miNI: Reducing Network Interface Memory Requirements with Dynamic Handle Lookup

Reza Azimi and Angelos Bilas[1]
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 3G4, Canada
{azimi,bilas}@eecg.toronto.edu

ABSTRACT

Recent work in low-latency, high-bandwidth communication systems has resulted in building user–level Network Interface Controllers (NICs) and communication abstractions that support direct access from the NIC to applications virtual memory to avoid both data copies and operating system intervention. Such mechanisms require the ability to directly manipulate user–level communication buffers for delivering data and achieving protection. To provide such abilities, NICs must maintain appropriate translation data structures. Most user–level NICs manage these data structures statically, which results both in high memory requirements for the NIC and limitations on the total size and number of communication buffers that a NIC can handle.

In this paper, we categorize the types of data structures used by NICs and propose *dynamic handle lookup* as a mechanism to manage such data structures dynamically. We implement our approach in a modern, user–level communication system and evaluate our system, *miNI*, with both micro-benchmarks and real applications. We also study the impact of various cache parameters on system performance. We find that, with appropriate cache tuning, our approach reduces the amount of NIC memory required in our system by a factor of two for the total NIC memory and by more than 80% for the lookup data structures. Moreover, by pinning physical memory automatically and on demand, our approach eliminates the limitations and complexities imposed by static memory pinning that is used in most user–level communication systems. Our approach increases execution time by at most 3% for all but one applications we examine.

[1]Currently, with the Institute of Computer Science, Foundation of Research and Technology – Hellas, P.O. Box 1385, Heraklion, GR 711 10, Greece.

ICS'03, June 23–26, 2003, San Francisco, California, USA.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design Studies

General Terms

Performance Design Measurement

Keywords

System Area Networks, Parallel Architectures

1. INTRODUCTION

Recent work in improving the performance of interconnection networks in scalable servers and storage systems has resulted in new network interface controllers (NICs) that support user–level communication [6, 7, 14]. The main capabilities of these NICs are: (i) to give the ability for user programs to directly access the network in a protected manner for sending and receiving data and (ii) to transfer data directly to- and from virtual memory without OS intervention. These capabilities are now part of industry standards that are in use today or are being proposed for future interconnects [12, 18, 19]. However, modern NICs require certain extensions to provide these capabilities.

First, they require efficient support for translating between virtual and physical memory addresses. NICs usually perform DMA transfers only to and from physical host memory. However, user programs employ virtual addresses to specify communication buffers. Therefore, most user–level communication standards [12, 18] and NICs [14, 7] require efficient address translation.

Second, to allow user programs to directly access the network without OS intervention, NICs must be able to verify user requests. For this purpose, the virtual memory regions used as communication buffers must be registered with the NIC prior to the actual communication operations. Registration implies that the NIC is aware of virtual memory regions that can be used for direct data transfer. Therefore, NICs must maintain information about registered communication buffers [20]. Such systems require a mechanism to authenticate remote programs. This requires NICs to maintain authentication information for point–to–point communication connections. Finally, given that most servers today are required to support multi–programmed workloads, NICs must handle requests of multiple user processes simultaneously. For protection purposes, NICs must directly identify

NIC Memory Size (MBytes)	2	4	8
NIC Price (USD)	995	1,295	1,595

Table 1: Memory configuration and price information for the Myrinet NICs.

local user processes by some handle that is used to retrieve user-specific communication contexts.

Therefore, NICs must lookup at least three different types of resources during their operation: (i) address translation information (ii) registered communication buffers and authentication information, and (iii) processes communication context information. The solution employed in most NICs is to use static lookup tables in NIC memory. The user applications must provide all lookup entries required for the NIC operation prior to any communication operation. This static method for management of NIC memory has two major implications.

First, it requires modern NICs to support large amounts of on-board memory at significant cost. For instance, Myrinet NICs are priced as shown in Table 1 [17]. We see that increasing the amount of memory by 2 MBytes increases the NIC cost by more than 30% whereas increasing memory by 6 MBytes increases the NIC cost by more than 60%. Other products, such as the VI/IP and iSCSI NICs by Emulex [13] employ higher amounts of on-board memory at higher costs. Generally, in many modern clusters the cost of the system area network (including switch costs) that interconnects the nodes of the system is about the same as the cost of the system nodes.

More importantly, all modern NICs impose static limits on the amount of host memory that can be used for communication buffers. In all cases, the user is able to only use a subset of the host memory for communication. This is a severe limitation for application servers in various domains, such as compute servers [20] or database storage servers [26], especially as application needs change over time or in multiprogrammed workloads. The solution that has been employed in these cases is some type of application level management of communication buffers, which is cumbersome, complex, and sometimes not even possible due to OS restrictions. The easier approach is to further increase the amount of NIC memory. However, this only postpones the problem until application working sets increase further and unnecessarily increases the cost of the base system.

In this paper we first categorize the major NIC data structures and functions that contribute to the high memory requirements in modern NICs. We propose a generic scheme, called *dynamic handle lookup* that maintains all important lookup tables in the larger host memory and use parts of NIC memory as a cache. We implement our approach in a modern user–level communication system and evaluate our system *miNI*, with both synthetic micro–benchmark and real applications from the SPLASH-2 [25] suite.

Our approach eliminates the limitations of user–level communication systems on the total size of the communication virtual address space, the total number of communication buffers and connections, and the number of processes that use the system concurrently. In *miNI*, the buffer registration operations are only needed for protection reasons, namely for specifying which memory regions may be used as communication buffers. All pinning and unpinning of physical

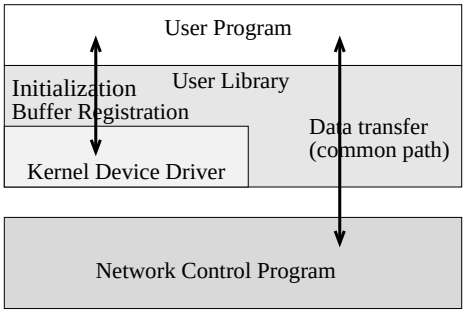

Figure 1: The general architecture of user-level communication systems.

memory happens automatically and on demand. The overhead of using more complex dynamic lookup structures is at most 3% across all but one applications we examine. However, by using dynamic handle lookup, NIC memory size can be reduced substantially by as much as 50% for the total amount of memory and up to 80% for lookup data structures.

In addition to reducing NIC memory requirements, dynamic handle lookup eliminates or relaxes all static constraints on the total size and number of communication buffers that an application can use. Without this approach, NICs must be equipped with resources for the worst case application requirements in each setup, which might result in not using expensive NIC resources most of the time.

The rest of the paper is organized as follows. Section 2 provides the necessary background on user–level communication systems. Section 3 discusses related work. Section 4 describes our design of the dynamic lookup mechanisms. Section 5 presents the results of our performance evaluation and analysis, both with synthetic micro benchmarks and real applications. Finally, Section 6 draws our conclusions.

2. USER–LEVEL COMMUNICATION

User–level communication systems support direct user access to network resources and data transfer operations to local and remote virtual memory without host processor and OS intervention [15, 11, 7]. User–level communication systems use the capabilities of modern NICs to reduce communication overheads. Modern NICs usually employ a communication assist, which can be a special–purpose network processor or a general–purpose processor and usually a few MBytes of static or dynamic memory. NICs also have one or more DMA engines for transferring data between the host memory and the NIC buffers and also between the NIC buffers and the network link. The firmware runs on the communication assist and implements the communication protocol by managing NIC resources, mainly controlling the DMA engines and responding to system events.

Figure 1 shows the general architecture of most user–level communication systems. There are three major components in such an architecture: (i) A kernel–level device driver that is responsible for initializing the NIC and performing trusted functions that cannot be performed directly by the user. (ii) A network control program (NCP) that runs on the NIC and performs all protocol processing tasks. The NCP usually implements a set of state machines that handle all system

events. (iii) A lightweight user–level library to provide the communication API for user programs.

The user–level library in co-operation with the NCP manages per-process, memory-mapped, send, receive, and completion queues on NIC memory. The path in which data is transferred from host memory to NIC memory and then to the network link is called *send path*. The path in which data is transferred from the network link to NIC memory and then to host memory is called *receive path*.

User–level communication systems provide two major categories of communication operations: connection establishment and data transfer.

Connection establishment operations are responsible for setting up a communication channel between the sender and the receiver that is used later on to transfer data back and forth. These operations also initialize the required data structures for memory protection, remote DMA (RDMA) data transfers, and user authentication.

Data transfer operations include both traditional send and receive operations as well as RDMA read and write operations. All memory regions that are involved in RDMA operations need to be *pinned* for the duration of the transfer. That means all pages within the region has to be marked so that the OS excludes them in its attempts to free physical memory.

In general, NIC memory is divided into three regions:

NCP code: This region contains the firmware code that runs on the communication assist and handles all system events. It usually varies between 50-150 KBytes.

Send and receive data buffers: Data that is being received or sent is first transfered to buffers on the NIC. Such buffers are short-lived and usually a small number is adequate for good performance [24]. Furthermore, they are always managed dynamically and their number can be adjusted based on the available NIC memory.

Lookup data structures: This region contains all data structures needed for lookup operations. The most important types of such lookup data structures are used to maintain information for address translation, registered communication buffers, connections, and process communication contexts.

Unlike the NCP code and the data buffers, the size of the lookup data structures grows with the application communication working set size. Given the current trend for increasing application working set sizes, applications that use user-level communication to improve the cost of communication operations require large numbers of communication buffers with a corresponding increase in the size of lookup data structures. For instance, application servers with 32 GBytes of main memory would require approximately 32 MBytes of memory only for virtual address translation, which exceeds by far the capabilities of modern NICs used in system area networks. Furthermore, building NICs with similar capabilities increases the NIC cost significantly. In summary, in cluster configurations where communication among nodes is intense, lookup data structures constitute the largest component of NIC memory.

Another important issue in most user–level communication systems is that pinning host physical memory has to happen statically at the buffer registration time. The buffers physical memory remains pinned until the application explicitly deregisters them. The major problem with this approach is that it limits the total size of the regions of the virtual address space that are designated as communication buffers to a proportion of the physical memory. As a result, the application (or a user–level library linked to the application) would have to break large buffers into smaller buffers and manage their registration and deregistration dynamically. This solution has two drawbacks. First, it complicates the design of an already complex parallel application (or its communication library). Second, if some applications do not do not register and deregister their buffers properly, host physical memory will be wasted.

3. RELATED WORK

An overview of the design and evaluation of dynamic handle lookup for address translation entries has been presented in [1]. In this paper we present the design of the full system. The system design reveals a number of subtle but important issues in generalizing dynamic handle lookup for other on-NIC data structures. We also present a more complete set of results.

The authors in [10] propose a User-managed Translation Look-aside Buffer (UTLB) to dynamically manage translations for virtual memory used as send communication buffers. UTLB is implemented as a per-process two-level custom page table in the host memory. A shared TLB cache resides in NIC memory and caches the most popular entries of the driver-level tables. On a miss the NIC fetches the appropriate table entry using DMA. However, UTLB's dynamic approach is only applicable to the send path of the NIC. Address translation on the receive path is static, which means that all translation entries for registered communication buffers reside in the NIC memory throughout program execution. This results in high memory requirements and limitations on the total size as well as the number of the registered communication buffers. The limitations of this approach have motivated to a large extent our work.

The authors in [21] propose the concept of *virtual networks* to address the issue of supporting large numbers of users over fast, user-level communication systems. The main goal of the work is to efficiently multiplex the system resources among multiple applications with different demands. The main abstraction is the network *endpoint* that consists of the process send and receive queue and the address translation information of the static buffers involved in the communication. Endpoints reside in host memory and are cached on NIC memory. Unlike many user–level communication systems, the virtual networks abstraction does not support RDMA operations to arbitrary memory locations, which eliminates the need for permanent lookup entries in NIC memory. Also, in [8] a scheme similar to dynamic handle lookup is mentioned. However, the authors do not discuss details of the design, nor analyze the performance of the approach.

U-Net/MM [4] is an extension of the U-Net [3] system that also uses endpoints as the main communication facility. Each endpoint is associated with a buffer area that is pinned to contiguous physical memory and holds all buffers used with that endpoint. The U-Net/MM incorporates a TLB structure, in order to handle arbitrary virtual addresses. Similar to our work, U-Net/MM uses an interrupt-based mechanism for handling TLB misses. However, U-Net/MM does not support RDMA operations. Thus, applications can provide the NIC with address translation information upon posting send and receive requests, whereas with RDMA op-

erations, data transfer occurs asynchronously. Moreover, in U-Net/MM, whenever there is an eviction from the TLB, the OS has to be notified to unpin the corresponding pages. In contrast, in *miNI* there is no need to notify the host system in the case of an eviction. Furthermore, U-Net/MM does not deal with the protection issues of communication buffers. Finally, UNet/MM is only evaluated with micro-benchmarks, and therefore, there is little evidence how different communication working set size of the real applications affects its performance.

The Virtual Interface (VI) architecture [12] is an industry standard for user-level communication. VI supports both send and receive operations as well as RDMA operations. Similar to *miNI*, communication buffers must be registered prior to any data transfer. The registration includes both pinning the virtual region and setting up the protection information for it. In current VI implementations [9, 22, 2, 14] all registered regions are statically pinned. This limits the total size of the registered buffer to a fraction of the total physical memory available, unless the user process dynamically manages registration and deregistration of communication buffers. For instance, the cLAN NICs [14] impose 1GByte limit on registered memory, which is not sufficient for modern applications [26]. Another industry standard in system area networks is Infiniband [18]. The core operations and protection model of Infiniband is significantly influenced by VI. More specifically, it follows the same rules as VI in registering the communication buffers with small refinements in the protection model.

Finally, authors in [23] present a survey of different mechanisms used for address translation in NICs. They define four requirements for address translation: (i) Flexibility of use for higher system layers. (ii) The ability to cover all of the user address space. (iii) The ability to take advantage of locality. (iv) Graceful degradation when system limits are exceeded. They evaluate various alternative methods of implementing address translation in NICs by using simulation. They find that hardware lookup structures in the NIC are not required since software schemes are fast enough. They suggest that the NIC should handle all address translation misses which leads to the limitations discussed in this work.

4. DESIGN OF MINI

In this section we describe *miNI* and the design and implementation of the dynamic structures for handling the most important lookup operations in the NIC. First, we define some important terms that are used frequently in the description of our design.

Virtual Memory Handle specifies the virtual address in host memory of the source or destination location for data transfer operations. VMHs are used for virtual to physical address translation in the send and receive path of the NIC. In our implementation the VMH is 48 bits, composed from a virtual address and the process Id.

Communication Buffer Handle specifies the communication buffer used in data transfer operations. CBHs are used by the receiving NIC to obtain protection information about a buffer and by the sending NIC to specify a communication buffer in the cluster. In our current implementation the CBH is 48 bits, composed from the buffer Id and the process Id.

Process Communication Handle (PCH) specifies a process in a communication operation in multiprogrammed workloads. In our implementation it is a 16-bit integer.

4.1 Basic Mechanisms

The main idea in *miNI* is to move all lookup tables to the much larger host memory and use NIC memory as a cache. On a cache miss, the NCP issues an interrupt to the host CPU that is handled by the NIC device driver. The handler fetches the missed entries from lookup tables in host memory and places them in the cache in NIC memory. The NIC cache is shared among all processes that use the communication system. This allows for better management of the NIC memory. However, the performance of the communication system might substantially degrade if many processes compete for the same cache locations. The two major problems in such a design are: (i) How to fetch the missed entries from host memory and (ii) how should the NCP handle requests that miss. Next, we describe the details of our solutions to these problems.

4.1.1 Fetching missed entries

There are two ways for the NCP to request entries that miss: To interrupt the host CPU or to DMA the entries directly from the host memory. The latter is faster and does not impose any overhead on the host CPU. However, it requires that the host CPU prepares all required entries before the execution of the communication operation that needs the entries. This requires a priori knowledge of what operations are going to take place, which is difficult to obtain for long RDMA operations. Moreover, with this method the NCP must be aware of the physical structure of the lookup tables in host memory. This limits the design of the lookup tables, since complex remote lookup procedures from the NCP might become expensive. In contrast, the interrupt–based method allows the device driver to prepare the required entries on demand. Moreover, it has the advantage that the device driver may organize the lookup tables arbitrarily.

Handling cache miss interrupts can be done in two ways: Using an interrupt handler or using a separate thread running in the kernel. The former is faster since it incurs no context switch overhead and unpredictable scheduling delays. However, it has two shortcomings. First, certain functions in the standard kernel API for device drivers cannot be called in the context of an interrupt handler. Thus, this approach may require implementing custom internal kernel functions to provide the functionality, which is not possible in proprietary OS's, and reduces system portability in general. Moreover, interrupt handlers cannot be blocked in many OS's, limiting the kernel functions that can be called in the interrupt handler context.

In this work we choose to handle lookup cache miss interrupts in a separate thread running in the kernel. Figure 2 shows the cache miss handling path. The NIC issues a miss interrupt when a lookup in one of the cache data structures misses. The interrupt handler in the kernel device driver just prepares a *miss event descriptor* with the information given by the NIC and places it in the miss event queue. It then wakes up the in-kernel *miss event thread*, which does all processing and updates the on-NIC cache.

4.1.2 The NCP behavior in a cache miss

When a cache miss happens during processing a request, the NCP blocks processing the request and processes other active connections. Meanwhile, the NCP has two ways to

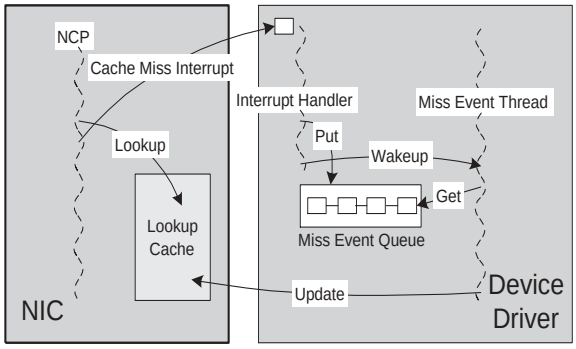

Figure 2: Cache miss handling path.

handle the blocked request: To buffer the request in NIC memory until the missed entries are fetched from the lookup tables in the cache or to drop the request and rely on the underlying transport protocol to retransmit the unacknowledged request. Buffering requests requires a potentially high amount of NIC memory, since the miss handling delay from host memory may be long. Moreover, it complicates the design of the NCP. On the other hand, dropping a request is simple and requires no NIC memory. However, it incurs the retransmission overhead. For these reasons, we use the second approach of dropping requests that miss.

In summary, in *miNI* we favor design choices that lead to a simpler and more portable implementation, despite the fact that this may increase miss penalties. Performance-wise, as explained in Section 5, our evaluation confirms our intuition that we can limit the overall performance overhead by reducing the miss rate with proper cache configuration and tuning.

4.2 VMH Lookup

VMH lookup is required both in the send and the receive paths. Also, RDMA read and write operations are handled differently. For RDMA reads, in the send path data is sent as the result of a remote read request. In this case the RDMA read request contains the CBH of the communication buffer and the offset of the data to be read within the buffer. The CBH is used to lookup the virtual address of the start of the buffer. This address is added to the offset to form the VMH of the area from which the data must be sent. In the receive path, NIC processing is simpler since the VMH of the destination of the data on the host memory has already been posted at the time of issuing the RDMA operation. Conversely, for RDMA writes, in the receive path data has to be written into a buffer as a result of an RDMA write operation, issued from a remote node. In this case, the RDMA request contains the CBH of the communication buffer and the offset within the buffer. The CBH is used to lookup the starting virtual address of the communication buffer. This address is added to the offset to form the VMH of the area to which the data will be written. For RDMA writes the send path is simpler since data is sent as a result of an RDMA write request issued at the local node and the VMH of the source data is in the request descriptor posted by the user process.

There are two data structures that hold the address translation information used during VMH lookup: (i) the *VMH Lookup Table* that resides in host memory and (ii) the *VMH Lookup Cache* that is located in NIC memory and acts as a cache for the VMH table.

4.2.1 VMH Lookup Table

For each process, there is one VMH table in the device driver that keeps the virtual–to–physical address mappings for all pinned pages of the process. When there is a miss in the VMH cache, the device driver looks for the missed entries in the VMH table. If they are found, the driver just updates the VMH cache. Otherwise, it pins all the pages in the missed address range and updates both the VMH table and the VMH cache with the physical addresses.

We implement the VMH table structure as a two-level page table. However, only the pinned addresses have valid entries in the VMH table. Although possible, we do not use system page tables for portability and performance reasons. Since there is a limit on the total number of pinned pages in the system, we need to manage the number of pinned pages per process. When the total number of pinned pages in a VMH table reaches a high water mark, a replacement algorithm is activated to unpin a set of virtual pages and evict them from the VMH table, until the total number of pages in the VMH table becomes equal to a low water mark. The actual values for the high water mark and low water mark depend on the application memory usage pattern and can be tuned dynamically.

The eviction must be synchronized with the VMH cache to avoid unpinning an in–use virtual page. In our implementation we conservatively assume all entries that are in the VMH cache as potentially in–use and avoid evicting them from the VMH table. This is both efficient and easy to implement. However, it is suitable only for systems where the size of the VMH table is significantly larger than the VMH cache, so that the there are sufficient out–of–cache candidates for unpinning.

4.2.2 VMH Lookup Cache

The send and receive paths of the NIC use the same cache for uniformity and simplicity. We use a set-associative cache structure with configurable associativity, cache line size, and cache size. Figure 3 shows the structure of the VMH cache. We use the lower bits of the virtual address in the VMH as the set index and the rest of the virtual address bits and the process Id as the tag information. This prevents static partitioning of the cache among processes. Also, it reduces the chance of conflicts among the addresses close to each other for which we expect spatial locality. The VMH cache is shared among all processes.

The VMH cache line size can be configured at compile time. Multiple–entry cache lines allow for a smaller tag-array. For each entry in a cache line, the full 32-bit word is allocated for the physical page address, allowing up to 16 TBytes of physical memory to be used for communication buffers. The cache line is also the unit of data transfer between the VMH table and cache. Therefore, when there is a miss for a virtual address in the cache, the missed virtual address will be aligned to the cache line boundaries. This may result in effective prefetching for applications with sufficient spatial locality in their communication buffer access pattern. However, this also increases the cache miss penalty, mostly due to larger pinning costs.

The VMH cache can be configured with different associativity levels. Although with higher associativities the

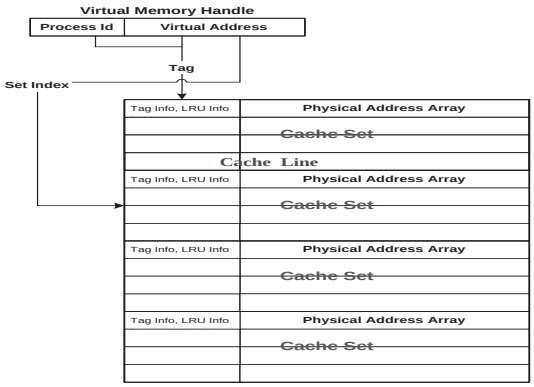

Figure 3: Structure of the VMH lookup cache.

number of conflicts may be reduced, the cost of lookup in the cache tag array increases, which is specially an issue in caches implemented in software.

The VMH cache uses an LRU replacement policy for the entries of each set. The cache replacement is implemented in the NIC. Since the NCP access to the VMH cache is read–only, the replacement does not include any write–back operation and the entries to be replaced are simply thrown away. When an entry is replaced in the VMH cache, it remains in the host VMH table. Entries in the VMH table are pinned as well. However, as mentioned above, once a line is evicted from the VMH cache, it also becomes a candidate for eviction from the VMH table, in case it reaches the high water mark. However, VMH table replacement happens periodically with long intervals. If a page is accessed frequently, it will be brought back to the VMH cache and thus, will not be evicted from the VMH table.

While the data is being transfered from or to a virtual page in a cache line, the whole cache line is locked in the VMH cache, preventing any line entry from being replaced. For this purpose, we assign a *lock* bit for each cache line. We set the lock bit when there is an on–going DMA from a page of the line and reset the bit once the DMA is complete. Since different areas in a cache line can be used for DMA in the send and receive paths simultaneously, we duplicate the lock bit for the send and receive paths. We assume there is only one DMA engine active in each path (send and receive) at a time, therefore, two lock bits are sufficient.

Since the NCP implements a complex state machine, it is not easy to deal with all deadlock and live-lock scenarios that can happen due to concurrent lock and wait-for-lock operations on each VMH cache entry between the send and receive paths. For this reason and to simplify the NCP design, we have implemented an atomic *lock-use-unlock* scheme between the send and receive paths that assumes a level of associativity of at least two. This allows us to also relax contention over a single cache line between the send and receive paths.

4.3 CBH Lookup

Similarly, the CBH lookup uses a lookup table located in host memory and a lookup cache located in NIC memory.

4.3.1 CBH Lookup Table

There is one CBH lookup table per process, located in the device driver memory. The buffer Id (which is part of the CBH) of each incoming packet is used as a key to lookup the appropriate communication buffer descriptor. Since in our system there is no restriction on the value of the buffer Id, we implement the CBH lookup table as a hash table. The communication buffer descriptors in the CBH lookup table contain a superset of the fields available in the CBH lookup cache with the extra descriptor fields used only by the device driver.

User programs call the device driver to register a communication buffer. The driver allocates a descriptor for the new buffer, initializes it, generates a protection key, and inserts the newly created descriptor to the CBH lookup table. We use the lower bits of the buffer Id as the hash index. However, more sophisticated hash functions might be required for applications with thousands of communication buffers to distribute the buffers into the hash bins more evenly. The descriptors in the CBH lookup table are permanent until the user process de-registers a specific buffer, in which case *miNI* invalidates both the descriptors in the CBH lookup table and the CBH lookup cache.

4.3.2 CBH Lookup Cache

The CBH lookup cache contains a subset of the communication buffer descriptors in the CBH lookup table and resides in NIC memory. Each descriptor in the cache includes the virtual address and the length of the buffer and the buffer protection key. The lower bits of the buffer Id in the CBH are used as the cache index, whereas the rest of the bits in the buffer and process Ids are used as the tag.

The CBH lookup cache is set–associative and the level of associativity can be configured at compile–time. The replacement within a cache set is done by the NIC. Since the descriptors in the cache are read–only for the NIC, the replacement does not include any write–back operation. Unlike the VMH cache, the CBH lookup cache line size is always one, since we do not expect any locality of access in terms of CBH.

When the NCP receives a packet from a remote node, it extracts the destination buffer and process Id and checks the CBH lookup cache for a matching entry. On a cache hit, the NCP will proceed with handling the incoming request. On a cache miss, the NCP reserves an entry in the CBH lookup cache and drops the incoming request. Then, it raises an interrupt and notifies the device driver for a *CBH lookup cache miss* event. The event handler in the driver searches for the requested descriptor in the CBH lookup table of the corresponding process. If the buffer request is valid, the driver allocates a CBH lookup cache entry and inserts the buffer descriptor. If the buffer request is invalid, the driver marks the CBH lookup cache entry with an *invalid* flag. When the network request is retransmitted through the NCP retransmission mechanism, the NCP will respond to the request with an error code.

4.3.3 Imported CBH Lookup

As mentioned in Section 1, the imported CBH lookup table is used for authentication of user requests in the send path. For each connection to a remote buffer there is a connection descriptor that stores for each communication buffer, its CBH, length, Id of the remote node, and the buffer protection keys. Since a process might have hundreds or thousands of active connections, the total NIC memory required for maintaining connection information in

a multiprogramming environment might grow up to several MBytes. Our approach to dealing with these requirements is to slightly change the system protection model and to move all information about remote buffers to the user-level library. All RDMA read and write requests posted by the user to the NIC must therefore contain information about the remote communication buffer. The implication of this modification is that the NIC at the requesting node is not able to verify if user requests are valid. However, this check is still performed on the receive side.

To verify incoming requests, we use a capability mechanism based on keys. In our scheme, a random key is generated for every communication buffer at registration time. The key is a large enough number (64 or 128 bits) so that it is hard to guess. During connection establishment, the key is sent to the remote node that uses it in every subsequent communication operation. Keys are invalidated when the owner decides to deregister the communication buffer.

Although this modification changes the original protection model, the practical implications are minor and the advantages outweigh the shortcomings. As a result, a user can maliciously flood the interconnect with invalid requests that will be dropped on the receive side. However, this is not an issue for the systems under consideration. On the other side, removing the connection descriptors from NIC memory results in reducing memory requirements, and most importantly simplifying the on-NIC translation code, which is a significant consideration for all systems at this level of complexity. In general, we believe that in most system designs, performing access checks either only on the receive or only on the send side is adequate. In our implementation we perform all protection checks in the node that owns each communication buffer.

4.4 PCH Lookup

The private post queues for each user process may increase memory requirements significantly in multiprogrammed environments. In our work we propose maintaining a small number of post queues on the NIC and sharing them among multiple users by mapping them to their virtual address spaces on demand. This is possible, since the post queues are used synchronously to program execution only when a process explicitly posts a data transfer request. At these points, the post operation results in a page fault if the post queue is not mapped and the system takes control to perform the necessary mapping actions. For the numbers of users that need to be supported on such systems, the rest of the user context information can be maintained on the NIC.

In servers that provide services to multiprogrammed workloads, the common case involves a small number of user processes. Since most NICs today can easily support at least a few –ten or so– post queues, in our evaluation we allow each user process to have its own post queue, and we do not evaluate this mechanism any further.

5. RESULTS

The goal of our evaluation is twofold: (i) we would like to tune the lookup cache parameters in our extensions so that we both reduce the memory requirements and impose little additional overhead to real applications, and (ii) we want to gain insight on how the system behaves as the parameters vary within a wide range of values and applications.

As the base for implementing our approach, we use Virtual Memory Mapped Communication (VMMC) [11]. VMMC provides protected, user–level communication between the sender's and the receiver's virtual address spaces. VMMC guarantees FIFO message delivery between any two processes in the system and tolerates transient network errors by using packet retransmission. More specifically, VMMC includes a reliability mechanism [24] that retransmits packets until they are acknowledged by the peer node.

Processors	2 x Intel Pentium III, 800 MHz
Cache	32K (L1), 512K (L2)
Memory	512MB SDRAM
OS	RedHat Linux Kernel 2.2.16-3smp
PCI buses	32 bits, 33 MHz, 133MBytes/s
NIC	Myricom M3M-PCI64B
NIC CPU	LANai9, 133 MHz
Network Link	Bi-directional, 160MBytes/s/direction

Table 2: Cluster node configuration.

The miss penalty depends on the retransmission timer interval used in the NCP [24]. The retransmission interval is the time that the requesting node waits for acknowledgments for transmitted packets before retransmitting each packet. Our experiments show that a retransmission interval of 200-300μs results in optimal system behavior. Furthermore, we verify that each request that misses involves on average about one retransmissions. Setting the retransmission timer to a lower value interferes with normal system behavior [24].

The experimental system we use for evaluation is a cluster of four 2-way PentiumIII nodes. The exact configuration of each node is shown in Table 2. Nodes in the cluster are interconnected with a Myrinet network [7]. All system nodes are connected with a 16-port, full crossbar, Myrinet switch. The PCI–based NIC is composed of a 32–bit, general-purpose processor (LANai9) with 2 MBytes of SRAM.

To evaluate the impact of our approach on the system performance, we use both synthetic micro-benchmarks as well as real applications. We use micro-benchmarks mainly to provide basic measurements and to independently stress specific components of the system.

The applications we use are a subset of the SPLASH-2 suite [25] on top of a shared virtual memory (SVM) system that provides the illusion of a single system image. The specific applications we use are FFT, WaterNsquared, WaterSpatial, LUContiguous, Ocean, Volrend, Radix, and Raytrace. This set of applications covers a wide range of communication patterns. The SVM protocol we use is GeNIMA [5], a home-based, page-level protocol. GeNIMA has been optimized to be used with system area networks that support remote RDMA operations. For the SPLASH-2 applications, we use the largest problem set size permitted by the 2-GByte address space of the Linux OS. Using larger cluster configurations would result in less stress for the NIC memory subsystem, since the application working set is divided among more nodes and therefore, the communication working set of each node would be smaller.

We present results both for the VMH and the CBH lookup mechanisms. For space reasons, we focus more on the VMH lookup, since it is more critical to system performance. We present statistics on both miss rates and application speedups. We divide cache misses based on two factors: (i) the path in which they occur (send or receive) and (ii) the miss type,

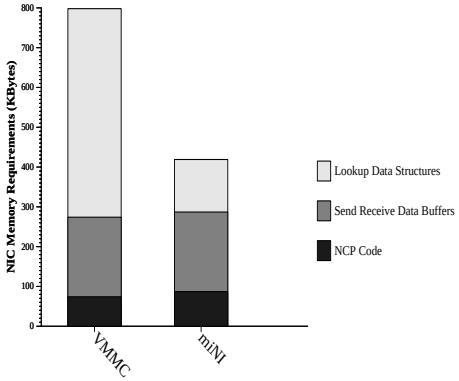

Figure 4: Breakdown of NIC memory allocation in VMMC and *miNI*.

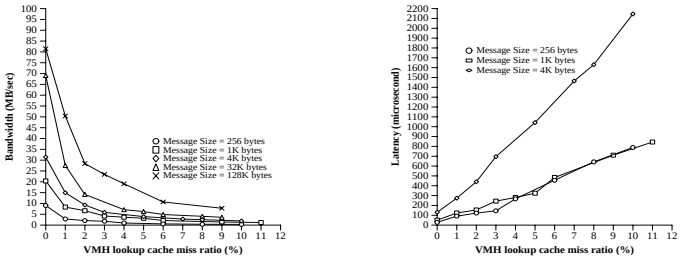

Figure 5: Effect of VMH miss on ping-pong latency and bandwidth.

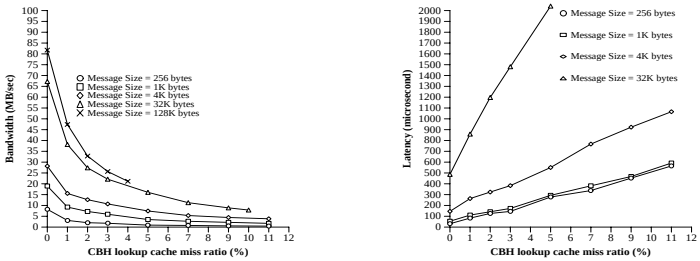

Figure 6: Effect of CBH miss on ping-pong latency and bandwidth.

i.e. cold misses versus capacity or conflict misses (non-cold), [16]. The first factor provides information about the communication working sets in either paths for each application whereas the second one helps us understand better the impact of each cache parameter.

5.1 NIC Memory Requirements

By using dynamic handle lookup in our system we are able to reduce the total NIC memory requirements from 800 KBytes to 400 KBytes. In particular, the memory used for all lookup structures is reduced by about 80% from 520 KBytes to 130 KBytes. Figure 4 summarizes the breakdown of the memory consumption for code and the data structures on the NIC memory. Moreover, *miNI* does not have any of the constraints on the total size and number of communication buffers and the total number of connections. Finally, registration operations are only needed for protection purposes, namely for specifying the boundaries of the communication buffers. All memory pinning and unpinning operations happen dynamically and on demand.

Memory saving on larger system configurations can be even more significant. A system that uses 32 GBytes of host memory for application communication buffers, as is common in data-base servers [26], would require about 32 MBytes of NIC memory for VMH translation, which exceeds the capabilities of most NICs, even excluding CBH translation. Extrapolating from our results, our approach would require at most 8 MBytes of memory (including all NIC data structures), which is within the capabilities of existing NICs. Moreover, our approach is able to use all 32 GBytes of host memory for communication, even with smaller NIC memories, although, potentially at a higher performance cost.

5.2 Micro-benchmarks

Figure 5 shows how latency and bandwidth vary as we change the VMH cache miss ratio. The micro-benchmark we use is essentially a ping-pong test, where requests miss on one of the two nodes only. The micro-benchmark is able to control the miss rate by using specific source and destination buffers for communication. We see that the bandwidth degrades rapidly and the latency increases linearly as the miss ratio increases. In both cases the negative effect of a miss is higher for larger message sizes. This is due to the fact that the cost of dropping and retransmitting on a miss increases with message size due to the lack of negative acknowledgments and receive side buffering in our system.

Similarly, we use a micro-benchmark to examine the impact of the miss ratio on basic ping-pong latency and bandwidth. In order to isolate the impact of CBH cache misses, we configure the VMH cache to incur a negligible number of misses. Figure 6 shows that the effect of the CBH cache misses both on bandwidth and latency is very similar to that of the VMH cache. For larger message sizes the penalty of dropping packets and retransmission is higher, which results in higher performance penalty per cache miss. However, a small CBH cache is adequate to eliminate most cache misses, alleviating the effort of the high miss penalty. More importantly, by implementing dynamic CBH management we are able to reduce the amount of memory required on the NIC to support the problem sizes we examine from 163 KBytes to 25 KBytes without any noticeable impact on performance.

5.3 Real Applications

Due to the importance of VMH translation for the common communication path we mostly focus our evaluation on the VMH cache. For each application we choose a fairly large problem size, as shown in Table 3 to stress the VMH lookup cache.

We vary each of the three VMH lookup cache design parameters, cache size (**C**), line size (**L**), and associativity (**A**) within a wide range, as shown in Table 4. We first examine the impact of each cache parameter on the cache miss ratio. Figure 7 shows the VMH lookup cache miss ratio breakdown for each set of parameters. The first observation is that the number of misses varies greatly, between about 0-40% among different configurations. Overall, the miss ratio of WaterNsquared, WaterSpatial, Ocean, Raytrace, and

Application	Input Size
WaterSpatial	4096 Mols
WaterNsquared	4096 Mols
Raytrace	1024x1024 car
Radix	8M Keys
Ocean	514x514
LUContiguous	2048x2048
Volrend	CST head
FFT	4M Complex Numbers

Table 3: Problem sizes for SPLASH-2 benchmark applications.

Parameter	Values
Cache Size **(C)**	8K, 16K, 32K entries
Cache Line **(L)**	8, 16, 64, 128 entries
Associativity **(A)**	2, 4, 8 lines

Table 4: Range of values for each of the VMH lookup cache parameters.

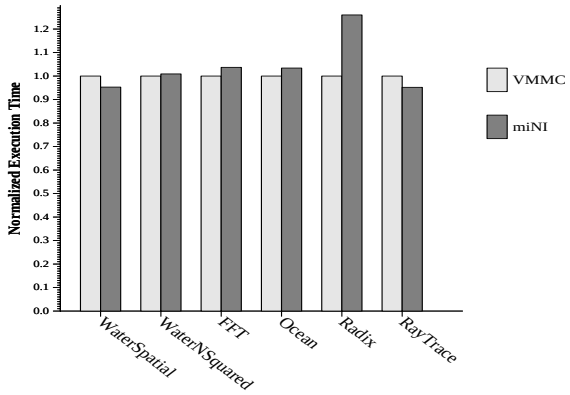

Figure 9: Normalized execution time of each application in the original VMMC system versus *miNI* for the best VMH and CBH cache configurations.

Volrend is within the range 0-2% range, whereas for FFT, LUContiguous, and Radix the miss ratio is much larger. Second, we observe that all parameters have a significant effect on system miss rates, as explained next.

Increasing the cache line size results in very effective prefetching and reduces the number of misses for most applications even when the cache line size is increased up to 128 entries. However, when increasing the cache line size from 16 to 64 and then to 128 entries, Ocean exhibits a large number of conflicts.

Cache size has a small effect on the number of misses at small line sizes, but the importance of the cache size increases for larger line sizes. Doubling the cache size from 16 to 32K entries more than halves the total number of misses for FFT and LUContiguous in the L128 configurations (Figure 7).

Finally, cache associativity is important mostly at smaller cache and line sizes. Increasing associativity from 2 to 4 has a positive effect on most applications. Increasing the associativity from 4 to 8 seems to have a smaller effect. Furthermore, since the VMH lookup cache is implemented in software, higher associativities result in higher lookup times (e.g. for Radix C8.L128, Ocean C32.L128, and WaterSpatial C8.L64, and C8.L128 Figure 7). For these reasons, an associativity of 4 seems to be the best compromise between reducing the number of misses and increasing lookup overhead.

We also look at the impact of cache configuration on parallel speedups, as shown in Figure 8. There is a close correlation between the VMH lookup cache miss rates and parallel speedup, even for configurations with small miss ratio. Generally we can divide applications in two categories. In the first category belong WaterNsquared, WaterSpatial, Volrend, Raytrace, and Ocean that, except for few configurations, are not very sensitive to the cache configuration due to the small miss rates they exhibit. Overall, for these applications, a small VMH lookup cache of 8K entries results in similar performance to the static system configuration, where the full VMH table is maintained on NIC memory. The second category includes FFT, Radix, and LUContiguous that show significant variations in speedups. For these applications the cache line size is the most important parameter, resulting in more than 100% in speedup variation.

Figure 9 shows the normalized execution time of each application in the original VMMC system versus their best performance in *miNI*. We observe that except for Radix, the overhead of *miNI* is negligible. For some applications the new system performs slightly better. We have not investigated this effect completely yet, but we believe that the changes in the common path due to the imported CBH table elimination result in somewhat lower system overheads for cases that do not exhibit many VMH and CBH misses.

We observe that a configuration of (C16, L64, A4) results in the best tradeoff between performance and NIC memory requirements. This configuration uses about 75 KBytes of NIC memory that is a substantial reduction from the original configuration, which uses more than 256 KBytes for the full VMH lookup table. Moreover, the current configuration does not have a limit on the amount of host memory it can support, although performance tradeoffs may change as application working set sizes increase.

6. CONCLUSIONS

User–level NICs need to translate between virtual and physical addresses as well as between other types of user and system handles. In current user–level communication systems the mechanisms used for address translation are static. With increasing host memory sizes, this approach requires large on-NIC memory to maintain various mappings and, more importantly, it imposes limitations on the amount of host memory that can be used for communication buffers.

In this work we deal with these shortcomings by extending a user–level communication system to use the NIC memory only as a cache for the larger host memory. We implement our approach in a commodity, programmable NIC (Myrinet) and measure its impact on system performance with both micro-benchmarks and real applications.

Overall, dynamic handle lookup results in significant reductions in memory requirements with a small impact on system performance for the configurations we examine. Total NIC memory requirements are reduced by about 50%, whereas memory requirements for lookup operations are reduced by as much as 80% in our prototype. The overhead of using more complex dynamic lookup structures relative to the static data structures is at most 3% across all but

one applications we examine. In larger system configurations, memory savings can be even more significant. Our extensions eliminate any NIC-imposed restrictions on host memory can be used for communication buffers, an important problem in scalable storage, database, and application servers.

We find that the communication working sets of many realistic applications are in many cases small and that small on-NIC lookup caches have almost no impact on performance. In most cases the cache configuration parameters have a large impact both on the number of cache misses and the overall system performance. We find that there is considerable spatial locality in the communication access pattern of most applications and prefetching with large cache lines is effective and overshadows the increased miss penalty costs, due to the larger cache line size. Finally, in our system design we avoid unnecessary complexity that results in more robust systems, specially in more aggressive hardware implementations.

7. ACKNOWLEDGMENTS

We would like to thank the members of the ATHLOS project at the University of Toronto for the useful discussions during the course of this work. We thankfully acknowledge the support of Natural Sciences and Engineering Research Council of Canada, Canada Foundation for Innovation, Ontario Innovation Trust, the Nortel Institute of Technology, Communications and Information Technology Ontario, and Nortel Networks.

8. REFERENCES

[1] R. Azimi and A. Bilas. Evaluating the performance impact of dynamic handle lookup in modern network interfaces. In *Proc. of the 2nd Annual Workshop on Novel Uses of System Area Networks SAN-2, Anaheim, California*, 2003.

[2] M. Banikazemi, B. Abali, and D. Panda. Comparison and evaluation of design choices for implementing the virtual interface architecture (via). In *Workshop on Communication and Architectural Support for Netwok-based Parallel Computing CAPCN-HPCA*, 2000.

[3] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. *Proc. of the Fifteenth Symposium on Operating Systems Principles (SOSP15)*, December 1995.

[4] A. Basu, M. Welsh, and T. von Eicken. Incorporating memory management into user-level network interfaces. http://www2.cs.cornell.edu/U-Net/papers/unetmm.pdf, 1996.

[5] A. Bilas, C. Liao, and J. P. Singh. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proc. of the 26th International Symposium on Computer Architecture (ISCA26)*, May 1999.

[6] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proc. of the 21st International Symposium on Computer Architecture (ISCA21)*, pages 142–153, Apr. 1994.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[8] P. Buonadonna, J. Coates, S. Low, and D. Culler. Millennium sort: A cluster-based application for windows nt using dcom, river primitives and the virtual interface architecture. In *Proc. of the 3rd USENIX Windows NT Symposium, Seattle, WA*, July 1999.

[9] P. Buonadonnaa, A. Geweke, and D. Culler. An implementation and alaysis of the virtual interface architeture. In *Supercomputing (SC)*, pages 7–13, 1998.

[10] Y. Chen, A. Bilas, S. N. Damianakis, C. Dubnicki, and K. Li. UTLB: A mechanism for address translation on network interfaces. In *Proc. of The 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS8)*, pages 193–203, San Jose, CA, Oct. 1998.

[11] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proc. of The 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V)*, Aug. 1997. A short version of this appears in IEEE Micro, Jan/Feb, 1998.

[12] D. Dunning and G. Regnier. The Virtual Interface Architecture. In *Proc. of The 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V)*, Aug. 1997.

[13] Emulex. Gn9000/vi: 1gb/s vi/ip pci host bus adapter. http://www.emulex.com/products/viip/index.html.

[14] Giganet. Giganet cLAN family of products. http://www.emulex.com/products.html, 2001.

[15] R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proc. of the IEEE Spring COMPCON '96*, Feb. 1996.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[17] M. Inc. Myrinet product list and prices. http://www.myri.com/myrinet/product_list.html, 2002.

[18] InfiniBand Trade Association. Infiniband architecture specification, ver. 1.0. http://www.infinibandta.org, Oct. 2000.

[19] Internet Engineering Task Force (IETF). iSCSI, version 08. In *IP Storage (IPS), Internet Draft, Document: draft-ietf-ips-iscsi-08.txt*, Sept. 2001.

[20] P. Jamieson and A. Bilas. Cables: Thread control and memory management extensions for shared virtual memory clusters. In *Proc. of The 8th IEEE Symposium on High-Performance Computer Architecture (HPCA8)*, Feb. 2002.

[21] A. M. Mainwaring and D. E. Culler. Design challenges of virtual networks: Fast, general-purpose communication. In *Proc. of The 1999 ACM Symposium on Principles and Practice of Parallel Programming (PPoPP99)*, pages 119–130, May 1999.

[22] National Energy Research Scientific Computing Center. M–via: A high performance modular via for linux. http://www.nersc.gov/research/FTG/via, 1998.

[23] I. Schoinas and M. D. Hill. Address translation mechanisms in network interfaces. In *Proc. of The 4th IEEE Symposium on High-Performance Computer Architecture (HPCA4)*, 1998.

[24] J. Tang and A. Bilas. Tolerating network failures in system area networks. In *Proc. of the 2002 International Conference on Parallel Processing (ICPP02)*, Aug. 2002.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd International Symposium on Computer Architecture (ISCA22)*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[26] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. Philbin, and K. Li. Experiences with vi communication for database storage. In *Proc. of the 29th International Symposium on Computer Architecture (ISCA29)*, May 2002.

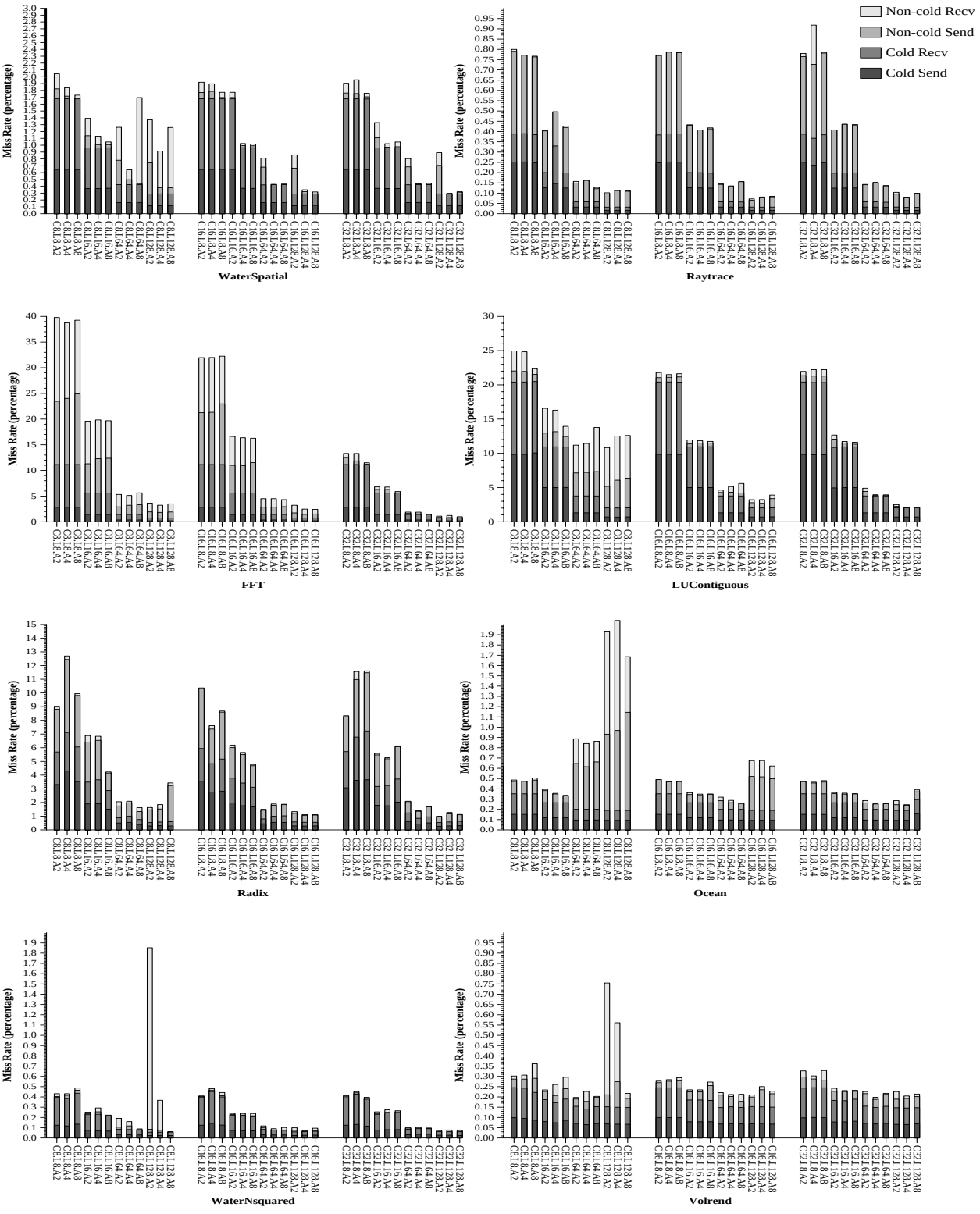

Figure 7: VMH lookup cache miss breakdown of the SPLASH-2 applications.

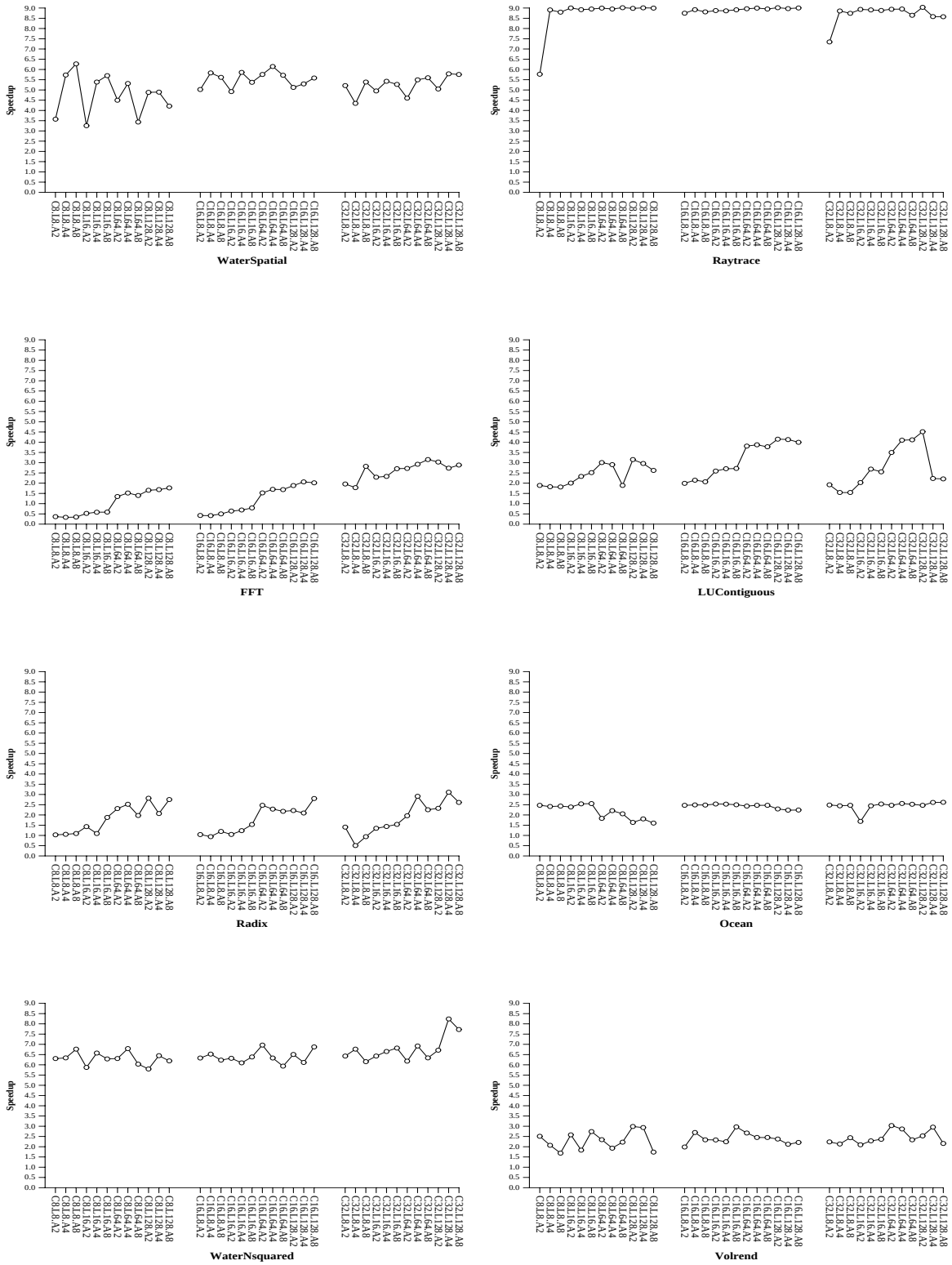

Figure 8: Parallel speedup of the SPLASH-2 applications in different VMH lookup cache configurations (8 processors used).