a high-precision (800 ns) cycle counter. The standard deviation for latency results of ten iterations of the experiment was less than 0.5 ns for READ requests on UOWN, SHRD, and CEXM states, and less than 2.5 ns for READ requests on CEXH, DEXD, and DEXT states, and RDEX requests on all states (higher variation for these results is due to the smaller size of the test data, which must fit in the processor's secondary cache). The standard deviation for bandwidth results was less than 0.3 MB/s for READ and RDSH requests on UOWN, SHRD and CEXM states, and less than 3 MB/s for other request/state combinations. Even for results with higher measured standard deviation, the median and average results for those experiments were usually found to be close to the value reported here, which suggests that a temporary system activity during the measurement produced an outlier. Similar standard deviations were observed during experiments on other systems whose results are presented in this section.

bitmap	*back-to-back (ns)*			*restart*	*bandwidth (MB/s)*			
state	*READ*	*RDEX*	*UPGRD*	*READ*	*READ*[1]	*RDEX*	*zero*	*UPGRD*
UOWN	476	516		323	518	480	266	
SHRD	476	895	707	323	517	311	205	423
CEXH	700	742		613	343	317		
CEXM	702	742		616	337	317	211	
DEXD	1022			857	176			
DEXT		1083		880[2]		174		

[1] Identical results to RDSH.

[2] This experiment used RDEX request.

Table 4.6: Local results for a R10K 195/130/4 system

The first four columns in Table 4.6 show restart results for various combinations of directory requests and cache line states. The READ results for UOWN lines estimate local memory latency. The results suggest that in the absence of the SysAD bus and memory contention, the load latency is 323–476 ns. The 153 ns difference is due to the contention for the processor L2 bus: the best case is estimated by measuring the restart latency while the worst case is estimated by measuring the back-to-back latency. When comparing READ latencies for various coherence states, we see that the UOWN and SHRD results are identical, while the latencies for other coherence states are much higher. This is due to different directory transactions involved in each case. Read requests for SHRD lines result in identical transactions as read requests for UOWN lines (unowned transactions). On the other hand, clean-exclusive and dirty-exclusive intervention transactions involve the other processor on the local node, resulting in a much higher latency. The intervention cost is very high

because the SysAD bus is multiplexed between the two processors on the node.

The RDEX latency for SHRD lines is almost twice as high as the READ latency. This difference is due to the cost of the invalidate transaction, which is required when the processor requests the exclusive ownership of the cache line with the RDEX request. In local SHRD experiments, both processors on the node first touch all cache lines in the test array with load instructions. Then one processor waits while the other processor measures the time it takes to chase the linked list. In the RDEX case, a dummy store on each cache line is performed before dereferencing the pointer, which results in an external RDEX transaction. Since the R10000 processor uses strict memory ordering, the store cannot be retired until the cache line is invalidated in the other processor's cache. This involves an invalidation transaction on the SysAD bus and a coherency response from the other processor, which compete with the data response transaction to the requestor. Compared to the RDEX latency, the UPGRD latency is much lower. In this case, the processor already holds a valid copy and it only needs to wait until the other processor invalidates its copy before the transaction is completed.[3]

Surprisingly, the RDEX latencies for UOWN, CEXH and CEXM lines are ≈ 40 ns higher than the corresponding READ latencies, even though the directory transactions are the same. This phenomenon is again due to the contention for the processor L2 bus. The RDEX requests are generated by modifying each element in the linked list before dereferencing the pointer. The size of the linked list is chosen not to exceed processor L2 size to avoid writebacks to main memory. However, the processor needs to write the modified 32-byte L1 cache lines back to L2. The L1 writebacks will contend for the L2 bus together with the tag check for the next item in the linked list. After the L2 bus becomes free, the L2 writeback is performed first, followed by the tag check, further delaying the next RDEX request on the SysAD bus. Faster cache bus speed and the modifications in the secondary cache controller introduced with R12000 have eliminated this anomaly.

The bandwidth results in Table 4.6 show the same tendencies as the latency results. The highest bandwidth (518 MB/s) is achieved with READ requests on UOWN lines. Composite cache states, which result in intervention and invalidate transactions, cause a significant (albeit not as high) decrease in achieved bandwidth. Similarly, comparing READ and RDEX bandwidths reveals a slight decrease due to the contention for the processor L2 cache bus. We also measured the bandwidth results for RDSH requests; as expected, we found them to be identical to the READ results.

Table 4.7 summarizes local latencies and bandwidths measured on an Origin system with 250 MHz R10000 processors. Compared to the 195 MHz results shown in Table 4.6, there is a decrease in back-to-back and restart latencies. The bandwidth results have improved because of the higher processor core frequency and the higher Hub frequency (the SysAD bus now runs at 100 MHz instead of 97.5 MHz). The RDEX latency increase is smaller compared to the 195 MHz systems. Again, this is due to the faster L2 cache bus.

The local results for a 300 MHz R12000 system are summarized in Table 4.8. The microarchitectural changes in the L2 cache controller help reduce back-to-back latencies. The back-to-back READ and RDEX latencies on unowned lines are now equal: the R12000 can overlap L2 line fill from SysAD with the L1 writeback. However, the L2 bus occupancy seems to be a problem with the CEX latencies. The restart latencies are slightly higher than on 250 MHz systems; since both systems run 100 MHz Hubs, the difference can be attributed to the $3:2$ secondary cache ratio. The READ bandwidths have improved because of the higher core clock frequency. However, the RDEX

[3] See Section 4.3.4 for a discussion of UPGRD latencies.

bootleg	*back-to-back (ns)*			*restart*	*bandwidth (MB/s)*			
state	*READ*	*RDEX*	*UPGRD*	*READ*	*READ*	*RDEX*	*zero*	*UPGRD*
UOWN	425	460		285	548	494	273	
SHRD	425	745	603	283	548	353	235	432
CEXH	682	683		565	360	353		
CEXM	683	684		571	360	354	210	
DEXD	983			787	180			
DEXT		966		782		178		

Table 4.7: Local results for a R10K 250/250/4 system

and *zero* bandwidths on shared and clean-exclusive lines are actually slightly lower compared to the 250 MHz system, probably because of the Hub and processor timing interactions.

arctic	*back-to-back (ns)*			*restart*	*bandwidth (MB/s)*			
state	*READ*	*RDEX*	*UPGRD*	*READ*	*READ*	*RDEX*	*zero*	*UPGRD*
UOWN	384	385		298	558	513	266	
SHRD	384	607	341	298	558	326	217	384
CEXH	681	713		577	360	336		
CEXM	683	714		584	354	336	234	
DEXD	978			800	179			
DEXT		897		818		181		

Table 4.8: Local results for a R12K 300/200/8 system

Table 4.9 shows local results for a 400 MHz system. The restart latency has improved and is now about 5 ns lower than the restart latency for 250 MHz R10000 systems. Increasing the clock frequency yields diminishing returns, because the latency bottleneck is now the memory interface. The latencies for CEX lines are the same for READ and RDEX requests due to the faster L2 cache bus. Other results are comparable to the 300 MHz system.

bwana	*back-to-back (ns)*			*restart*	*bandwidth (MB/s)*			
state	*READ*	*RDEX*	*UPGRD*	*READ*	*READ*	*RDEX*	*zero*	*UPGRD*
UOWN	384	383		279	553	535	264	
SHRD	383	637	329	279	553	329	223	398
CEXH	680	682		560	368	340		
CEXM	682	682		564	368	338	229	
DEXD	978			784	179			
DEXT		881		813		182		

Table 4.9: Local results for a R12K 400/266/8 system

Comparing System Generations

Table 4.10 summarizes the important differences between several generations of the Origin node boards. The first column is the back-to-back READ latency, followed by the restart latency; the Δ column is the restart penalty (the difference of back-to-back and restart latencies). The *1P* column shows the sustained uniprocessor read bandwidth. The *SysAD* entries give the sustained dual-processor read bandwidth, followed by the ratio of the SysAD over the single processor bandwidth. The *memory* entries give the cumulative bandwidth of four threads placed on four different nodes all issuing READ requests to a single node; the ratio following the bandwidth numbers is again computed as the total memory bandwidth over a single processor's.

	READ latency (ns)			*READ bandwidth (MB/s)*				
system	*b-to-b*	*restart*	Δ	*1P*	*SysAD*		*memory*	
R10K 195/130/4	476	322	154	518	555	1.07	604	1.17
R10K 250/250/4	425	285	140	548	560	1.02	624	1.14
R12K 300/200/8	384	297	86	559	566	1.01	617	1.09
R12K 400/266/8	384	280	104	553	563	1.02	612	1.10

Table 4.10: A comparison of local results

The back-to-back latency numbers decrease with successive generations. Starting with the 300 MHz systems, the latency is limited by the speed of the Hub instead of the speed of the processor. The restart latency for 400 MHz systems is slightly better compared to the 300 MHz systems, a result of the faster processor clock and the cache interface. Similar effects can be observed with single-processor bandwidth numbers: the values increase with faster processor speeds, almost to the point of using up the full SysAD bandwidth. The *SysAD* and *memory* bandwidths were obtained with the `bwmp-read` experiment. Even though this experiment was intended to evaluate the cumulative bandwidth of several threads issuing memory requests, the results were not as accurate as the other results in this section. The *SysAD* results varied less than 1%; the *memory* results were noisier, varying between 3-5% from the average shown in Table 4.10.

4.3.2 Remote Transactions

The most important difference in accessing memory on a conventional SMP system and on a ccNUMA system is that the memory on a ccNUMA system is divided across multiple nodes. Load and store instructions will transparently access a memory location on a remote node; however, the latency will be significantly different than when the memory location is in the local node.

Table 4.11 shows how latency and bandwidth depend on the distance between the requesting processor and the home node. The data were measured on a 64 processor system with 300 MHz R12000 processors. The values shown in the latency column are the back-to-back latencies for lines in unowned state. The read and zero columns give bandwidths for `sum` and `fill` kernels from Figure 4.2; all cache lines are in unowned state. The transaction is a simple request/reply pair. There are no invalidations or interventions; the only variable is the distance between the requestor and the home node. The values in Table 4.11 are averages for all nodes at the same distance from the home node.

arctic		b-to-b latency		READ		zero	
hops	nodes	ns	incr	MB/s	decr	MB/s	decr
0	1	385		558		266	
1	1	721	87%	446	20%	258	3%
2	8	831	116%	427	24%	245	8%
3	12	946	146%	392	30%	237	11%
4	8	1062	176%	362	35%	230	13%
5	2	1179	207%	335	40%	219	18%

Table 4.11: Remote results on a 64P Origin R12K 300/200/8

There is a relatively large latency increase when the memory location is one router hop away. Latencies two or more hops away increase as a linear function of the number of router hops. Even though the remote latencies increase quite significantly, the remote bandwidth results show less sensitivity to the distance from the home node. This can be attributed to the overlap of multiple outstanding misses which hide some of the remote latency, combined with the small variances in the SysAD bus timings. Remote stores are even less sensitive to the distance to home node. Each store instruction displaces a dirty line in the cache, generating two directory requests: a writeback of the displaced line and an exclusive read of the missed cache line. While the bandwidth of writes is almost half the bandwidth of reads, the decline is not as steep.

Systems with a Metarouter

The nodes in Origin 2000 systems up to 64 processors are connected in a multidimensional hypercube. Larger systems are organized in a fat hypercube where three or more 32-processor hypercubes are linked together with a metarouter. Table 4.12 shows remote latencies and bandwidths collected on a 128-processor system.

stinger		latency		READ		zero	
hops	nodes	ns	incr	MB/s	decr	MB/s	decr
0	1	384		557		266	
1	1	763	99%	437	22%	232	13%
2	6	914	138%	418	25%	235	12%
3	12	1093	184%	365	34%	224	16%
4	20	1264	229%	327	41%	215	19%
5	18	1424	270%	298	47%	203	24%
6	6	1575	310%	274	51%	192	28%

Table 4.12: Remote results on a 128P Origin R12K 300/200/8

The 128-processor system is physically larger and the cables connecting nodes to metarouters are longer. Compared to the 64-processor system, we expect the remote latencies to be 5–10 ns higher due to longer cables which are used in the 128P systems. However, the latencies are about 50 ns higher, even for nodes which are physically in the same 32-processor hypercube as the

home node. The cabling of the standalone hypercubes is the same as in the hypercubes which are linked together with a metarouter. The relatively high remote latency increase is a result of the workaround for a hardware bug in the metarouters. To prevent a race condition in hardware, the IRIX operating system disables the router bypass mechanism in all systems with a metarouter. This workaround adds a two-cycle delay for each micropacket. Both the request and the reply micropackets are delayed; at 100 MHz router clock this results in a 40 ns latency increase. Additional latency is due to longer cables in large Origin systems.

Detailed Remote Results

Latencies and bandwidths differ between nodes even when they are at the same distance from the home node. The nodes are connected to routers via the system backplane. Two routers in the same module are connected via the backplane as well. The routers in different modules are connected via external cables. Figure 3.8 shows the physical configuration of a 32-processor hypercube. The external cables vary in length from 65 to 180 inches. Considering the propagation delay (about 5 ns/m), the cabling adds 8–23 ns to the one-way latency. These differences are noticable in the complete per-node remote results, shown in Table 4.13. Each row in this table gives latency and bandwidth results for one node in a 64-processor system; the nodes are shown in increasing distance from the home node where the test data were placed. The back-to-back and restart latencies are The latency result is the minimum from ten iterations of the experiment while the bandwidth result is the maximum from ten iterations; they are followed by the standard deviation of the results. The latency and bandwidth values for each node are shown in Figure 4.7. These results were collected on a 64-processor R12K/300/200/8 system; detailed results for a 128-processor system with a metarouter are presented in Appendix A.

The test memory for all results was allocated on node 1. The closest neighbor (node 0) is one router hop away; both nodes are connected to the router via the backplane. There are eight nodes two hops away. Nodes 0–3 are in the same module, together with two routers that are connected via the backplane; the path from node 1 to nodes 2 and 3 goes entirely through the backplane. The extra router hop adds almost exactly 100 ns. The latencies for nodes 2 and 3 are about 10 ns less than the latencies for nodes 4, 5, 8 and 9 which are in the other module. Routers in different modules are connected via a cable which adds the extra latency. Nodes 16 and 17 are in the other 32P metacube; the cables connecting routers in different metacubes are longer than cables connecting routers in different modules, which explains additional 10 ns latency increase. Nodes three hops away reveal similar results: the path from node 1 to nodes 6, 7, 10 and 11 goes once through the backplane and once through the cable, whereas the path to nodes 12–13 goes through two cables, which adds another 12 ns. Various combinations of backplane and cable lengths explain similar latency variations in other nodes.

Modeling Remote Latency

Figure 4.8 shows the critical path for an unowned transaction. Local processor (L) requests a cache line via the SysAD bus transaction. The Hub processor interface (PI) allocates a CRB entry and sends out a READ request. If the node id of the requested cache line matches the local node id, the crossbar (#) routes the request to the local memory/directory interface. Requests for remote lines enter the interconnect network through the Hub network interface (NI), pass through one or more

arctic		*latency (ns)*		*bandwidth (MB/s)*		
node	*hops*	*b-to-b*	*restart*	*READ*	*RDEX*	*zero*
1	0	385	298	558	512	266
0	1	721	629	446	376	258
2	2	820	733	432	350	240
3	2	818	732	432	349	253
4	2	832	747	425	344	252
5	2	831	745	425	344	252
8	2	831	747	424	344	252
9	2	831	746	424	342	252
16	2	843	754	425	345	222
17	2	841	754	425	342	239
6	3	937	851	393	317	234
7	3	936	852	392	320	234
10	3	938	850	392	317	234
11	3	936	849	392	319	234
12	3	949	864	390	316	246
13	3	949	863	390	317	246
18	3	945	858	398	316	234
19	3	944	857	395	318	234
20	3	957	870	389	315	240
21	3	956	870	390	315	233
24	3	955	869	390	314	233
25	3	955	868	389	313	243
14	4	1054	968	364	294	239
15	4	1054	967	364	294	228
22	4	1063	973	361	292	238
23	4	1063	973	361	291	227
26	4	1060	972	362	293	228
27	4	1059	971	362	293	227
28	4	1072	984	359	289	224
29	4	1073	983	359	290	227
30	5	1179	1091	336	271	220
31	5	1179	1091	335	271	218

Table 4.13: Remote results for a 64P R12K 300/200/8 system

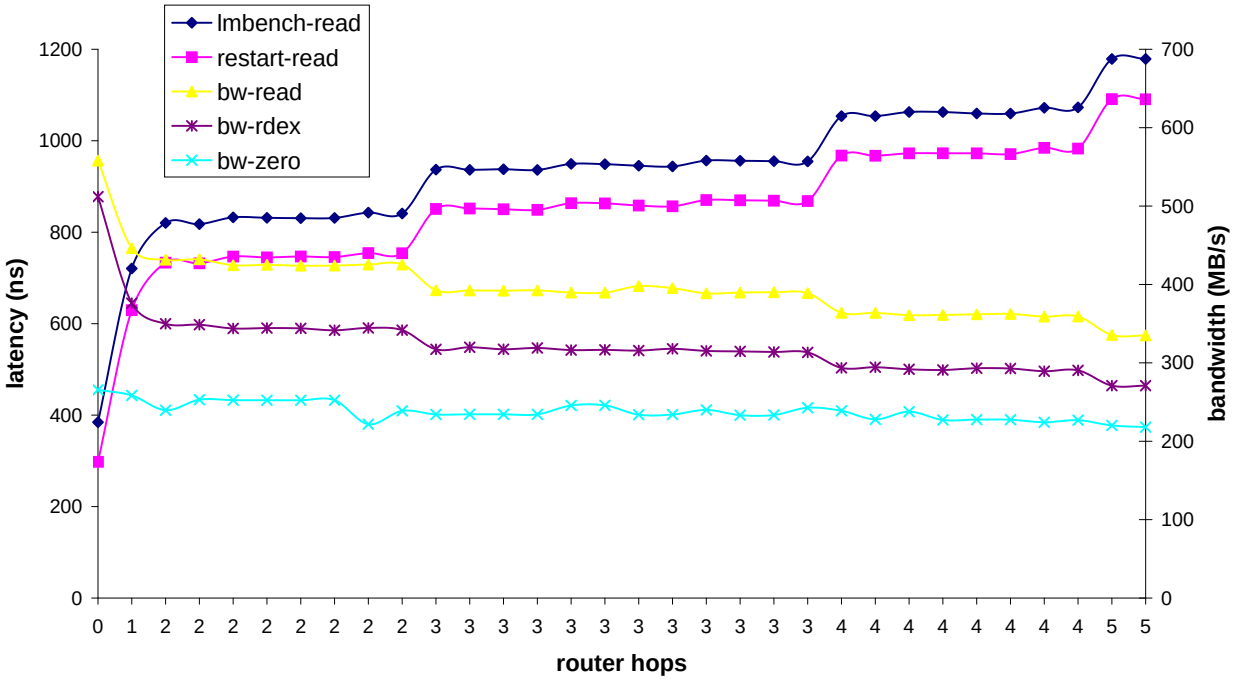


Figure 4.7: Remote latency and bandwidth chart

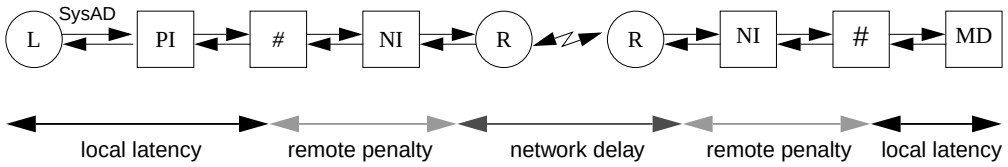


Figure 4.8: Message flow in unowned transactions

routers (R), and then enter the network interface of the (remote) home node. The crossbar on the home node forwards the request to the memory/directory interface (MD), which reads the cache line from memory and performs the directory lookup. The reply goes back through the same path back to the PI section of the local Hub, which deallocates the CRB entry after forwarding the data to the processor via the SysAD bus response.

The latency can be broken down into three parts: local latency, remote penalty, and network delay. The local latency is the time it takes to get the cache line from the local memory; it covers message flow from the processor through the processor interface, the crossbar, the memory/directory interface, and back. The *remote penalty* is the fixed cost of sending a message into the interconnect and receiving it back into the node. This penalty is added to all remote requests regardless of the distance to the home node. The *network delay* is the time spent in the network; it depends on the number of router hops.

Table 4.14 shows the average router delay measured on several Origin systems. Except for `stinger`, which is a system with a metarouter, the average delays for all systems are in the 105–120 ns range. Router pin-to-pin latency with bypass enabled is 40 ns (four 100 MHz router core

clock cycles). Since we are measuring the round trip delay, we get 80 ns spent in the router and 35–50 ns due to the signal propagation delay. Combining the two together, we get an average router delay of 111-114 ns. With a metarouter, the round trip pin-to-pin latency is 120 ns and the signal propagation delay is 23–59 ns, for a total of 165 ns average router delay.

	stinger		arctic		lego-oil		bootleg	
hop	*nodes*	*delay*	*nodes*	*delay*	*nodes*	*delay*	*nodes*	*delay*
1	1	149	1	106	1	107	1	111
2	6	151	8	110	6	107	6	118
3	12	179	12	115	6	115	6	113
4	20	171	8	116	2	114	2	110
5	18	160	2	117				
6	6	151						
avg		165		114		111		114

Table 4.14: Remote penalty and average router delays

The extra latency from the home node to its nearest router combines two quantities, the router delay and the remote penalty. Two neighboring nodes are connected to the router via the backplane. From the detailed analysis of remote latencies it follows that the backplane delay is ≈ 100 ns. The remaining time is spent in node network interface and on the system backplane. We have chosen an uniform remote penalty of 230 ns. The router delay for the first hop is the unowned remote latency less the remote penalty.

The remote penalty of 230 ns is much higher than the value suggested by the previously published results (130 ns) and the value used in standard Origin literature (165 ns). It is possible that this discrepancy is due to the systems used in previous studies. Previous researchers measured results on a system that had the 390 MHz backplane frequency synchronized with the 97.5 MHz Hub. Newer systems have the Hub running at 100 MHz; additional latency may be due to this crossing of clock domains.

4.3.3 Interventions

Intervention transactions are a group of transactions that are used to change the ownership or downgrade an exclusive copy of the cache line. In all cases, intervention transactions involve three participants. The requestor issues a read request and the home node finds the cache line owned by a remote processor. In response, the home node issues sends intervention read-shared or intervention read-exclusive request to the remote processor and a speculative reply to the requestor. The remote processor downgrades the state of the cache line to either shared or invalid, and forwards the reply to the requesting processor and a revision message to the home node. Depending on the state of the cache line in remote processor's cache, the remote sends a copy of the cache line to the requestor if the line was modified (a dirty-exclusive transaction), or a simple acknowledgement (a clean-exclusive transaction) if not.

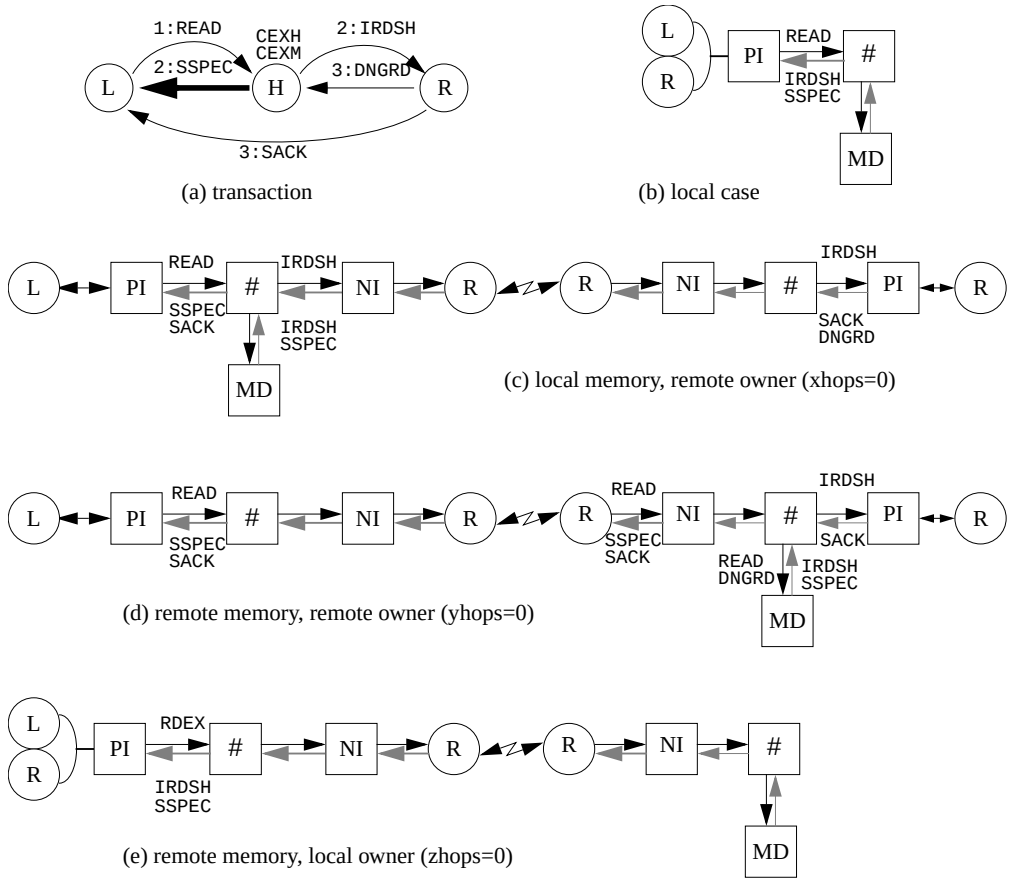


Figure 4.9: Message flow in clean-exclusive transactions

Clean-Exclusive Miss Transactions

To evaluate the latency of clean-exclusive miss (CEXM) transactions, the remote thread touches all data items in the test array (which is larger than the processor's L2 cache) and then invalidates the cache. This places the cache lines in exclusive directory state while the lines are dropped from the remote's cache. The local thread then chases a list of pointers in the test array: each read request results in an intervention transaction. Figure 4.9 (a) shows the resulting message flow.

In general, the transaction participants can be placed anywhere in the system. The transaction latency depends on the length of the critical path: the number of hops between the requestor and the home node ($xhops$), the home node and the remote processor ($yhops$), and the remote processor and the requestor ($zhops$). The simplest case is when all three participants are located on the same node; Figure 4.9 (b) shows the resulting message path from the processor interface (PI) through the crossbar into the memory/directory unit (MD). The speculative reply and the intervention request both travel back to the local PI, which issues an intervention request on the SysAD bus to the remote processor (R), and a data response to the requestor (L). When the remote processor responds to the intervention request, the local PI sends the acknowledge through the crossbar back to itself, which finally results in the data acknowledge transaction on the SysAD.

When the home node or the remote processor are on a remote node, the messages have to go

through the interconnect network. As we saw in the previous section, remote transactions incur the remote penalty and the network delay. In order to compare the intervention latencies with unowned latencies we have evaluated three placements which limit the number of variables by placing two participants on the same node. In the first experiment, the requestor is placed on the home node ($xhops = 0$); in the second experiment, the remote processor is placed on the home node ($yhops = 0$); in the last experiment, the local and the remote processors are placed on the same node while the home node is remote ($zhops = 0$). All three experiments varied the distance between the two nodes between zero hops (local case) and the maximum distance in the 64P system. The message paths for each case are shown in Figure 4.9.

Table 4.15 shows latency data for remote clean-exclusive miss transactions measured on a R12K 300/200/8 system. The intervention latencies are compared to remote latencies. The UOWN column shows remote latencies for unowned transactions. Each of the three CEXM transaction placements gives the latency and the difference compared to the unowned transaction the same number of router hops away. The zero-hop row shows local results where all the participants are on the same node. The ΔU columns show the average difference between the CEXM and UOWN latencies.

	UOWN	$xhops = 0$		$yhops = 0$		$zhops = 0$	
hops	*ns*	*ns*	ΔU	*ns*	ΔU	*ns*	ΔU
0	385	683	298	683	298	683	298
1	721	987	266	942	221	1030	309
2	831	1105	274	1053	222	1174	343
3	946	1213	267	1165	218	1303	357
4	1062	1338	276	1283	221	1438	376
5	1179	1446	267	1397	219	1575	396
avg			270		220		n/a

Table 4.15: Remote latencies for clean-exclusive miss transactions

The smallest latency increase is shown when the remote processor is placed on the home node ($yhops = 0$). The average 220 ns latency increase is the time needed for the PI to issue the intervention on the remote node's SysAD bus and for the remote processor to perform the L2 cache lookup. The speculative reply message reaches the local node first; the local PI immediately acquires the SysAD bus and starts transferring the data into local processor's cache (it takes 16 Hub cycles to transfer the entire cache line over the SysAD). By the time the SACK message reaches the local PI 22 Hub cycles later, the data is already in the cache and the PI completes the transaction by issuing the completion response on the SysAD bus. When the home node is local, the speculative reply message reaches the local PI well before the SACK message which completes the transaction. The extra 50 ns measured in the $xhops = 0$ case could be because the PI has released the SysAD bus after the speculative data were transferred in the requestor's cache and the PI needs to reacquire the bus mastership.

When the requesting processor and the remote owner are placed on the same node and the home node is remote, the difference in CEXM and UOWN latencies shows no clear trend. The additional latency is much higher compared to the other two cases. The bottleneck here is obviously the contention for the SysAD bus: the local PI has to issue an intervention to the remote owner and the

speculative data response to the requestor. It is also not clear what is the reason for the increasing difference between UOWN and CEXM latencies. A possible explanation is that the SSPEC message is delayed in the interconnect; when it reaches the local PI section it occupies the SysAD bus, which further delays the processing of the intervention response from the remote processor.

Clean-Exclusive Hit Transactions

This type of transaction is very similar to the clean-exclusive (miss) transactions discussed above. All the protocol messages are the same; the only difference is that the remote processor keeps the data in its L2 cache in a clean state. The intervention request issued on the SysAD bus hits in the L2 cache, but the response is clean, which means that the requestor can use the data received in the speculative reply.

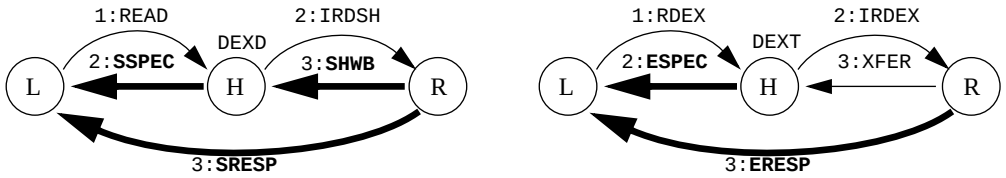
Table 4.16 shows the latency data for clean-exclusive hit transactions. The values are comparable to the clean-exclusive miss case. The noticable difference is the 12 ns latency decrease for the cases where the current owner is on a remote node ($xhops = 0$ and $yhops = 0$ cases). This difference is due to the organization of the L2 cache. The two-way set associative cache lookup is performed first by checking the tags in the most-recently used set. If the lookup fails, another L2 bus transaction checks the tags in the other set. Since the test data used in the experiment amount to only half the size of processor's L2 cache and the majority of the cache lines are from the test data, it is likely that the match occurs on the lookup of the first cache set. In the clean-exclusive miss case, the lookup misses both in the first and in the second set, incurs an additional L2 bus transaction. With the 200 MHz cache bus frequency, the penalty for the additional cache set lookup is 6 cycles.

	UOWN	$xhops = 0$		$yhops = 0$		$zhops = 0$	
hops	*ns*	*ns*	ΔU	*ns*	ΔU	*ns*	ΔU
0	385	681	297	681	297	681	297
1	721	979	259	926	206	1027	307
2	831	1092	261	1046	215	1172	341
3	946	1203	256	1152	206	1301	355
4	1062	1325	263	1269	207	1437	375
5	1179	1434	255	1385	206	1572	393
avg			259		208		n/a

Table 4.16: Remote latencies for clean-exclusive hit transactions

Dirty-Exclusive Transactions

In dirty-exclusive transactions, the remote owner holds a modified copy of the cache line. The requestor uses the data supplied by the remote processor and discards the data sent with the speculative reply from the home node. There are two types of dirty-exclusive transactions, depending on the type of request issued by the requestor. When the requestor issues a READ request the cache line is downgraded from exclusive to shared state; the remote processor supplies the data



(a) dirty-exclusive downgrade (b) dirty-exclusive transfer

Figure 4.10: Dirty-exclusive transactions

to the requestor and also updates the memory copy by sending the shared writeback (SHWB) revision message (this is necessary because the home node supplies the data when the cache line is in shared state). This dirty-exclusive downgrade (DEXD) transaction is shown in Figure 4.10 (a). When the local processor requires an exclusive copy of the dirty cache line, the remote processor still supplies the data with the ERESP message. However, the memory copy does not have to be updated, because the cache line still remains in the exclusive state. The XFER revision message simply notifies the home node that the transaction has completed. The dirty-exclusive ownership transfer (DEXT) transaction is shown in Figure 4.10 (b).

Table 4.17 shows dirty-exclusive transfer latencies.[4] All three cases show a significant increase compared to the clean-exclusive results. Additionally, the data sent with the exclusive reply adds more latency as the number of hops is increased. The only exception here seems to be the case where the local processor is placed on the home node: the speculative reply sent by the home node is received well before the dirty data from the remote node—the PI can simply transfer new data over the SysAD to complete the transaction.

	UOWN	$xhops = 0$		$yhops = 0$		$zhops = 0$	
hops	*ns*	*ns*	ΔU	*ns*	ΔU	*ns*	ΔU
0	384	897	513	897	513	897	513
1	722	1149	428	1053	331	1236	514
2	830	1245	415	1177	348	1380	551
3	945	1359	414	1295	350	1513	568
4	1061	1480	419	1424	363	1650	590
5	1177	1598	421	1551	374	1786	609
avg			420		n/a		n/a

Table 4.17: Remote latencies for dirty-exclusive transfer (DEXT) transactions

The most expensive intervention transactions are dirty-exclusive downgrades. The latencies shown in Table 4.18 are even higher than the dirty-exclusive transfers—this is the impact of the sharing writeback message. As in all intervention transactions, the most visible impact is in the

[4]Note that this table uses the `lmbench-rdex` latencies on unowned lines as the baseline for comparison to DEXT transactions, as opposed to the `lmbench-read` values that are used as the baseline for the DEXD transactions. On R12000 processors, the RDEX latencies are almost identical to the READ latencies. On R10000 processors, the RDEX latencies are ≈ 40 ns higher.

	UOWN	$xhops = 0$		$yhops = 0$		$zhops = 0$	
hops	*ns*	*ns*	ΔU	*ns*	ΔU	*ns*	ΔU
0	385	978	593	978	593	978	593
1	721	1164	444	1044	324	1329	608
2	831	1366	535	1182	351	1469	638
3	946	1452	505	1308	361	1600	653
4	1062	1586	524	1445	383	1735	673
5	1179	1673	494	1583	404	1869	690
avg			n/a		n/a		n/a

Table 4.18: Remote latencies for dirty-exclusive downgrade (DEXD) transactions

local case, when the multiplexed SysAD bus shows its limitations. There is a large variance in timing, even when the local processor is placed on the home node: the data in the sharing writeback impacts the timing of the shared reply.

4.3.4 Invalidations

The third important transaction is generated when a processor wants to get exclusive access to a cache line that is shared among several processors. This transaction involves the local processor, the home node, and one or more nodes which have a shared copy of the cache line. If the processor does not have a shared copy, it will issue a read-exclusive request. The home node will determine that the cache line is shared among a list of nodes, and it will send each sharer an invalidate request. If the local processor has a copy of the cache line, it will request exclusive access by issuing the upgrade request; the home node will respond by sending out invalidate messsages to the sharers and a copy of the cache line to the local processor. In both cases the transaction cannot complete until all the acknowledgements have been received from the sharer nodes, because the Origin protocol implements sequential memory consistency model.

The performance of invalidate transactions depends not just on the distances between the requestor, the home, and the sharer nodes, but also on the number of nodes which have a shared copy and thus need to be invalidated. We have analyzed the impact of distance between the requestor and the home node in Section 4.3.2. To evaluate the impact of the number of sharers on the performance of the invalidation transactions, we have executed a series of experiments with the local processor and the test array on the same node. The cache lines in the test array were placed in the shared state by having processors on remote (sharer) nodes read, and subsequently drop, the cache lines in the entire test array. The local processor then executed latency and bandwidth kernels where each store instruction triggered an invalidate transaction. The first set of experiments was performed with one sharer placed on the same node as the local processor and the home node. Each subsequent experiment added one node to the list of sharers, with the nearest nodes added first.

Figure 4.11 shows the message flow in the invalidate experiments. The local processor issues a read or upgrade bus transaction, which is translated into a RDEX or UPGRD request by the processor interface section of the local Hub. The request travels through the crossbar to the memory/directory interface on the local node. The MD performs a directory lookup, finds out that the line is in

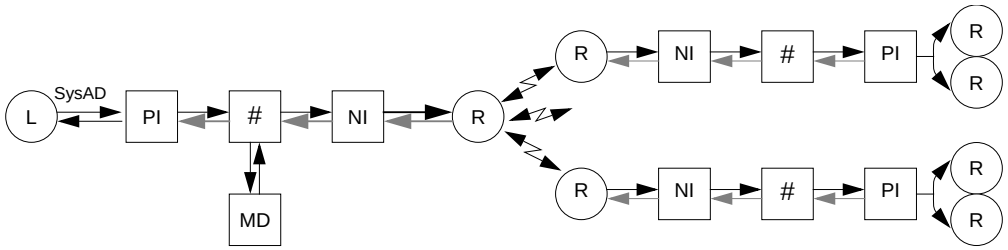


Figure 4.11: Message flow in invalidate experiments

the shared state, and sends the invalidate message with the bit vector representing sharers to the network interface; additionally, a response with the number of sharers is sent to the local PI. The network interface expands the bit vector into individual invalidate messages and injects them into the interconnect. Each sharer node responds to the invalidate message by issuing an invalidate SysAD bus transaction to both processors and sends an acknowledgement to the local PI. When all outstanding acknowledges have been collected, the local PI completes the transaction by issuing a response to the local processor on the SysAD bus.

The data in Table 4.19 show the impact of the number of sharers on the latency and bandwidth results. The local processor is placed on the home node. The experiment starts with one sharer (the other processor on the home node). In each successive row in Table 4.19, a new node is added to the list of sharers. The nodes are added in increasing distance (in network hops) from the home node. To illustrate, the sharer list for one sharer is simply the home node (node 1). Two-sharer list consists of the home node and its nearest neighbor (node 0). Three-sharer list consists of nodes 1, 0 and 2, and so on. All results were collected on a 64-processor 300 MHz R12000 system, which consists of 32 nodes. The 64-node results are shown in Appendix A.

Invalidate Latencies

Figure 4.12 shows the impact of the number of sharers on the invalidate latencies. The number of sharers is given on the x-axis. The y-axis shows the RDEX and UPGRD latencies from Table 4.19. Only in the single-sharer case are the messages confined to a single node; in the other cases, all but one interventio request and their corresponding acknowledgments traverse one or more router hop. Two parameters impact the invalidate latencies: the number of sharers and the longest distance from the home node to the nodes in the sharer list. To help isolate the impact of the number of sharers, Figure 4.12 includes a plot of remote unowned latencies from the home node to the nodes in the sharer list. Since the local processor is placed on the home node in invalidate experiments, the difference between the invalidate and remote latencies estimates the impact of the number of sharers.

The single sharer case has a higher latency than the unowned case: even though the transaction is confined to a single node, the additional step and the SysAD transaction to the other processor have a high impact. With two sharers, the read-exclusive latency is almost the same as the unowned latency to a home node one hop away. The time to send the invalidate to the nearest neighbor is longer than the time to invalidate the local processor, but not much more than fetching a cache line from the nearest neighbor. The same argument applies in the three-sharer case. The latency to the

arctic			*latency (ns)*		*bandwidth (MB/s)*		
#sh	*n*	*h*	*RDEX*	*UPGRD*	*RDEX*	*UPGRD*	*zero*
1	1	0	608	341	328	385	215
2	0	1	734	433	289	315	206
3	2	2	825	486	264	279	196
4	3	2	858	505	249	268	187
5	4	2	881	520	242	258	179
6	5	2	907	533	232	252	172
7	8	2	927	542	223	246	165
8	9	2	942	558	214	238	158
9	16	2	956	583	205	230	153
10	17	2	977	612	196	221	147
11	6	3	1031	657	186	206	141
12	7	3	1067	687	177	198	137
13	10	3	1092	715	171	189	132
14	11	3	1111	742	164	181	128
15	12	3	1131	771	158	173	124
16	13	3	1146	797	152	165	120
17	18	3	1167	822	146	159	116
18	19	3	1183	850	141	153	113
19	20	3	1198	876	136	148	110
20	21	3	1214	904	132	142	107
21	24	3	1235	940	127	137	104
22	25	3	1254	975	123	132	102
23	14	4	1318	1021	119	126	99
24	15	4	1348	1054	115	121	97
25	22	4	1384	1093	112	117	94
26	23	4	1400	1125	109	114	92
27	26	4	1416	1157	106	110	90
28	27	4	1434	1197	103	107	88
29	28	4	1453	1234	101	104	86
30	29	4	1469	1264	98	101	84
31	30	5	1519	1316	95	98	82
32	31	5	1539	1353	93	95	81

Table 4.19: Invalidate results for a 64P R12K 300/200/8 system

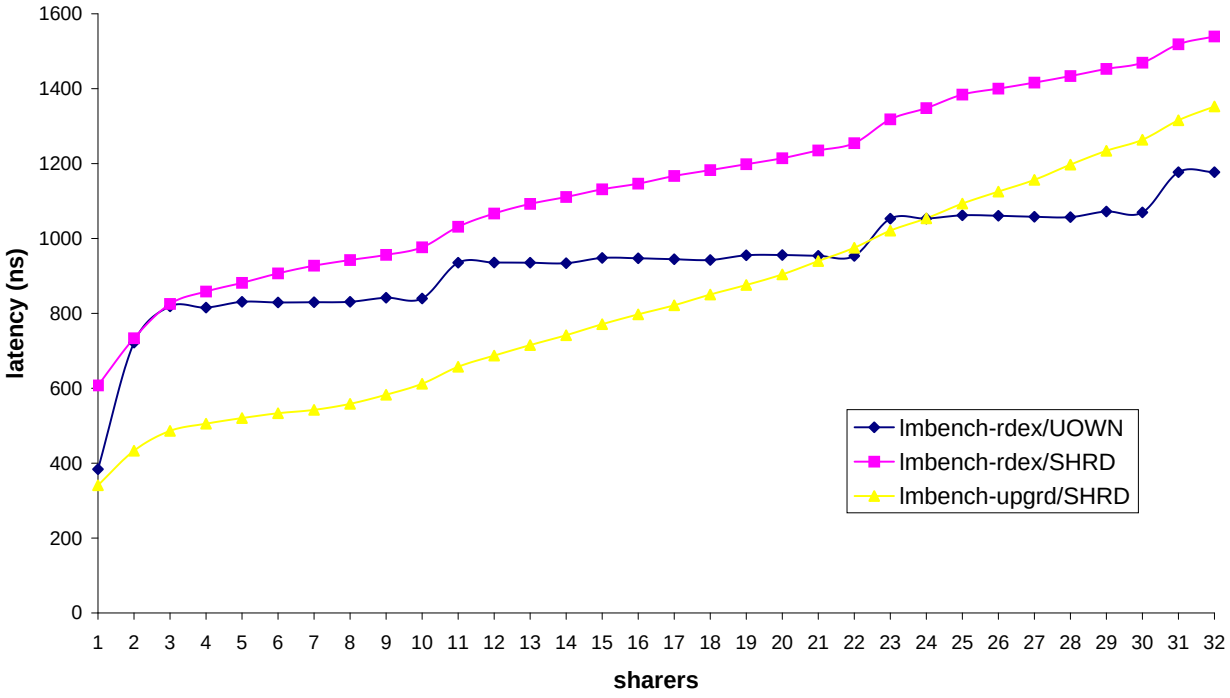


Figure 4.12: Invalidate latency chart

third node two hops away determines the latency of this invalidate transaction. With four or more sharers, the latency plot starts to rise above the unowned back-to-back latency. This could be due to the order in which the invalidate messages are sent out: it looks as if the invalidates for the nodes furthest away were sent last. This experiment placed the home on node 1; nodes with a numerically high node number are physically more distant. The latency behavior could be explained if the Hub generated invalidates in numerically ascending order. Therefore, the invalidates are sent to the nearest nodes first and to the most distant last; the time to generate invalidates is added to the remote latency.

The upgrade plot is somewhat surprising: the latency is much lower than the unowned back-to-back latency, which is clearly not possible. This surprise demonstrates the limitation of the lmbench-upgrd experiment: to generate the upgrade transaction, the line must be present in the requestor's cache (otherwise the store would be translated into a RDEX request). The modify kernel in Figure 4.3 issues a store on the cache line followed by a pointer indirection step. The store triggers the upgrade request while the load completes immediately (because the data is already there). In the worst case, the load is satisfied in the secondary cache. The execution does not depend on the store that triggers the upgrade transaction, and the upgrade transactions are overlapped. The upgrade latency results do not measure the latency of the invalidate transaction in isolation; the stores overlap until an internal processor resource (most likely the load/store issue queue) is saturated.

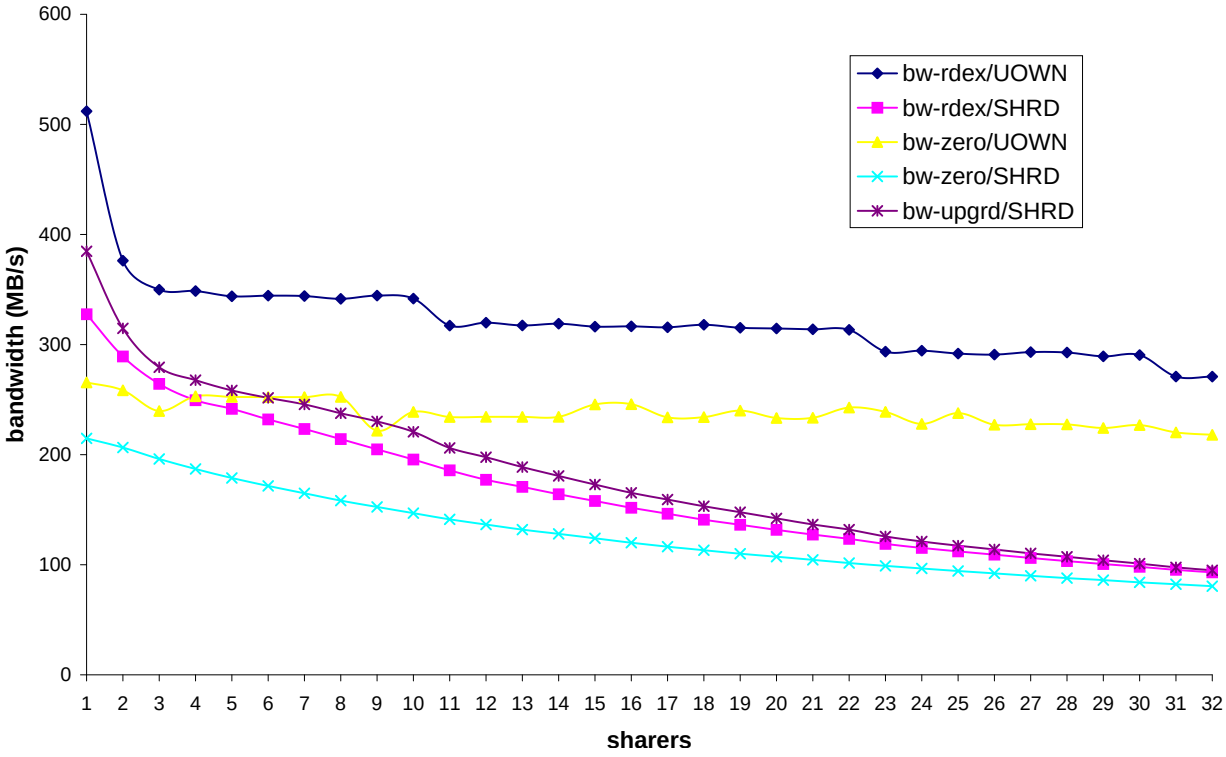


Figure 4.13: Invalidate bandwidth chart

Invalidate Bandwidths

Figure 4.13 shows how the number of sharers impacts the invalidate bandwidth results. Just like the latency invalidate chart, we have included the remote RDEX and *zero* bandwidths for reference. There are no surprising results here. The UPGRD bandwidth is slightly higher than the RDEX bandwidth because upgrades do not carry any data. The *zero* results are less sensitive to the number of sharers although the *zero* bandwidth is slowly descending.

Single Sharer Timings

The last set of invalidate experiments measures the time required to invalidate a cache line in a remote processor cache. Figure 4.14 shows the transactions and the resulting message flows. The cache line is first placed in the shared state by the local processor and a processor on the remote node. The local processor then performs a linked list chase where it issues a dummy store before the pointer dereference. Just like the intervention experiments, there are three nodes involved in single sharer experiments. The results presented in this section are presented in a way similar to the intervention results: the first set of measurements places the requestor on the home node and varies the distance between the home and the sharer nodes ($xhops = 0$); the second set places the home node and the sharer CPU together and varies the distance between the requestor and the home ($yhops = 0$); the last set has the requestor and the sharer CPU on the same node and varies the distance to the home node ($zhops = 0$).

Table 4.20 compares the latency of the RDEX single-sharer invalidations to the remote unowned

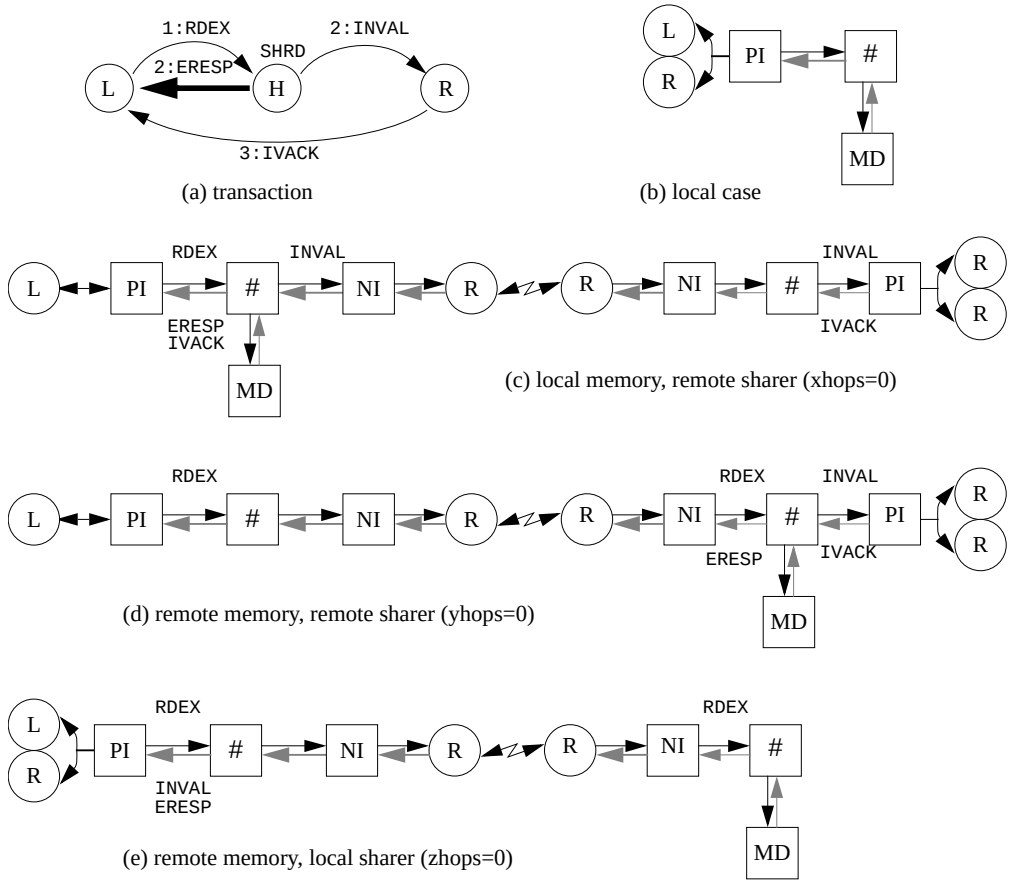


Figure 4.14: Single sharer message flow

transactions. The UOWN column gives average latencies for the RDEX request while increasing the distance to the home node. The three sets of measurement are shown in separate columns. The *ns* column is the absolute time in nanoseconds; the ΔU is the difference from the UOWN time. The results for the local case (where all three participants are on the same node) are very similar to the case where the requestor and the sharer are local and the home is remote ($zhops = 0$): it takes an additional 220 ns to invalidate the sharer cache. This suggests that the INVAL request arrives before the SRESP reply: the local PI acquires the SysAD bus and places the invalidate request first, followed by the data response targeting the requestor. The cache line transfer takes 16 SysAD cycles at 100 MHz; an additional six cycles are required to collect the invalidate response from the sharer CPU, send the IVACK reply from the PI output queue through the crossbar into PI input queue, and send acknowledge to the requesting CPU, which completes the transaction. When the sharer is placed on the remote home node, the ERESP message again arrives before the IVACK reply; again, the difference between both messages depends on the time it takes the sharer CPU to respond to the SysAD invalidation request. Finally, when the requestor is placed on the home node while the sharer is remote, the difference in unowned and invalidation latencies decreases as the distance to the sharer node increases. The data returned in the SRESP reply do not traverse any network hops. They are transferred on the SysAD bus before the IVACK (which completes the transaction) is received from the remote node. Since the SysAD is not occupied with the cache

line data transfer, the next RDEX request does not have to wait and can be issued on the SysAD immediately.

	UOWN	$xhops = 0$		$yhops = 0$		$zhops = 0$	
hops	*ns*	*ns*	ΔU	*ns*	ΔU	*ns*	ΔU
0	384	609	226	609	226	609	226
1	722	729	7	993	271	940	218
2	830	793	-37	1080	251	1059	229
3	945	848	-97	1191	246	1172	227
4	1061	906	-154	1307	247	1283	222
5	1177	960	-217	1422	245	1395	218
avg			n/a		247		220

Table 4.20: Remote latencies for single-sharer invalidations

Chapter 5

The ccNUMA Memory Profiler

The previous chapter has shown how the cost of a memory access can differ greatly in the ccNUMA environment. Simple performance metrics such as the number of cache misses are not sufficient, because they do not give enough information to estimate the cost of a memory access. This chapter presents the design and implementation of the ccNUMA memory profiling tool. This tool is to be used with other performance analysis tools to help determine application memory behavior and system resource utilization. Section 5.1 introduces a set of performance metrics applicable to ccNUMA systems. Section 5.2 gives an overview of the hardware event counters implemented in various Origin 2000 ASICs. Section 5.3 describes the implementation details of four components of the memory profiler: the loadable kernel module, which exports hardware event counters to user programs; the system activity monitor, which offers an interactive view of the system resource utilization; the application launcher, which runs an application and collects event traces while the application is running; and the post-mortem analysis tool, which collates event traces from the application run to compute application resource usage.

5.1 ccNUMA Performance Metrics

Distributed memory systems such as the Origin 2000 replace a single memory controller with a set of memory controllers embedded within each node board. I/O devices can be attached to any node in the system. A fast interconnect network replaces the shared bus. Scalability is achieved by replacing a single resource (memory, shared bus) with a set of distributed resources. The key to achieving application scalability is efficient load balancing: memory bandwidth-intensive applications should place the data across several nodes to benefit from the aggregate memory bandwidth of several nodes; if the data are placed on a single node, the limited memory bandwidth on that node can become the performance bottleneck.

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` [37] 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.

5.1.1 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.

In Chapter 4 we saw how different coherence transactions incur significantly different costs. 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.

5.1.2 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; as we saw in Chapter 4, 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 utiliza-

tion is sufficiently high, it could be worth placing one thread per node instead of two threads per node.

5.1.3 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.

5.2 Origin Hardware Event Counters

The designers of the MIPS R10000 processor understood the importance of hardware support for event counters that can be used to determine performance bottlenecks. The processor event counters are widely used in SGI systems. For example, the SpeedShop performance analysis tool uses processor event counters to help users optimize application performance by reducing the number of cache and TLB misses and branch mispredicts. Unfortunately, the processor event counters are not sufficient to evaluate the performance of the memory system. While it is possible to collect the number of cache misses, the count alone does not give any information about the transactions necessary to bring data items into the processor's cache. This is not a problem in an SMP system: each memory access costs the same. When the cost of a memory access is not uniform, additional information is needed to estimate the cost of accessing memory.

In addition to the hardware event counters in the R10000 processor, the designers of the Origin 2000 have incorporated hardware event counters in the Hub and the Router ASICs. The events defined by these counters can be used to evaluate application memory behavior and to determine potential system bottlenecks. Unlike the processor counters, there is almost no OS support for the Hub and Router event counters and there are no tools which use them. We have implemented a device driver that exports the counters to the user programs. We have performed a sanity check of the counters, verifying that they count events as described in the documentation. This section describes various event counters in the Origin 2000.

5.2.1 MIPS R10000 Performance Counters

The R10000 performance counters can be used to estimate the number of requests generated by each processor. The full description of the R10000 event counters is given in [48, pp. 264–272]; the IRIX API is described in [44] and the use of R10000 performance counters in application performance analysis is described in [57]. The majority of events defined by the R10000 performance counters are related to the working of various processor units. Table 5.1 describes the events that are generated by memory accesses and cache coherence traffic.

The MIPS R10000 has two physical counters that can be programmed to count 32 events; the Irix API that provides access to the performance counters is capable of multiplexing multiple events on a single counter. The operating system reads both hardware event counters during the clock tick interrupt processing (typically every 10 ms). The internally maintained counters for all events are

event	*description*
7	Quadwords written back from secondary cache[1]
10	Secondary instruction cache misses
12	External interventions
13	External invalidations
26	Secondary data cache misses
28	External intervention hits in secondary cache[2]
29	External invalidation hits in secondary cache[2]
31	Store/prefetch exclusive to shared block in secondary cache[2]

[1] Quadword is a 16 byte quantity.

[2] The semantics of this event changes in R12000.

Table 5.1: R10000 performance counter events

64-bit wide. Even though the kernel API attempts to hide the hardware restrictions by allowing event multiplexing, the processor requires that events 0–15 be counted by physical counter 0 and events 16–31 be counted on physical counter 1. The kernel will resort to multiplexing when two events that have to be counted on the same physical counter are selected simultaneously. These restrictions limit the use of processor performance counters: only two events should be counted simultaneously to obtain an accurate count when the measuring interval is short. Fortunately, it is possible to capture almost all processor generated SysAD traffic by counting events 7 and 26 (cache writebacks and secondary data cache misses); in this case, the only unaccounted traffic is due to instruction cache misses and the stores which result in an upgrade request.

5.2.2 Hub Event Counters

The Hub ASIC is divided into five sections: the processor interface, the memory/directory unit, the I/O interface, the network interface, and the crossbar at the center that connects other units. The memory/directory unit and the I/O interface both include registers, which function as event counters. These registers are accessible by issuing uncached loads and stores to the I/O space; they are isolated in a separate 16 KB page of physical address space, which makes it safe to map the page in user space.

Memory/Directory Counters

The register set in the Hub MD unit includes six 20-bit registers which can be programmed to count six types of events. Table 5.2 gives a summary of the Hub MD counting modes.

There is an undocumented system call that provides access to the Hub MD counters. It implements 64-bit virtual event counters and is capable of multiplexing counting modes in a manner similar to multiplexing events on the processor performance counters. The virtualized event counters are updated with the hardware values on every clock tick (10 ms); this is also the multiplexing interval if the user selects more than one counting mode. The 10 ms sampling granularity is barely sufficient to prevent counter overflow. When counting memory activity cycles, one of the coun-

mode	*description*
0	memory activity breakdown
1	outgoing message classification
2	outgoing intervention/invalidation classification
3	incoming message classification
4	directory state for read classification
5	count of requests by local processors

Table 5.2: Hub Memory/Directory event counting modes

ters is incremented every Hub clock tick; with a 100 MHz clock, the 20-bit counter will overflow in 10.5 ms. Additionally, the 10 ms sampling interval is longer than some important application phases (e.g., the matrix transpose in the FFT kernel).

I/O Interface Counters

The Hub IO unit has two 20-bit counters that can count various events. The two counters were meant to be independently controlled by a separate control register. The IO counters are not functional in Hub revisions up to revision 2.1. Subsequent revisions implemented one 20-bit LFSR counter, which is incremented when either the first or second selected event is seen. Table 5.3 lists a subset of events that can be counted with Hub IO counters. Just like Hub MD counters, these counters are grouped together in a separate 16 KB page.

select	*description*
`0xF0`	Micropackets transmitted to the XBow
`0x0F`	Micropackets received from the XBow
`0xF1`	Data-only micropackets received from XBow
`0x1F`	Data-only micropackets transmitted to the XBow
`0xF2`	Micropackets transmitted to the crossbar
`0x2F`	Micropackets received from the crossbar
`0xF3`	Data-only micropackets received from the crossbar
`0x3F`	Data-only micropackets transmitted to the crossbar

Table 5.3: Hub IO event counter definitions

The micropackets are internal 64-bit messages sent or received from the Hub IO unit to the crossbar (for a processor, memory, or the network interface) or on the Crosstalk interface that connects the Hub to the I/O bridge. The number of micropackets in each direction need not be the same: the IO unit provides two block transfer engines (BTEs) that copy cache lines between memory locations; the cache lines copied by the BTEs are sent between the IO unit and the crossbar only whereas the I/O data go on the Crosstalk interface.

5.2.3 Router Histogram Counters

The Router ASIC maintains a set of four counters for each link. The counters are 16 bits wide, and they can all be accessed by a single 64-bit read operation. The counters can operate in three modes. In the link utilization mode the counters are incremented whenever a micropacket is sent, received, or granted a bypass. The DAMQ histogram mode gives an estimate of link contention by counting the number of micropackets on a particular virtual channel waiting to be sent to the outgoing link. In the last mode, the counters measure a histogram of packet ages. Table 5.4 gives the description of each counter.

counter	*link utilization*	*DAMQ histogram*	*age histogram*
0	bypass utilization	0 entries used	0–63
1	receive utilization	1–7 entries used	64–127
2	send utilization	8–14 entries used	128–191
3	50 MHz counter	15 entries used	192–255

Table 5.4: Router histogram counter registers

The 50 MHz counter runs at full link speed. The send and receive utilization counters are incremented whenever a micropacket is sent to or received from the link. The bypass utilization counter is incremented whenever a received micropacket bypasses the age arbitration (the DAMQ queue is empty) and there is no contention for the outgoing port in the internal crossbar. The bypass utilization represents the zero-contention case. When the input queue is not empty, the router will allocate a DAMQ entry for each incoming packet. The DAMQ structure is used to implement fair routing based on packet priorities. The number of active DAMQ entries is a good measure of link contention. Note that the Origin interconnect implements four virtual channels. Packets on the same virtual channel are delivered in-order; however, packets on different virtual channel can be delivered out of order. The DAMQ histogram is computed for a selected virtual channel only. Packets are routed based on their priorities—higher age means higher priority. Packet age is incremented while waiting in the DAMQ queue. The age histogram can be used to evaluate the efficiency of the aging algorithm. In practice, packet ages are not important on small system configurations.

At 50 MHz, a 16-bit counter will overflow in 1.3 ms. Since the latency of accessing Router registers is relatively high, the counters operate in a histogram mode: whenever one of the counters overflows all four counters are divided by two. This operation loses the exact count, but maintains the correct ratio between the counters. This makes it possible to reduce the sampling rate and still get accurate utilization ratios.

5.3 Implementation

The ccNUMA memory profiler uses the information from the Origin hardware event counters to compute various thread, memory and network metrics. Not all of the metrics discussed in Section 5.1 can be computed from the Origin event counters. The follow-on system, Origin 3000, implements new hardware event counters, which can be used to compute additional metrics.

The memory profiler is used in a manner similar to the SpeedShop tools[51, 46]. 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` [54] or `jgraph` [31] 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 resulting loadable kernel module is described in Section 5.3.1. The system activity monitor is described next in Section 5.3.2, together with a discussion of implemented node and network metrics. Section 5.3.3 describes the implementation of the application launcher and a discussion of the implemented thread metrics. Finally, Section 5.3.4 describes the post-mortem analysis tools.

5.3.1 Loadable Kernel Module

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, though.

The loadable kernel module creates a device file for each source of hardware event counters in the system. The device files are created in the `/hw` filesystem [38]. There are three kinds of devices: *map* devices are created for registers that can be mapped into user space by means of a `mmap` system call; *link* devices map event counters which reside on the Router ASIC, and a special operation (the *vector read/write* operations) is required to access them. *Rreg* devices are similar to *link* devices; they are created as part of the extended LKM interface and provide an interface to the complete router register space. Table 5.5 summarizes the device files created when the loadable kernel module is loaded into the kernel.

All devices implement `open`, `close`, and `ioctl` system calls; additionally, the *map* devices implement `mmap` and `munmap` system calls. The `ioctl` interface is used to obtain device infor-

path	*device*	*description*
`/hw/.../node/hub/md`	*map*	Hub memory/directory counters
`/hw/.../node/hub/io`	*map*	Hub IO counters
`/hw/.../node/hub/ni`	*link*	alias for the node outgoing link
`/hw/.../router/stat/[1-6]`	*link*	link histogram counters

Table 5.5: Device files created by the LKM

mation, access *link* counters, and set device parameters. Table 5.6 shows the `ioctl` commands implemented by the kernel module and the devices to which they apply.

command	*device*	*description*
`SNPC_QUERY`		Get device information
`SNPC_HIST_POLL`	*link*	Read link histogram counters
`SNPC_HIST_READ`	*link*	Read and clear link histogram counters
`SNPC_HISTSEL_GET`	*link*	Get link histogram counting mode
`SNPC_HISTSEL_SET`	*link*	Change link histogram counting mode
`SNPC_VECOP`	*rreg*	Issue a vector read/write operation
`SNPC_VECTBL`	*link, rreg*	Get vector route to the router

Table 5.6: The `ioctl` commands defined by the LKM

All devices support the `SNPC_QUERY` command, which returns device information in the `snpc_query_t` structure. The `snpc_id` field is used to distinguish various devices exported by the loadable kernel module. The LKM assigns an unique identification to each device type; this is necessary to properly implement sampling in user space and to distinguish similar sources of performance data between generations of Origin systems. The `snpc_base` and `snpc_size` fields give the physical address and the size of the mappable address space; they only apply to *map* devices. The `snpc_master` field points to the master device, either a Hub or a Router. The `snpc_port` field gives the Router port number (1–6) that this device refers to. The `snpc_route` field gives a vector route from the node where the process issuing the `ioctl` call was running to the Router to which the device refers to. Both `snpc_port` and `snpc_route` are only valid for *link* devices.

The `SNPC_HIST_POLL` and `SNPC_HIST_READ` commands return the values of all four Router histogram counters in the `link_hist_t` structure. Both commands return the current snapshots of the hardware counters; the `SNPC_HIST_READ` command also clears the counters. The vector operations have a relatively long latency, compared to the uncached load and store operations that are used to access the Hub registers: the typical latency for router reads is around 5 μs, whereas the local Hub registers can be accessed in 300 ns. The overhead is due to the length of the call path and the latency of vector routing, which is slower than normal table-driven routing. Besides link utilization, the Router histogram counters support two additional modes of operation: packet age breakdown and DAMQ utilization. Commands `SNPC_HISTSEL_GET` and `SNPC_HISTSEL_SET` may be used to query/set Router histogram counting mode. By default, when the *link* device is opened the LKM selects link utilization mode.

The IRIX OS does not maintain a table of vector routes from each node to all the routers in the system. Since a user process that accesses router histogram counters can run on any processor in the system, the loadable kernel module needs to compute the full matrix of vector routes in the system. The matrix is constructed when the LKM is installed in the kernel. The algorithm first constructs a list of all objects (nodes and routers) in the system and then uses a breadth-first search algorithm to construct the shortest path from each router to all the nodes. Rows of this matrix represent routers and columns represent node numbers. The `SNPC_VECTBL` command returns a row of this matrix for a router where the device is located. The returned array holds vector routes indexed by the node number. This command is intended for debugging purposes only.

Accessing Hub Event Counters

The event counters in Hub memory/directory and I/O interfaces are accessed by mapping the physical address of the event registers into user space. Figure 5.1 shows a code fragment that opens and initializes the Hub MD counters.

Function `hubmd_open` opens the Hub MD device for node *cnid*, maps the registers into user space, and returns a pointer to the `hubmd_regs_t` structure, which defines the layout of the Hub MD counters. First, a path to the Hub MD device is constructed at line 10. If the open is successful, the `SNPC_QUERY` call returns the information about the device; the assertions at lines 17–18 make sure that the device corresponds to Hub MD event counters, and that the device supports the `mmap` call. Lines 19–23 map a section of the physical memory that contains the counters into process address space; the size of the mapped area is provided by the `snpc_size` field. Upon a successful completion of the `mmap` call, the *addr* variable points to the start of the mapped area. The cast at line 24 computes the virtual address of the `hubmd_regs_t` structure, which describes the physical layout of the event counting registers. The Hub MD event counter interface consists of one control register and six counter registers. The pointer dereferences at lines 25–27 are translated into uncached references to the physical registers. Their effect is to stop event counting and clear the counter registers.

Accessing Router Histogram Counters

The interface to the Router histogram counters is similar to that of the Hub registers. The only difference is that the histogram counters require an `ioctl` call to read the values, instead of pointer dereferences. Figure 5.2 shows a code fragment that opens and initializes the link device.

The construction of the device path at line 8 and the initialization at lines 9–14 is similar to the code in Figure 5.1. The assertions at lines 15–16 check for a link device, instead of a Hub MD device. The `ioctl` call at line 17 reads and clears the histogram counters; the appropriate vector read and write operations are issued by the driver code in the kernel, which stands in sharp contrast to the pointer dereferencing used to access Hub event counters.

Extended LKM Interface

The devices created by the regular LKM interface allow access to the event counters in the Hub and Router ASICS. The interface is safe to use by non-privileged programs. The physical memory

```
#include <snpc.h>

hubmd_regs_t* hubmd_open(int cnid)
{
    int fd;
    void* addr;
    size_t size;
    snpc_query_t info;
    hubmd_regs_t* regs;
    char path[PATH_MAX];

    sprintf(path, "/hw/nodenum/%d/%s/%s", cnid,
        SNPC_EDGE_HUB, SNPC_EDGE_MD);

    if ((fd = open(path, O_RDWR)) < 0)
        return 0;

    if (ioctl(fd, SNPC_QUERY, &info) < 0) {
        close(fd);
        return 0;
    }

    ASSERT(info.snpc_id == SNPC_SN0_HUBMD);
    ASSERT(info.snpc_flags & SNPC_MAP);

    size = info.snpc_size;
    addr = mmap(0, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    if (addr == MAP_FAILED)
        return 0;

    regs = (hubmd_regs_t*)
               ((char*) addr + (info.snpc_base & (size-1)));

    regs->sel = 0;
    for (i=0; i < HUBMD_COUNTERS; i++)
        regs->count[i] = 0;

    return regs;
}
```

Figure 5.1: Sample code fragment opening and initializing a Hub MD device

```
#include <snpc.h>

int hubni_open(int cnid)
{
    int fd;
    link_hist_t hist;
    snpc_query_t info;
    char path[PATH_MAX];

    sprintf(path, "/hw/nodenum/%d/%s/%s", cnid,
       SNPC_EDGE_HUB, SNPC_EDGE_NI);

    if ((fd = open(path, O_RDWR)) < 0)
        return -1;

    if (ioctl(fd, SNPC_QUERY, &info) < 0) {
        close(fd);
        return -1;
    }

    ASSERT(info.snpc_id == SNPC_SN0_LINK);
    ASSERT(info.snpc_flags & SNPC_LINK);

    if (ioctl(fd, SNPC_HIST_READ, (void*) &hist) < 0) {
        close(fd);
        return -1;
    }

    return fd;
}
```

Figure 5.2: Sample code fragment opening and initializing a link device

mappings are restricted to Hub event counters, and regular `ioctl` calls can access Router histogram counters only. However, the same mechanism that is used to access a subset of Hub and Router counters can be extended to the whole physical I/O address space. The loadable kernel module has a compile-time option that allows LKM to create additional devices in the `/hw` filesystem, which provide access to the full Hub and Router I/O register space. Table 5.7 shows devices that are part of the extended LKM interface. The `rreg` devices are used to access full Router register space by means of the `SNPC_VECOP` command. The `hubreg` and `ialias` devices provide a mapping of the Hub local register space (see [49, pp. 28–31] for details).

path	*device*
`/hw/.../router/rreg`	Router register space
`/hw/.../node/hub/hubreg`	Hub register space
`/hw/.../node/hub/ialias`	Hub register space on the local node

Table 5.7: Device files in the extended LKM interface

The extended interface was added to the LKM to allow access to the system routing tables. The Irix OS does not provide an interface that returns static routes used for regular packet routing in the Craylink interconnect; a user-level program `snroute` uses the extended LKM interface to read Hub and Router routing tables and prints the routes between nodes. However, the extended LKM interface does not provide an adequate access control mechanism which would make it safe for general use. The interface allows any program that can open the `rreg` device read access to the entire Hub I/O register space. Since even reads to I/O space can have side effects, the extended LKM interface is inherently unsafe and should be used by experienced users only.

5.3.2 System Activity Monitor

The `snsar` program uses the loadable kernel module interfaces to the Origin hardware event counters to continuously print the values of various node and network metrics. All sampling is done in user space: the hardware event counters are virtualized into 64-bit values, and the sampling interval is chosen to be fast enough to prevent any hardware counter overflow. Table 5.8 shows the options recognized by `snsar`. The the options are divided into three groups: the list of objects to be monitored, the list of metrics to be measured, and a set of general options which control the sampling process. The options are followed by two arguments: the first one specifies the time between value updates on standard output; the optional second argument specifies the number of update steps.

The `snsar` program can monitor two kinds of objects: nodes and router ports. Nodes are specified either as compact node ids or as paths in the `/hw` filesystem. Router ports are specified by the `/hw` path to the router followed by the port number. Any number of nodes and ports can be monitored at the same time. The cost of uncached references used to access the hardware registers is relatively low (local Hub registers can be accessed in ≈ 300 ns). Sampling is controlled by two options: the `clock` option specifies the timer source, which is used to implement periodic sampling process wakeup; the `sample` option specifies the sampling interval. The IRIX operating system implements a regular realtime clock that can be used by any process; the realtime clock resolution is 10 ms. Privileged processes can use the fast realtime clock, whose resolution is limited

option	*description*
`--node n`	Print metrics for node n
`--port path:n`	Print metrics for port n on router `path`
`--clock clk`	System clock to be used for sampling
`--sample ms`	Sampling interval in milliseconds
`--runon n`	Bind `snsar` process to CPU n
`--rtprio[=p]`	Schedule `snsar` process with realtime priority p
`--output file`	Save metric data on `file`

Table 5.8: Options recognized by the system activity monitor

```
1  rapture$ snsar -n1 --mem --mdidle --mdbusy 1 5
2
2  IRIX64 rapture 6.5 10120733 IP27  Thu Jan  4 20:53:58 2001
3
3  n1 = /hw/module/11/slot/n2/node
4
4  mem      = memory utilization
5  mdidle   = memory idle cycles
6  mdbusy   = memory busy cycles
7
7  time  loc       mem   mdidle   mdbusy
8  0.003 n1      1.890 98108158  1889585
9  1.003 n1      1.973 98021138  1972694
10 2.003 n1      1.985 98008426  1985296
11 3.003 n1      2.012 97982060  2011668
12 4.004 n1      1.978 98015526  1978196
13
13 [0.003,5.004]: 500 ticks, 10.00 ms/tick, 9.60 us/sample
```

Figure 5.3: Sample invocation of `snsar`

only by the operating system processing overhead; when the sampling process is bound to a specific processor, the observed clock resolution with sampling overhead was as low as 100 microseconds.

The sampling process can be bound to a specific processor: option `runon` binds the `snsar` process to the specified processor. In this way, the sampling activity can be moved away from the portion of the system where the application is running, which avoids scheduler interference. Binding a process to a specific processor also reduces the OS overhead involved with processing a clock interrupt and waking up the `snsar` process, which increases sampling accuracy. Another option to increase sampling accuracy is to put the `snsar` process in the realtime scheduling class, which puts the process priority above all time-shared processes. The realtime priority is available only to privileged users.

It is possible to save the `snsar` output to a file; the resulting binary file can be viewed with the `sndump` and `sninfo` commands, described in Section 5.3.4.

Figure 5.3 shows a sample invocation of the `snsar` command. The output shows five one-second snapshots of the memory utilization on node 1, along with the number of idle and busy

cycles. The preamble starts at line 2, which shows the system information (similar to the `uname` output) and the current time. The list of monitored objects starts at line 3: selected node 1 is shown with the full path in the `/hw` filesystem. Lines 4–6 complete the preamble output; they show a list of metrics to be printed on standard output, together with a brief description of each metric. The remaining lines show one-second updates of the three metrics measured on node 1. The results were obtained on a mostly idle system. The memory utilization is less than 2% (busy cycles are mostly due to memory refresh which is typically around 1.8%). The sum of idle and busy cycles shows that the Hub on node 1 runs at 100 MHz. The last line printed by `snsar` gives the sampling summary.

Implemented Node Metrics

Memory utilization is derived from the first set of Hub MD counters. In this mode, the event counters count Hub cycles in which the memory controller was idle, blocked due to no output queue space, busy refreshing DRAM memory, busy serving requests which involve directory lookup only (such as revision messages), busy serving requests which involve memory and directory access, and busy due to miscellaneous conditions. The memory utilization is defined as the fraction of time when the memory controller was not idle. The busy time is not proportional to the number of incoming requests: the directory controller is pipelined—multiple memory requests can be served simultaneously.

Incoming memory requests can be broken down into read requests, write replies, revision messages that update directory state but do not return any data, and other miscellaneous requests. Additionally, it is possible to collect the statistics about fetch-and-op requests, such as the total number of fetch-and-op requests and the hit rate in the small fetch-and-op cache in the Hub.

The breakdown of the outgoing message traffic can be used to determine the distribution of ccNUMA transaction types. It is possible to determine the fraction of requests that result in a backoff reply or a negative acknowledgement. It is also possible to compare the number of data responses to the number of intervention and invalidate messages. A separate Hub MD mode can be used to break down intervention and invalidate traffic. However, it is not possible to distinguish between clean-exclusive and dirty-exclusive transactions. The home node sends out an intervention and a speculative reply. It is up to the remote processor to determine whether it has dirty data in its cache. Additionally, it is not possible to distinguish outgoing traffic based on whether the original request was local or remote. In the absence of the counters in the Hub PI section, it is not possible to compute per-thread transaction breakdown, only the application totals.

The read transaction breakdown can also be approximated by looking at the directory state of the cache line. Requests for unowned lines and requests for exclusive lines issued by the current owner always result in simple two-step transactions. Requests for exclusive lines owned by a processor other than the current owner result in three-step intervention transactions. Read requests for shared lines result in a simple two step transactions, while read-exclusive requests for shared lines result in invalidate transactions. Unfortunately, it is not possible to get an estimate of the number of invalidate messages generated by invalidate transactions.

The last Hub MD counting mode gives a breakdown of requests generated by local processors and I/O. It is possible to separately count the number of read and writeback requests generated by local processors and the number of read and write invalidates generated by the local I/O. It is not possible to get the number of requests generated by remote nodes or to determine which

transactions resulted from incoming local requests.

All the metrics described so far can be derived from the Hub MD counters. The counter facility in the Hub I/O interface can be used to measure the amount of data traffic between the crossbar and the I/O interface, and between the I/O interface and the Xtalk port. Due to Hub IO performance counter implementation limits, traffic can be measured in only a single direction.

It is not possible to measure the SysAD utilization: the only accurate way of establishing the SysAD traffic would be to have a counter facility in the processor interface section. The SysAD traffic can be approximated by having each thread count the number of cache misses that are translated into SysAD requests. However, `snsar` has no knowledge of threads. The SysAD traffic can be estimated with the application profiler, as described in Section 5.3.3.

While there are no performance counters in the Hub network interface, the node link utilization can be deduced from the Router counters for the router port that connects the node to its nearest router. The loadable kernel module creates the `/hw/.../node/hub/ni` alias to provide an easy way for the sampling process to measure a node's incoming and outgoing link utilization. The `snsar` program measures the node input and output link utilizations, which are simply the send and receive utilizations for the router port that corresponds to the `ni` alias.

Implemented Network Metrics

The `snsar` program can independently monitor all ports on every router in the Origin system. The ports are specified as the `/hw` path to the router followed by the port number. Unfortunately, it is not possible for the `snsar` to automatically determine the portion of the interconnect network that connects a set of nodes—the topology has to be inferred from the `topology` command [47]. For each port, the following metrics can be collected:

- The *link send utilization* gives the percentage of the total link bandwidth used by the outgoing traffic.
- The *link receive utilization* gives the percentage of the total link bandwidth used by the incoming traffic.
- The *bypass utilization* gives the percentage of received messages that were able to use the fast path through the router. Messages receiving the bypass grant have the pin-to-pin router latency of 40 ns; when the bypass is not granted the latency increases to 60 ns. The bypass utilization is a good initial estimate of the link contention.
- The *DAMQ histogram* for a given virtual channel can be used to get a more detailed profile of the link contention.
- The *message age histogram* can be used to collect data about packet aging.

All network metrics are also available as raw event counters. To obtain accurate counts, the router histogram counters have to be sampled at $\approx$ 1 ms intervals. The `snsar` program will attempt to use the high-precision clock in order to obtain accurate values. However, the high-precision clock is available to privileged users only. Regular users can still sample router histogram counters: the ratios are accurate, even if the raw counts are not.

5.3.3 Application Launcher

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 described in Section 5.3.4.

The `snrun` program takes all the options accepted by `snsar`: users needs to specify all the nodes and ports where they expect that the application will execute, along with the metrics in which they are interested. It is also necessary to specify the sampling interval length: this is the time between two consecutive snapshots of the hardware event counters stored in the trace file. The `snrun` program additionally accepts options which trigger the collection of thread metrics. The remaining arguments to `snrun` are interpreted as the command to be executed.

Figure 5.4 shows a sample application profiling session. Option `-r1` binds the snrun process to the processor 1 (on node 0). The sampling interval is 50 ms (`-s50`); the trace file will contain event counters from nodes 1 and 2 (`-n1 -n2`); and the `snrun` program will collect all event counters needed to compute the local memory access ratio (`-alocal`). The rest of the command line specifies the command to be run by `snrun`: the `snbench` command places home memory on node 1 and creates two threads, one on processor 3 (on node 1) and another on processor 5 (on node 2). Lines 2–4 show the output of `snbench`. The other output was generated by the thread wrapper: for each thread, the wrapper prints the file name where the traces have been stored, along with a short summary of the trace file.

The Thread Wrapper

The thread wrapper is a dynamic shared object (DSO). The `snrun` program adds the thread wrapper to the list of DSOs that form the address space of the application process. As part of process initialization, the run-time dynamic linker will unconditionally call the thread-wrapper initialization function. This mechanism allows for arbitrary code to be executed in the host process, without any host process modifications. The thread wrapper uses interval timers to set up periodic signal delivery. Whenever a signal is delivered, the thread wrapper stores new information about the location where the thread is executing, the program counter that points to the section of code interrupted by the signal, and all R10000 event counters that are needed to compute thread metrics

```
1 rapture$ snrun -r1 -s50 -n1 -n2 --alocal snbench -h1 -l3 -l5 bw-read
2 EXPERIMENT   STAT TID       MIN       MAX       MED       AVG    STDEV  UNIT
3 bw-read      UOWN   3    304.243   306.304    305.26   305.268  0.6203 MB/s
4 bw-read      UOWN   5    263.582   264.967   264.537   264.477  0.4599 MB/s
5 snbench.out.snp.p445554[0.212,1.971]: 35 ticks, 50.01 ms/tick, 60.00 us/sample
6 snbench.out.snp.p443132[0.192,1.971]: 35 ticks, 50.01 ms/tick, 43.20 us/sample
7 snbench.out.snp.e445028[0.057,2.019]: 39 ticks, 50.00 ms/tick, 34.40 us/sample

8 snbench.out[0.014,2.045]: 199 ticks, 10.20 ms/tick, 13.60 us/sample
```

Figure 5.4: Sample invocation of `snrun`

specified on the `snrun` command line. The CPU where the thread is running is needed in the post-mortem analysis in order to correlate the data from R10000 event counters with the Hub counters on the same node. The interrupted program counter can be mapped back into application source code to determine which section of code was executing when the interrupt took place. The R10000 event counter data are correlated with Hub counters to compute thread metrics, such as the ratio of local to remote memory accesses.

In addition to periodic sampling of the R10000 event counters, the thread wrapper traces the creation of new threads and processes. Whenever the application creates a new thread with the `sproc` system call or creates a new process with the `fork` call, the thread wrapper intercepts the call and creates a new trace file for the data collected in the new thread/process. In addition, if the application performs the `exec` call, the new process will include the wrapper DSO in the list of shared libraries, the run-time dynamic linker will map and initialize the thread wrapper, and the sampling process will restart. The ability to intercept thread creation, process creation, and the chaining of executables enables the application launcher to profile applications which use either fine-grained parallelism in a shared address space (the `sproc` model, OpenMP programs) or coarse-grained parallelism in separate address spaces (MPI programs that use the `fork` model). The interception of `exec` calls makes it possible to use manual data placing tools such as `dplace`. The only model that is not supported by the thread wrapper is the POSIX thread model. In Irix, POSIX threads do not have kernel scope; the thread scheduling is done entirely in user space.

Implemented Thread Metrics

When sampling the R10000 event counters, the thread wrapper always includes the processsor number where the thread is executing along with the program counter of the interrupted application in each sample. The processor number enables the post-mortem analysis tools to associate thread events with Hub and Router counters. The program counter makes it possible to correlate the changes in performance metrics with source code locations.

One of the thread metrics that can be computed in the post-mortem analysis is the *local memory access ratio*. This metric gives the percentage of memory references generated by the thread that were satisfied by the local node. The metric is computed by having the thread wrapper sample the number of secondary cache misses, whereas the `snrun` process samples the Hub MD counters to get a breakdown of local requests. The local memory access ratio is computed as the number of read requests reported by the Hub counters divided by the total number of secondary cache misses. It is also possible to compute memory access ratios for all memory requests (combined reads and writebacks); in this case, the thread wrapper also samples the number of quadwords written back from the secondary cache, which is correlated to the number of writeback requests generated by the local processor. The local memory access ratio is very sensitive to the interference of other processes, especially on node 0, because the operating system places many of its data structures there and it uses processor 0 for systemwide periodic tasks.

The post-mortem analysis can also derive the fraction of the total SysAD bus bandwidth used by a thread. The *thread SysAD utilization* is computed from the number of read and writeback requests generated by the thread. Each request transfers a 128-byte cache line over the SysAD bus, and the full bus bandwidth is assumed to be 570 MB/s. While the results reported in Table 4.10 vary slightly for different generations of Origin node boards, using 570 MB/s for the maximum SysAD bandwidth ensures that the utilization does not exceed 100%. This metric does not account

for transactions that transfer two cache lines over the SysAD bus (e.g., dirty-exclusive transactions that discard speculative replies). It is not possible to account for the dirty data pulled out of the secondary cache by intervention and invalidate requests.

Origin hardware event counters do not provide enough information to compute the ccNUMA transaction mix. The Hub MD counters cannot associate directory transactions with the source of the request (local/remote processor) and there are no event counters in the processor interface section of the Hub. For this reason alone it is not possible to compute the average memory access latency. An additional problem in computing average transaction latencies is the unpredictable nature of intervention transactions that depend not just on the distances between transaction participants but also on the SysAD bus timing.

5.3.4 Post-Mortem Analysis

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 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.

Figure 5.5 shows sample output from `sninfo`. The files specified on the command line come from the `snrun` experiment shown in Figure 5.4. Three files are generated by three threads, and the fourth file which the contents of the Hub counters. The output is similar to the preamble printed by `snsar`. It begins with a short description of the system where the traces were collected and the list of monitored objects. The trace files hold samples from two nodes and three threads. The next section of output shows all the metrics that can be computed with the data in the trace files. The list in Figure 5.5 is much longer than the metric which was specified in the `snrun` command line because the post-mortem analysis is able to compute several different metrics from the trace data. In the example above, the `-alocal` option triggered the collection of secondary cache misses and the number of quadwords written back from the secondary cache for each thread. In order to compute the ratio of local and remote accesses, the `snsar` program also counted the number of requests generated by the local processors. From this raw event data, the `sndump` process can derive both the local memory access ratio and the SysAD bus utilization for each thread. Additionally, it is possible to print sampled values of the event counters.

In addition to the preamble, the `sndump` program prints the values of the selected metrics. The output can be restricted by specifying the objects whose metrics are to be printed, or by specifying the desired metrics—by default, `sndump` will print the data from all objects and all metrics that can be derived from the data in the trace files specified on the command line. The output is sorted by increasing timestamp order.

```
rapture$ sninfo snbench.out*

IRIX64 rapture 6.5 10120733 IP27  Thu Jan  4 21:14:38 2001

n1 = /hw/module/11/slot/n2/node
n2 = /hw/module/11/slot/n3/node
t0 = snbench:445028
t1 = snbench:443132
t2 = snbench:445554

lp0      = total requests by local processor 0
lp1      = total requests by local processor 1
lio      = total requests by local I/O
lp0read  = read requests by local processor 0
lp0wback = writeback requests by local processor 0
lp1read  = read requests by local processor 1
lp1wback = writeback requests by local processor 1
lioread  = read requests by local I/O
liowinv  = write invalidate requests by local I/O
pc       = thread location
cpuid    = cpu where thread was running
local    = fraction of local read requests
alocal   = fraction of lcoal read+wback requests
tread    = thread READ requests
twback   = thread WB requests
sysad    = fraction of SysAD bandwidth used
tsysad   = total thread SysAD traffic (bytes)
rsmiss   = R10K secondary data cache misses
rsqwwb   = R10K quadwords written back (scache)

snbench.out[0.014,2.045]: 199 ticks, 10.20 ms/tick
snbench.out.snp.e445028[0.057,2.019]: 39 ticks, 50.00 ms/tick
snbench.out.snp.p443132[0.192,1.971]: 35 ticks, 50.01 ms/tick
snbench.out.snp.p445554[0.212,1.971]: 35 ticks, 50.01 ms/tick
```

Figure 5.5: Sample output from `sninfo`

Chapter 6

Examples

6.1 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 Chapter 5 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 6.1 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 6.2 compares memory and link utilizations. In this `snbench` experiment, the test memory was placed on one node and three threads were placed on neighboring nodes (one per node to

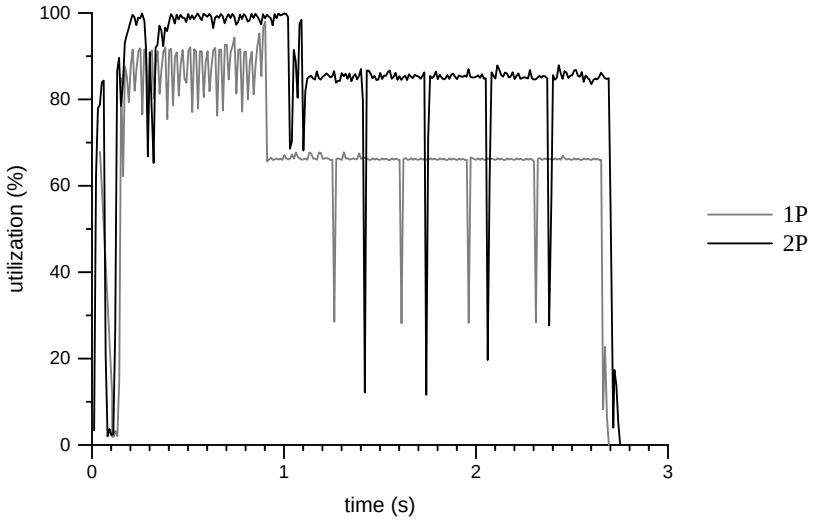


Figure 6.1: Memory utilization for 1- and 2-thread local reduction loop

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 (≈ 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.

6.2 Side Effects of Prefetch Instructions

The Origin cache coherence protocol implements three read requests, each one with different semantics. The `RDSH` request returns the cache line in shared state; the `RDEX` requests returns the cache line in exclusive state; and the `READ` request returns the line in exclusive state unless there are other sharers, in which case the line is returned as shared. The `READ` request helps uniprocessor applications: since the data are not shared, returning an exclusive copy helps if the line subsequently becomes a target of a store instruction; upgrading the cache line from exclusive to modified does not require an external directory transaction. The Hub translates all load instructions into `READ` directory requests. Store instructions receive exclusive ownership via the `RDEX` request. Instruction fetches are translated into `RDSH` requests; this does not pose a problem, even for uniprocessor applications, since the instruction space is treated read-only.

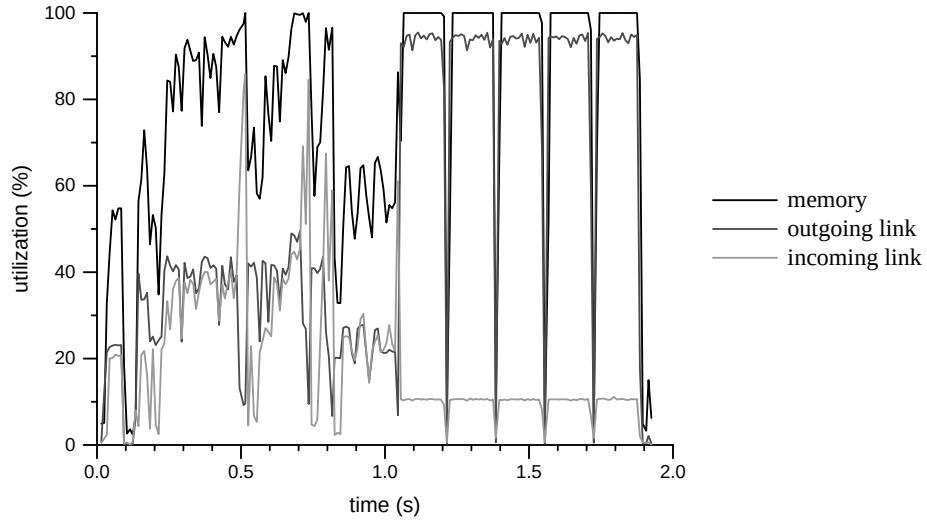


Figure 6.2: A comparison of memory and link utilizations

The MIPS R10000 processor implements four flavors of the prefetch instruction. Two flavors of the prefetch instruction are used to place cache lines in a specific way of the two-way set-associative L2 cache. Of the other two instructions, one is used by the compiler to prefetch data items on the right-hand side of the assignment: a read prefetch is translated into a RDSH request. A write prefetch is translated into a RDEX request; as a side effect of the write prefetch, the R10000 processor places the line in the L2 in the modified state, anticipating that the program will eventually modify the data. The semantics of the read prefetch instruction differs from the semantics of load. RDSH requests generated by read prefetches can hurt application performance in the uniprocessor case.

Figure 6.3 shows the breakdown of directory state for the read/prefetch requests generated by the uniprocessor STREAM benchmark. The first graph was generated with the STREAM code compiled without prefetch instructions, while the second graph shows the profile for the code compiled with prefetch enabled. The first graph shows a mixture of UOWN and EXCL lines, as expected for the uniprocessor application. The second graph shows the same fraction of UOWN lines as the first graph; however, the EXCL lines are mostly replaced with the SHRD lines. Why the difference?

The STREAM benchmark consists of ten itererations of four tests: copy, scale, add, and triad. The following code fragment shows the kernel of each test (loop constructs have been omitted):

```
c[i] = a[i]
b[i] = scalar * c[i]
c[i] = a[i] + b[i]
a[i] = b[i] + scalar * c[i]
```

The UOWN lines are due to the data items written in one kernel and read in the next kernel. For example, *c[i]* is the destination in the first test; since all arrays are larger than the L2 cache, the elements of *c[i]* are written back to memory and the cache lines are placed in the UOWN state. The fraction of UOWN hits is the same both in prefetched and non-prefetched case—the number of

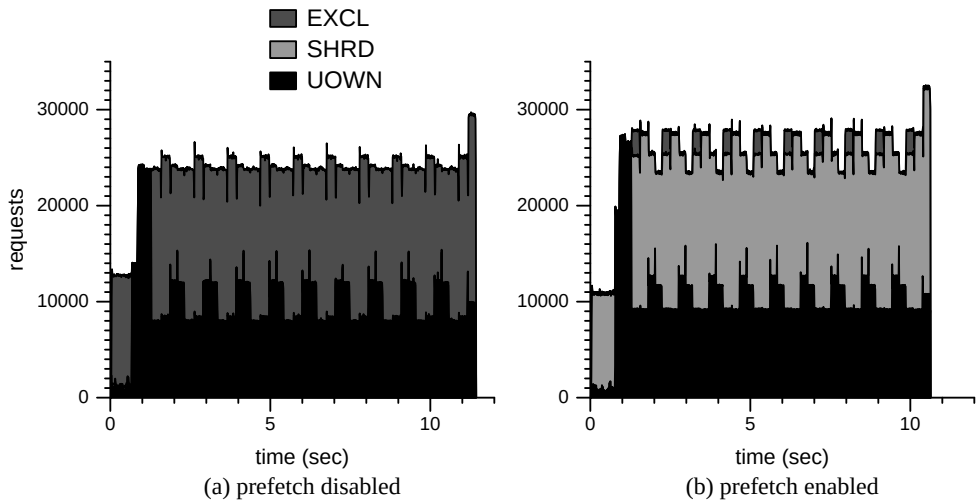


Figure 6.3: STREAM directory state breakdown

data items written back is the same in both cases. The prefetched case replaces a large share of EXCL lines with the SHRD lines—this is due to arrays which were read-prefetched in one of the preceeding kernel: for example, before the start of the third kernel both *a[i]* and *c[i]* will have their data items in SHRD state because they were both prefetched during the execution of the first and the second kernel. The small portion of EXCL hits is a result of data items that were brought into the L2 cache as the result of a load instruction instead of a prefetch.

The problem with placing lines in the shared state for an uniprocessor application is due to the extra invalidate traffic that is generated when the lines in SHRD state are written. The Origin protocol keeps the sharer list on a per-node instead of a per-cpu basis: the RDEX transactions resulting from a store or write-prefetch will have to invalidate the other processor on the node where the STREAM benchmark is running, even though the invalidates are redundant. The extra invalidate (and the corresponding SysAD occupancy) reduces the bandwidth available for STREAM. However, even with the extra invalidate traffic, the obtained STREAM bandwidth is higher than when the prefetch instructions are not used: Figure 6.3 shows that the total run time for case (b) is shorter than case (a).

6.3 Backoff Transactions

The Origin cache coherence protocol reverts to a strict request/reply protocol under heavy load when the system encounters a potential deadlock. The three-step intervention and invalidate transactions become four-step backoff intervention and invalidate transactions. The ccNUMA microbenchmarks described in Chapter 4 do not evaluate backoff latencies and bandwidths, because these transactions should occur very rarely and should not have a noticable impact on memory performance. In order to prove that this is indeed the case, the memory profiler was used to measure the fraction of all directory transactions that revert to a strict request/reply protocol.

We found that the SPLASH FFT and Radix sort kernels up to 32 processors did not produce a single backoff transaction even for large data sets. The FFT case is not surprising: each thread

operates only on a fraction of the input data, which means that the number of sharers for each cache line is relatively small. On the other hand, the Radix sort kernel has to exchange key histograms at the end of each local sort phase; the number of sharers is proportional to the number of threads. Since the histograms are recomputed in every phase the number of invalidations that have to be sent out is potentially quite large. The lack of backoff transactions in Radix sort could occur because the histogram data are exchanged between all nodes simultaneously, so that no single node becomes a hot spot.

We then used the `snbench` microbenchmarks to set up a pathological case where we expected to see backoff transactions. This set of experiments placed all the data on a single node; the sharer threads then all touched all the test data, which had the effect of placing the cache lines in a shared state with the number of sharers equal to the number of threads. After the initialization phase, the threads on each node modified disjoint portions of the test data by issuing store instructions which resulted in a number of invalidate transactions. Since all threads were accessing data on a single node we expected that the home node would eventually have to run out of its output buffering resources and have to revert to backoff invalidate transactions. We used the memory profiler to count both the number of incoming messages (broken into the number of read, writeback, and revision messages) and the number of outgoing messages (again separated into data responses, writeback acknowledges, interventions/invalidations, and backoff interventions/invalidations). The number of threads was increased from two to 14 (the results were taken on a 30P system where two processors were dedicated to the memory profiler and the master `snbench` thread). Table 6.1 shows the observed aggregate read/write bandwidth and the fraction of backoff invalidate transactions.

	zero bandwidth (MB/s)			*read bandwidth (MB/s)*		
threads	*RDEX*	*WB*	*% boff*	*RDEX*	*WB*	*% boff*
4	275	274	0%	385	92	0%
5	273	274	0%	359	70	0%
6	274	272	0%	333	50	4%
7	269	274	5%	340	55	26%
8	274	274	21%	368	49	59%
9	277	273	31%	379	49	59%
10	276	272	38%	395	42	68%
11	278	277	44%	404	36	74%
12	272	269	46%	407	39	77%
13	291	262	56%	407	31	80%
14	289	262	59%	403	30	80%

Table 6.1: Aggregate memory bandwidth for backoff invalidates

The first group of results shows the observed bandwidths when the threads were in the steady state of the `bw-zero` kernel. This kernel modifies cache lines in a large array; each store is designed to replace one cache line, which results in one `RDEX` and one `WB` protocol request. The `RDEX` and `WB` columns show the corresponding read and write bandwidths. (The small variations in the bandwidth results are because the steady state is relatively short (80–200 ms) and the results were computed from a 10 ms sampling interval.) There are no backoff transactions when the number of sharers for each cache line (and the number of threads invalidating them) is six or

less; with seven sharers, there is a small fraction of backoff invalidates; eight or more sharers increases the fraction significantly. With 14 threads invalidating cache lines, the fraction of backoff invalidates increases to almost 60%.

Surprisingly, the backoff invalidate transactions do not seem to have a strong impact on the aggregate node bandwidth: in the *zero* column, the total bandwidth adds to ≈ 550 MB/s, regardless of the number of threads or the fraction of backoff transactions. The bottleneck here is clearly the memory bandwidth out of a single node. Note that the node read/write bandwidth is considerably lower than the read bandwidth (measured to be ≈ 620 MB/s). [1]

Note that the backoff invalidations result from read-exclusive requests. For the writeback requests, the memory/directory interface updates the memory and sends back a writeback acknowledge. The fraction of backoff transactions is lower than what it could be because the writebacks use half the node memory bandwidth; the number of read requests that could be processed is considerably smaller. To estimate the fraction of backoff transactions in the absence of writes, we have computed the read and writeback bandwidths measured during the cache warmup—before the timed portion of the `bw-zero` experiment, the code first modifies all lines in the L2 cache; since the cache is flushed, there are very few writebacks in this stage. The second set of results shows the fraction of backoff messages when the read requests are dominant. The writeback requests have not been entirely eliminated, partly because of cache conflict misses, and partly because the threads do not synchronize between the cache warmup and the timed phase of the `bw-zero` experiment. The second effect is particularly strong with a small number of threads. When the writebacks are mostly eliminated, the four-step backoff invalidations have a noticable impact on the aggregate memory bandwidth, which hovers around 440 MB/s. The fraction of backoff transactions peaks at 80%. Increasing the number of requestor threads is not likely to increase the aggregate bandwidth since the memory seems to be running at its peak. On the other hand, increasing the number of sharers could have an impact on the fraction of backoff transactions because the individual invalidates, generated from a compact intra-Hub invalidate message, increases the outgoing link utilization.

A more realistic case that could generate backoff transactions is the NAS CG parallel benchmark. This code computes $r = bAx$, where A is a distributed matrix. Matrices b, x and r are read by all threads and updated in a distributed manner. Since all cache lines for these matrices are first read by all nodes, the subsequent updates could potentially generate many invalidates. McCalpin reported that the NAS CG benchmark on the Origin systems does not scale beyond 32 processors. [26]

6.4 SPLASH-2 FFT

All the experiments described so far used synthetic microbenchmarks that were used to saturate system resources. 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 [55] to study the application memory requirements and to evaluate three alternatives for a matrix transpose algorithm.

[1] The reduced *zero* bandwidth could be to one dead cycle for SDRAM turnaround after eight cycles of accessing SDRAM.

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 [3], 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 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 6.4 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.

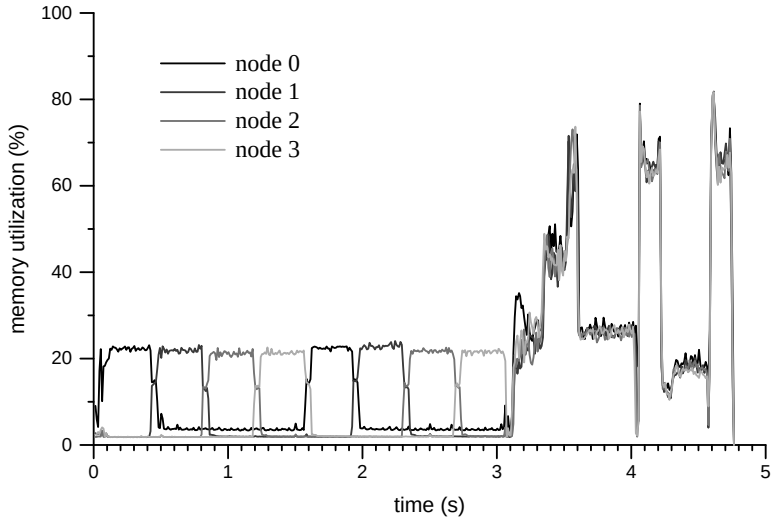


Figure 6.4: FFT memory utilization profile on four nodes

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 6.5–6.7 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 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 6.1). 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 (a 20–25% bandwidth reduction, shown in Table 4.11).

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 6.7, 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

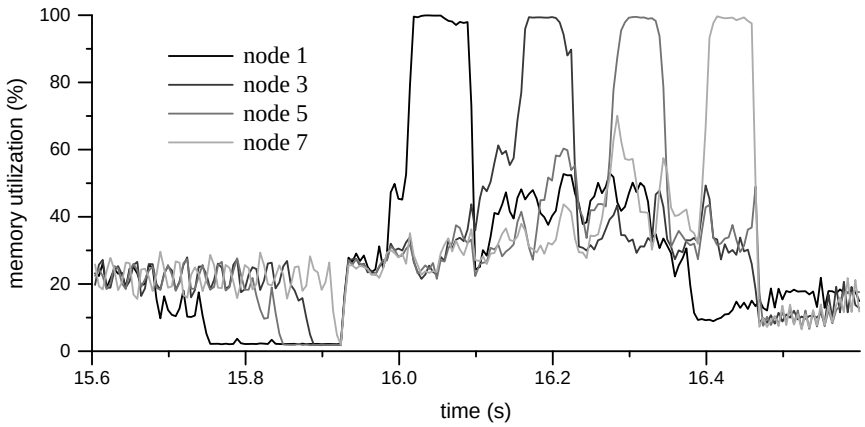


Figure 6.5: Unoptimized FFT matrix transpose without staggering

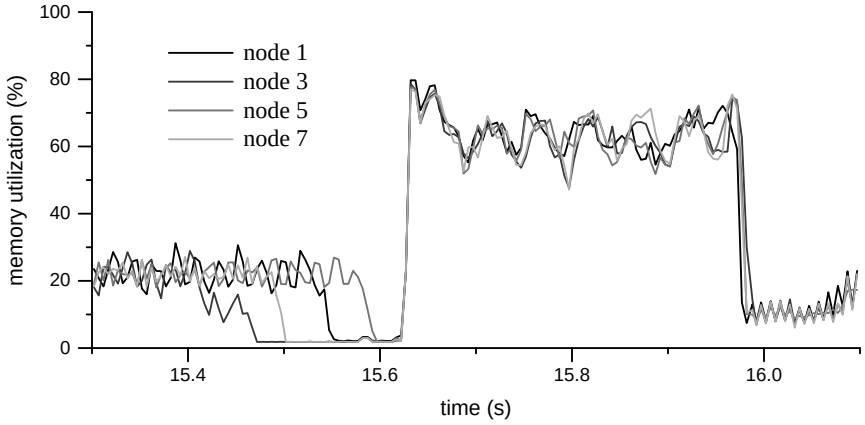


Figure 6.6: FFT matrix transpose with basic staggering

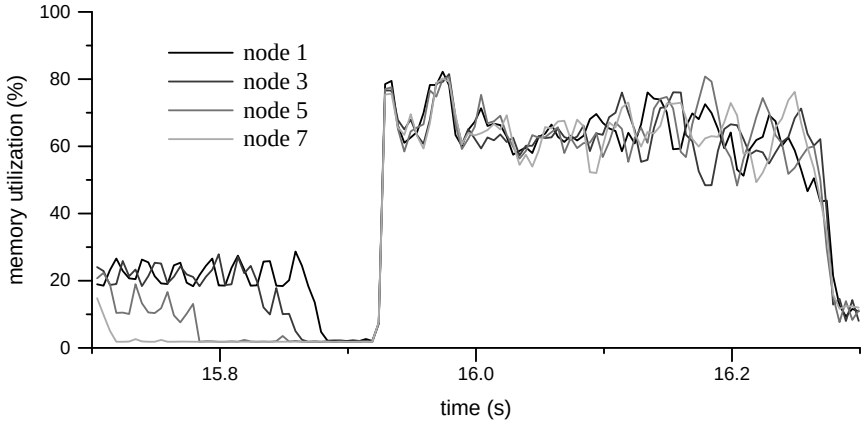


Figure 6.7: FFT matrix transpose with optimized staggering

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. [26] His algorithm used barriers inside the transpose phase. At 64P, the transpose time was 1.6 times local copy time.

Chapter 7

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. The work in this thesis focused on the SGI Origin 2000, one of the first commercially successful large-scale ccNUMA multiprocessors.

We developed a suite of microbenchmarks tailored for the ccNUMA environment and for the directory-based cache coherence protocol used by the Origin. These microbenchmarks were used to analyze the performance of a variety of Origin systems; we focused on memory latency and bandwidth, and how they interact with protocol transactions and design trade-offs. The snbench microbenchmark suite is capable of placing memory and threads anywhere in the system; the test memory can be placed in any of the composite cache coherence states before executing the test kernels—this allows snbench to generate almost every cache-coherence transaction. The latency of each transaction is estimated with two values: the back-to-back latency establishes the upper bound which includes contention for limited resources such as cache and memory bus occupancy; the restart latency defines the minimum time for the processor to issue a request and receive the critical word. We have found that the restart characteristics of the two processors used in the Origin systems, the R10000 and R12000, differ due to the internal implementation details.

The microbenchmarks were used to evaluate a number of different Origin systems. First, we compared the local memory characteristics of four different generations of node boards used in Origin systems. We found that the local results for the 195 MHz R10000 systems matched the results from a previous study [19]; we also present results from 250 MHz R10000 systems, and the 300 and 400 MHz R12000 systems. In the later generations of the node boards, the performance

is limited by the memory system, as opposed to the processor limitations in early systems. The next group of measurements focused on remote transactions, which involve messages sent over the interconnect network. We found that the remote penalty was underestimated in the existing Origin literature and that the router delay is not uniform in all systems, depending on the size of the system (i.e., whether it requres a metarouter or not) and the effects of different cable lengths. Interventions are another group of transactions analyzed with `snbench`. Unlike remote transactions, where we were able to model the latency relatively accurately with just a few parameters, the intervention transactions proved to be very hard to model. The placement of the three participants (local and remote processor and the home node) influenced the results, especially when the local and remote processor were placed on the same node. (The nonuniform results are due to the multiplexed SysAD bus.) We did find placements which exhibited uniform timings: clean-exclusive transactions incur 220–250 ns additional latency compared to similar unowned transactions. Dirty-exclusive transactions incur 330–690 ns extra latency; the timing is much more sensitive to the placement. The last group of transactions evaluated with `snbench` were the invalidations. We found that the invalidate latencies depend on the number of sharers: the additional overhead is 30% for 32 sharers and 50% for 64 sharers. The corresponding bandwidth reductions are even more pronounced.

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 show how STREAM kernels compiled with and without prefetch instructions results in a significantly different execution profile shown by the hardware event counters. We also attempted to evaluate the relative frequency of backoff transactions, which are used by the Origin directory cache coherence protocol to avoid deadlock. We found no evidence of backoff transactions during the execution of SPLASH-2 kernels on systems up to 32 processors. Attempts to generate backoff transactions with artificial thread and data placements show that backoff transactions can occur. However, our measurements were obtained on synthetic codes which involved many sharers and a high number of participating threads. We did not investigate the backoff behavior on other codes which potentially generate large backoff traffic (such as the NAS CG parallel benchmark). Interestingly, the backoff transactions do not have a significant impact for write operations; the impact on read operations is higher. Finally, 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.

Chapter 8

Acknowledgments

This thesis would not have been possible without the help of Silicon Graphics, its engineers and its special corporate culture. My internships at SGI were all unique learning experiences, and I feel lucky to be part of that culture. At SGI, many people helped me understand various aspects of Origin hardware and software design. Jeff Kuskin, Jim Laudon, Greg Marlan, Randy Passint, Doug Solomon, Swami Venkataraman and Mike Woodacre answered many questions I had about the Origin hardware design; Casey Leedom and Len Widra helped me navigate the Irix operating system; Dave Anderson, Jim Dehnert, and Mike Murphy put the compiler in the perspective; Tom Elken kindly allowed me to use the computing resources of his group. Jeff Gibson, a fellow SGI intern, helped me tame the SPLASH-2 benchmarks and discussed ideas about ccNUMA memory performance analysis. SGI also allowed me to put this work in the public domain, and let me to use their illustrations in this thesis. (Figures 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, and 3.10 were taken from the Origin technical manuals. Copyright © 2000 Silicon Graphics, Inc. Used by permission. All rights reserved.)

At the University of Utah, this work was sponsored by the Center for the Simulation of Accidental Fires and Explosions (C-SAFE). Thanks to Steven Parker I was able to use the computing resources of the Scientific Computing and Imaging Institute.

Finally, this thesis would not exist without the members of my thesis committee. Al Davis convinced me that supercomputers were cool, and provided support and guidance during research and writing. John McCalpin pointed out the problem that needed to be solved, and taught me many things about computer benchmarks and performance analysis. Wilson Hsieh's thorough reading of the thesis drafts and his invaluable feedback and support made the process of writing much easier.

Appendix A

128-Processor System Results

Table A.1 presents combined remote and invalidation results collected on `stinger`, a 128-processor 300 MHz R12000 system. Similar to the results in Tables 4.13 and 4.19, the home node is node 1—all the data was allocated there. The first column groups the node number *n* and the distance to the home node *h*. The second column group shows remote back-to-back and restart latencies for each node in the system; the data is in unowned state. Similarly, the third column group gives remote bandwidths for unowned lines. The last two column groups give intervention latencies and bandwidths. In these experiments, the local thread was placed on the home node; the intervention results in each row were obtained by placing lines in shared state where the list of sharers included all the nodes up to the node in the current row. The number of sharers is given in the *#sh* column.

Figure A.1 shows remote results in a graphical form. Figures A.2 and A.3 show intervention latency and bandwidth results, respectively.

Table A.1: Remote and intervention results for a 128P system

stinger			*UOWN latency*		*UOWN bandwidth*		*SHRD latency*		*SHRD bandwidth*		
n	*h*	*#sh*	*b-to-b*	*restart*	*READ*	*zero*	*RDEX*	*UPGRD*	*RDEX*	*UPGRD*	*zero*
1	0	1	384	297	557	266	605	341	327	384	214
0	1	2	763	675	437	232	753	441	285	308	205
2	2	3	908	822	417	242	869	503	251	267	194
3	2	4	909	813	417	226	887	517	244	259	186
4	2	5	915	829	415	238	917	535	237	250	178
5	2	6	919	828	422	235	937	544	227	245	171
8	2	7	916	830	419	228	955	553	220	239	164
9	2	8	917	830	419	237	971	571	211	233	158
6	3	9	1066	979	373	221	1070	639	194	208	150
7	3	10	1067	971	376	221	1092	668	187	200	145
10	3	11	1069	982	371	221	1111	696	181	195	141
11	3	12	1069	981	372	228	1140	717	174	189	136
12	3	13	1077	990	371	225	1168	740	168	182	132
13	3	14	1075	990	373	235	1188	769	161	176	127
16	3	15	1128	1039	355	213	1207	804	155	167	123
17	3	16	1128	1039	356	221	1215	832	149	160	120
32	3	17	1115	1026	357	225	1230	859	144	155	116
33	3	18	1112	1024	358	224	1248	887	139	149	113
48	3	19	1104	1011	362	228	1266	911	134	143	110
49	3	20	1104	1012	362	225	1284	939	130	138	107
14	4	21	1221	1128	337	220	1342	987	125	132	104
15	4	22	1218	1125	337	220	1363	1018	121	127	101
18	4	23	1277	1184	324	214	1391	1047	117	123	99
19	4	24	1277	1183	324	210	1434	1086	114	119	96
20	4	25	1289	1199	322	218	1456	1113	111	115	94
21	4	26	1284	1196	323	220	1476	1146	108	112	92
24	4	27	1277	1191	324	212	1498	1179	105	109	90
25	4	28	1278	1189	324	212	1526	1213	102	106	88
34	4	29	1264	1171	326	209	1549	1252	100	102	86
35	4	30	1262	1172	327	213	1572	1286	97	99	84
36	4	31	1272	1186	326	221	1588	1321	95	97	83
37	4	32	1272	1186	325	213	1608	1356	92	94	81
40	4	33	1271	1183	326	207	1620	1395	90	92	79
41	4	34	1271	1184	326	217	1645	1432	88	89	78
50	4	35	1254	1159	329	214	1663	1464	86	87	76
51	4	36	1254	1165	329	211	1697	1498	85	85	75
52	4	37	1260	1174	329	207	1712	1533	83	83	74
53	4	38	1261	1174	328	220	1724	1569	81	82	72
56	4	39	1261	1173	329	218	1756	1608	79	80	71

stinger			*UOWN latency*		*UOWN bandwidth*		*SHRD latency*		*SHRD bandwidth*		
n	*h*	*#sh*	*b-to-b*	*restart*	*READ*	*zero*	*RDEX*	*UPGRD*	*RDEX*	*UPGRD*	*zero*
57	4	40	1260	1170	329	216	1775	1642	78	78	70
22	5	41	1432	1347	297	204	1811	1688	76	76	68
23	5	42	1431	1346	296	197	1849	1738	74	74	67
26	5	43	1426	1341	297	210	1866	1772	72	72	66
27	5	44	1425	1333	297	201	1893	1803	71	71	65
28	5	45	1436	1345	296	201	1923	1835	69	70	64
29	5	46	1435	1346	296	196	1941	1877	69	69	63
38	5	47	1416	1330	299	203	1967	1915	67	67	62
39	5	48	1416	1327	298	193	1978	1951	65	66	61
42	5	49	1424	1343	296	197	2011	1989	65	65	60
43	5	50	1430	1344	296	204	2026	2015	63	63	59
44	5	51	1430	1344	296	206	2068	2052	63	63	58
45	5	52	1430	1339	297	206	2075	2072	62	62	57
54	5	53	1410	1324	301	215	2111	2119	60	61	57
55	5	54	1438	1317	301	205	2120	2163	59	59	56
58	5	55	1410	1316	302	210	2148	2196	58	58	55
59	5	56	1410	1317	300	202	2167	2236	57	57	54
60	5	57	1418	1327	299	202	2197	2265	56	57	54
61	5	58	1415	1330	299	203	2213	2287	55	56	53
30	6	59	1581	1496	272	188	2281	2352	54	55	52
31	6	60	1581	1486	273	190	2295	2389	54	54	51
46	6	61	1576	1483	274	189	2326	2421	53	53	51
47	6	62	1575	1481	274	197	2349	2464	52	52	50
62	6	63	1569	1483	275	195	2364	2496	51	51	49
63	6	64	1568	1480	275	191	2387	2534	50	50	49

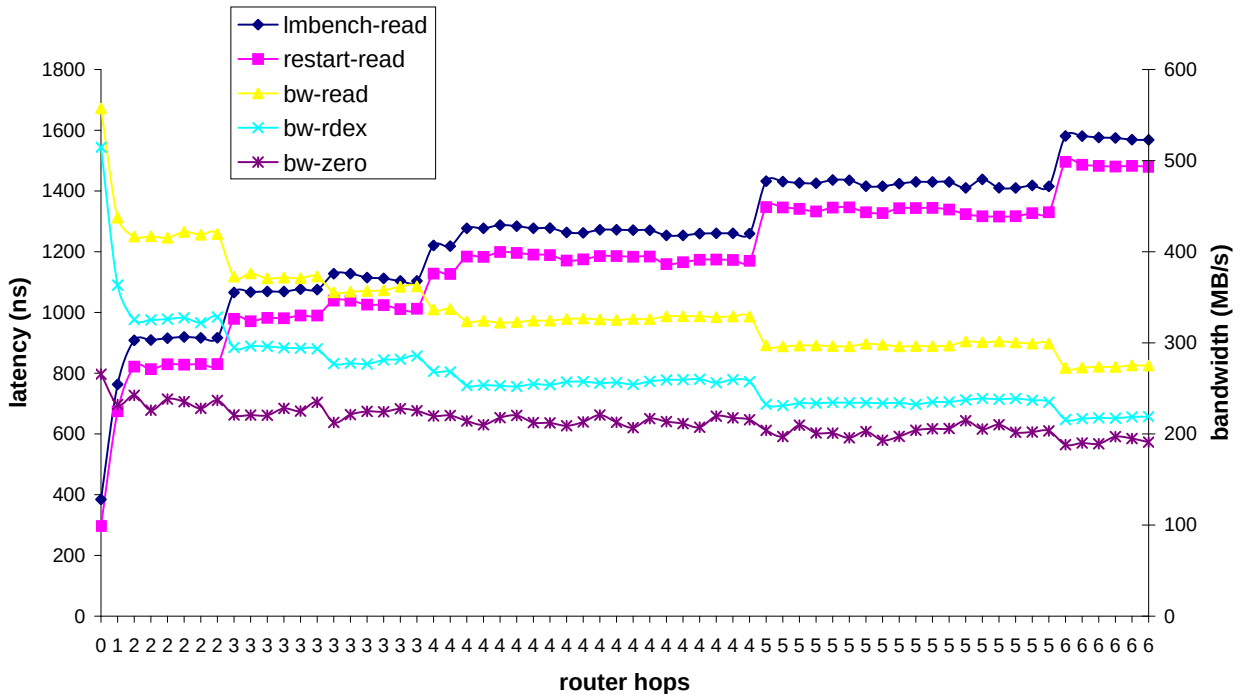


Figure A.1: 128-processor system remote latency and bandwidth chart

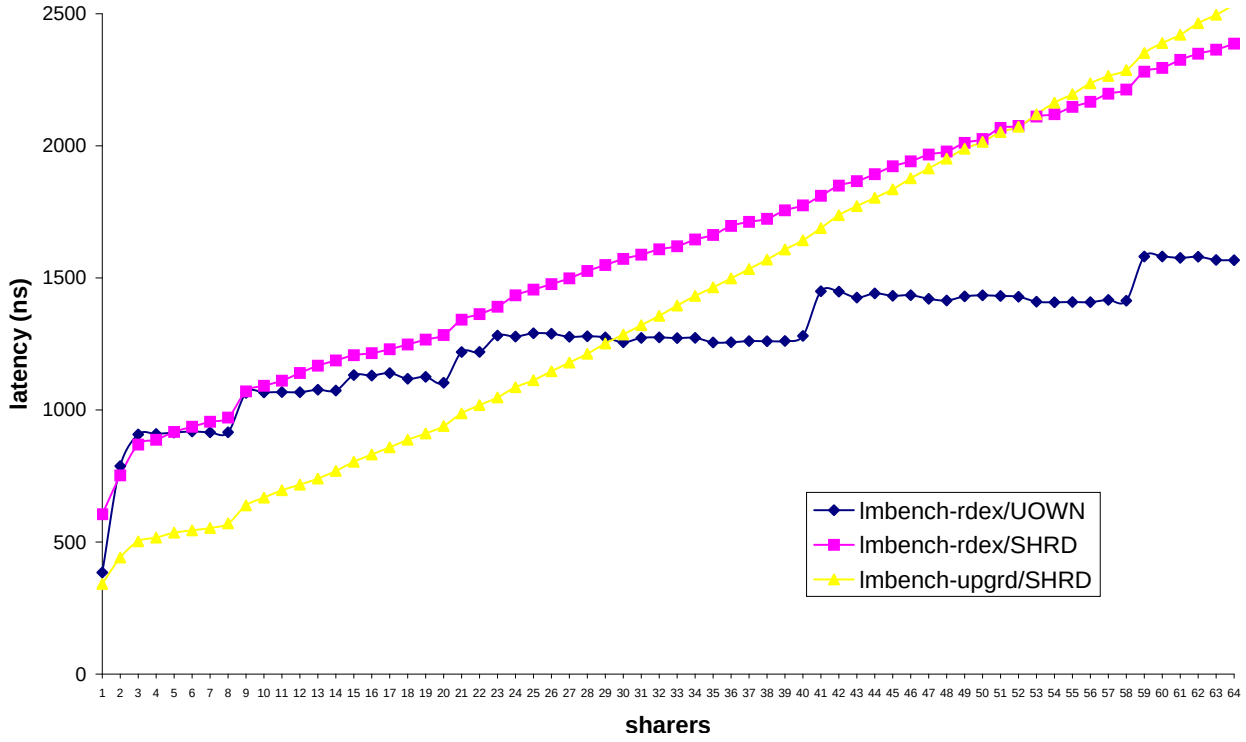


Figure A.2: 128-processor system intervention latency chart

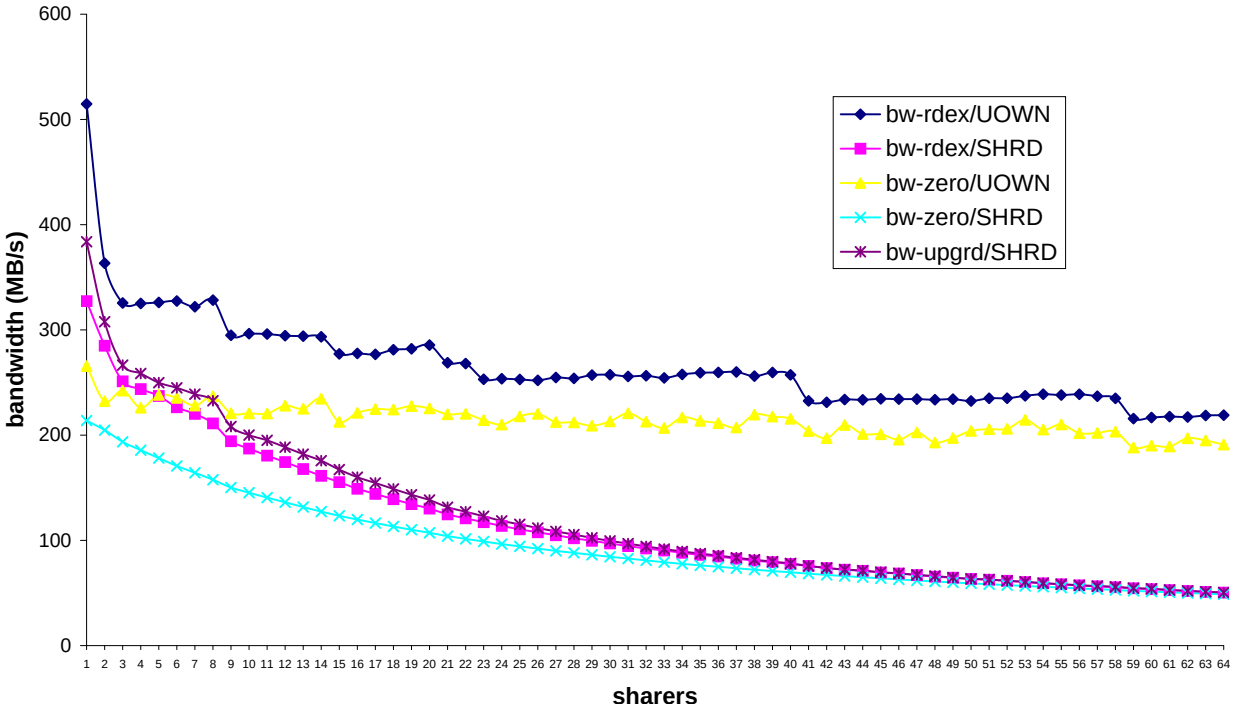


Figure A.3: 128-processor system intervention bandwidth chart

Bibliography

[1] AGARWAL, A., BIANCHINI, R., CHAIKEN, D., JOHNSON, K. L., KRANZ, D., KUBIATOWICZ, J., LIM, B.-H., MACKENZIE, K., AND YEUNG, D. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd International Symposium on Computer Architecture* (May/June 1995), pp. 2–13.

[2] 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.

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

[4] BELL LABORATORIES. *`prof(1)` - display profile data*.

[5] 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`.

[6] CARTER, J. B., BENNETT, J. K., AND ZWAENEPOEL, W. Implementation and performance of Munin. In *Proceedings of the 13th Symposium on Operating Systems Principles* (October 1991), pp. 152–164.

[7] CHANDRA, R., CHEN, D.-K., COX, R., MAYDAN, D. E., NEDELJKOVIC, N., AND ANDERSON, J. M. Data distribution support on distributed shared memory multiprocessors. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation* (Las Vegas, NV, June 1997).

[8] CLARK, R., AND ALNES, K. An SCI chipset and adapter. In *Symposium Record, Hot Interconnects IV* (August 1996), pp. 221–235.

[9] CONVEX COMPUTER CORPORATION. *Exemplar Architecture*, 1993.

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

[11] 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).

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

[13] FRANK, S., BURKHARDT III, H., AND ROTHNIE, J. The KSR1: Bridging the gap between shared memory and MPPs. In *Proc. COMPCON, Digest of Papers* (1993), pp. 475–480.

[14] GALLES, M. Scalable pipelined interconnect for distributed endpoint routing: the SGI SPIDER chip. In *Hot Interconnects '96* (1996).

[15] GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. gprof: A call graph execution profiler. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction '82* (Boston, MA, June 1982), pp. 120–126.

[16] GUSTAFSON, P. E. "MUTT: Memory utilization tracking tool". `http://www.lanl.gov/orgs/cic/cic8/para-dist-team/mutt/muttdoc.html`.

[17] GUSTAVSON, D. The scalable coherence interface and related standards project. *IEEE Micro 12*, 1 (1992), 10–22.

[18] HAGERSTEN, E., LANDIN, A., AND HARIDI, S. DDM—a cache only memory architecture. *IEEE Computer 25*, 9 (September 1992), 44–54.

[19] HRISTEA, C., LENOSKI, D., AND KEEN, J. Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks. In *Proceedings of Supercomputing '97* (San Jose, CA, November 1997).

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

[21] KELEHER, P., COX, A. L., DWARKADAS, S., AND ZWAENEPOEL, W. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of 1994 Winter USENIX* (January 1994), pp. 15–132.

[22] KUSKIN, J., OFELT, D., HEINRICH, M., HEINLEIN, J., SIMONI, R., GHARACHORLOO, K., CHAPIN, J., NAKAHIRA, D., BAXTER, J., HOROWITZ, M., GUPTA, A., ROSENBLUM, M., AND HENNESSY, J. The stanford flash multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture* (Chicago, IL, Apr. 18–21, 1994), pp. 312–313.

[23] LENOSKI, D. *The Stanford DASH Multiprocessor*. PhD thesis, Computer Systems Laboratory, Stanford University, 1992.

[24] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 7*, 4 (1989), 321–359.

[25] LOVETT, T., AND CLAPP, R. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture* (May 1996), pp. 308–317.

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

[27] MCCALPIN, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter* (December 1995).

[28] MCVOY, L., AND STAELIN, C. lmbench: Portable tools for performance analysis. In *Proceedings of 1996 Winter USENIX* (San Diego, CA, January 22-26, 1996).

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

[30] PAPAMARCOS, M., AND PATEL, J. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th International Symposium on Computer Architecture* (June 1984), pp. 348–354.

[31] 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.

[32] REINHARDT, S. K., LARUS, J. R., AND WOOD, D. A. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st International Symposium on Computer Architecture* (April 1994), pp. 325–337.

[33] SAAVEDRA, R. H. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. PhD thesis, University of California, Berkeley, 1992. Tech. Rept. No. UCB/CSD 92/684.

[34] SAAVEDRA, R. H., GAINES, R. S., AND CARLTON, M. J. Micro benchmark analysis of the KSR-1. In *Proceedings of Supercomputing '93* (Portland, OR, November 1993), pp. 202–213.

[35] SAAVEDRA, R. H., AND SMITH, A. J. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems 14*, 4 (1996), 334–348.

[36] SAULSBURY, A., WILKINSON, T., CARTER, J., AND LANDIN, A. An argument for simple COMA. In *Proceedings of the First IEEE Symposium on High Performance Computer Architecture* (January 1995), pp. 276–285.

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

[38] SGI. `hwgraph(4)` - hardware graph. Man page.

[39] SGI. `hwgraph_intro(D3X)` - hardware graph overview for device driver writers. Man page.

[40] SGI. `libfetchop(3)` - atomic synchronization. Man page.

[41] SGI. `migration(5)` - dynamic memory migration. Man page.

[42] SGI. `mmci(5)` - memory management control interface. Man page.

[43] SGI. `perfex(1)` - Command line interface to processor event counters. Man page.

[44] SGI. `r10k_counters(5)` - Programming the processor event counters. Man page.

[45] SGI. `replication(5)` - memory replication. Man page.

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

[47] SGI. `topology(1)` - machine topology information. Man page.

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

[49] SGI. *Origin and Onyx2 Programmer's Reference Manual*, 1996. Document no. 007-3410-001.

[50] SGI. *Origin and Onyx2 Theory of Operations Manual*, 1997. Document no. 007-3439-002.

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

[52] STALLINGS, W. *Data and Computer Communications*. Macmillan Publishing Co., 1988.

[53] TAMIR, Y., AND CHI, H.-C. Symmetric crossbar arbiters for VLSI communication switches. *IEEE Transactions on Parallel and Distributed Systems 4*, 1 (1993), 13–27.

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

[55] 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).

[56] YEAGER, K. C. Personal communication.

[57] ZAGHA, M., LARSON, B., TURNER, S., AND ITZKOWITZ, M. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of Supercomputing '96* (Pittsburgh, PA, November 1996).