UTLB: A Mechanism for Address Translation On Network Interfaces

Princeton University Technical Report TR-580-98

Yuqun Chen, Angelos Bilas, Stefanos Damianakis, Cezary Dubnicki, and Kai Li
Computer Science Department, Princeton University, Princeton, NJ-08544
{yuqun, bilas, snd, dubnicki, li}@cs.princeton.edu

Abstract

An important aspect of a high-speed network system is the ability to transfer data directly between the network interface and the application buffers. Such a *direct data path* requires the network interface to "know" the virtual-to-physical address translation of a user buffer, *i.e.*, the physical memory location of the buffer. This paper presents an efficient address translation architecture, User-managed TLB (UTLB), which eliminates system calls and device interrupts from the common path of communication. With the UTLB architecture, the network interface can translate a virtual page address in 0.5 μs in the best case. The average overhead of an address translation is less than 2 μs under the heavy load of multi-programming on a 4-way SMP. The paper describes the design of the UTLB and its implementation as part of a custom communication layer. Results from micro-measurements and trace-driven analysis are given to evaluate the effectiveness of the UTLB architecture.

1 Introduction

A parallel computer or computer cluster can exploit the aggregate computing power and I/O throughput of many microprocessors and I/O buses. Such a system usually consists of multiple processing elements connected via a high-speed network. The processing element (or the *host*) has one or more microprocessors. A network interface connects the processing element to the network. Together with a software layer (the communication subsystem), the network interface packetizes and transmits the raw data from the application process to the network. The communication subsystem at the other end (the receiver side) re-assembles the packets from the network and deposits the data into the buffers specified by the receiver process.

The data path between the network interface and the user buffer can be either indirect or direct. The *indirect* path involves copying data between the user buffer and the dedicated system buffer. This path is straightforward to implement but incurs significant overhead due to data copying.[1] In contrast, the *direct* data path allows data transfers between the network interface and the user buffers. It is usually accomplished by DMA. The direct path not only eliminates the copy overhead, but also reduces the amount of involvement by the host processor.

In order to use the direct path, the network interface must know the physical memory location of the user's virtual memory buffer. Computer systems operate in either of two modes: *user mode* and *kernel mode*. The mappings between virtual memory addresses and the physical memory addresses, or virtual-to-physical translations, are maintained inside the operating system (OS) and are inaccessible to the applications processes running in user mode. But a user process can invoke the OS to initiate data transfers in kernel mode. The OS looks up the address translation and issues the request to the network interface using physical addresses. This approach requires a system call when sending data to the network, and typically a device interrupt when receiving data from the network. System calls and device interrupts involve context switching whose overhead ranges from 10 μs to 100 μs even on the fastest microprocessor. As the speed of networking hardware increases, this overhead becomes significant compared to the raw communication time of a medium-sized message (a few hundred bytes) in the hardware.

The User-managed TLB (UTLB) is an address-translation architecture for network interfaces that use the direct path to transfer data. By exploiting spatial locality in most applications, UTLB eliminates system calls and device interrupts from the common case. The UTLB only invokes the operating system for *pinning* the user buffer when it is first used in communication. The unique translation table in the UTLB allows the user process to specify a data buffer with virtual addresses,

[1] With Programmed I/O (PIO), the process can directly write to the buffer on the network interface. PIO requires heavy involvement from the host processor. Therefore, it is typically used to send small messages.

each of which the network interface can *safely* translate into a physical address with just a table lookup. The network interface lookup costs 0.5 μs in the best case and less than 2 μs on average.

The UTLB architecture requires only a page-pinning routine which is available on almost all operating systems. By not relying on OS modifications nor on esoteric OS features, the UTLB architecture is therefore portable across a wide variety of OS platforms and network interfaces.

In the following sections, we describe the design and implementation of UTLB in detail. We first illustrate the address translation problem in the context of a direct memory-to-memory communication layer. We then describe the ideas of UTLB and its implementation. Finally, we present the results from micro-measurements and trace-driven analysis.

2 Address Translation in VMMC

In this section, we introduce the VMMC communication model and illustrate the address translation problem that VMMC faces.

2.1 The VMMC Model

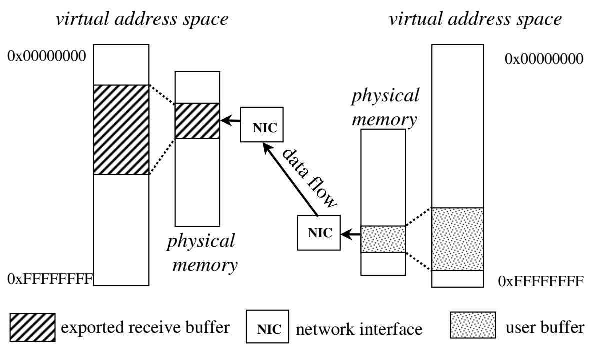


Figure 1: The VMMC Communication Model

Virtual Memory-Mapped Communication (VMMC) is a communication model that provides direct data transfer between the virtual address spaces of two processes. Two kinds of user buffers are used for data transfer: the *receive buffer*[2] and the *local buffer*. Both buffers reside in the virtual address space of a user process. The receive buffer is made visible to processes on remote hosts through an *export* system call. A process can gain access rights to an exported receive buffer by *importing* it. The basic VMMC model supports *remote store*, which allows a process to send data from a local buffer directly into a another process' receive buffer over a network (Figure 1).

[2]The name *receive buffer* is used for historical reasons and does not reflect how it is used in the extended VMMC model.

VMMC was initially implemented on the SHRIMP multi-computer [3] and ported to Myrinet network interface [9]. We later extended the VMMC model on the Myrinet network interface with two new features: *remote fetch* and *transfer redirection* [8].

- Remote-fetch allows a process to *fetch* data from a receive buffer into its local buffer.
- Transfer-redirection "redirects" the incoming data that would otherwise land on a contiguous subregion (*redirection window*) of a receive buffer into a different local buffer.

2.2 The Address Translation Problem

VMMC requires that the *receive buffer* be always resident in the physical memory. This requirement simplifies the communication protocol, because otherwise, incoming data may arrive on virtual page that has been paged out by the operating system.[3] During an export system call, the operating system is invoked to *pin* the receive buffer in memory and store the physical addresses on the network interface. The network interface can later obtain these physical addresses conveniently with the integer ID that represents the receive buffer in the system.

Unlike receiver buffers, the local buffers are dynamic resources. Their virtual addresses and sizes vary from one communication request to another. This is because VMMC permits an arbitrary buffer to be used in a remote store, remote fetch, or transfer redirection operation in order to support *zero-copy* [7]protocols. To be more specific,

- remote store - send data out of an arbitrary local buffer.
- remote fetch - fetch data into an arbitrary local buffer.
- transfer redirection - redirect incoming data into an arbitrary local buffer.

For local buffers, we need a method to inform the network interface of the virtual-to-physical translations of an arbitrary local buffer and guarantee that the buffer's virtual pages be resident in physical memory until the data transfer is complete.

On modern computer architectures, the virtual-to-physical translations are maintained and protected by

[3]To handle a missing page, the network interface will have to either interrupt the host processor to let it bring in the missing page from the slow disk or drop the data packet and rely on retransmission.

the host OS. A hierarchical page-table structure is typically used to store translations for 32-bit and 64-bit address spaces. On a platform such as Windows NT [6], the network interface cannot always "read" the system page tables because these tables may not be resident in physical memory. Furthermore, as an I/O device, the network interface has no control over paging and swapping in the operating system. Therefore, the translation that the network interface obtains from the system page table may become invalid when actual data transfer takes place.

3 Related Work

Translating arbitrary virtual addresses for the network interface is a difficult problem. A typical solution is the *system-call* approach which requires the user to initiate every data transfer inside the OS via a system call (or device driver `ioctl()`) and device interrupts. Typically a system call is used to send data and an interrupt to receive data, as in the Intel iPSC860 [10]. The OS can obtain correct address translations and prevent the memory subsystem from paging out relevant virtual pages. The system-call approach incurs high overhead due to the cost of system calls and device interrupts[4].

UNet-MM improves upon the system-call approach by caching translations in the network interface [1, 2]. When a virtual page address misses in the network interface translation cache, the host processor is interrupted to pin the virtual page and store the physical address in the cache. Interrupts are expensive and difficult to handle. For example, Windows NT cannot pin virtual pages in the interrupt context. UNet-MM resorts to using a separate thread that runs in normal context to pin pages. As a result, the overhead is quite high (16 μs).

The Intel Paragon presents an effective solution by dedicating one microprocessor in a Symmetric Multi-Processing (SMP) system as the communication processor [13, 12]. Running as part of the operating system, the communication processor has full access to the system page tables as well as complete control over paging and swapping. Instead of using the system call, the user process posts requests to a memory queue shared by both the compute processor and the communication processor. The Paragon approach eliminates the need for system calls and interrupts at the expense of taking a microprocessor away from doing useful computation.

In their paper, Buzzard et al [5] mention using some descriptors to store the physical addresses translations for a few user buffers. The descriptors are filled when the user process pins the buffer in physical memory. This concept is similar to UTLB which will be described shortly. However, their system has two drawbacks. First, the translation descriptors reside on the network interface and their number is limited. Second, pinning and unpinning of virtual pages is done on a per-buffer basis, whereas UTLB allows each virtual page to be pinned and unpinned independently.

4 Design of UTLB

The User-managed TLB (UTLB) mechanism is an effective and inexpensive solution to the address translation problem. In the UTLB architecture, the network interface looks up the virtual addresses in a translation table. The translation table is kept consistent with the system page table for the virtual pages that have been pinned in physical memory. The UTLB's user-level library is responsible for invoking the OS to pin virtual pages and update the translation table.

4.1 Main Ideas

The main ideas in UTLB are demand-driven page-pinning, protected translation table, and user-level lookup.

Our first idea is *demand-driven page-pinning*: pin the local buffer when it is used in communication for the first time. The address translations can be provided to the network interface during page-pinning. A special system call or device driver call is invoked to pin the virtual pages in physical memory. The buffer remains pinned in the physical memory so that subsequent data transfers involving the same buffer require no system call. This scheme improves upon the system-call approach by exploiting the spatial locality in most programs. The cost of a page-pinning system call is amortized over multiple communication requests.

Our second idea is to establish a *protected translation table* for pinned virtual pages. In the UTLB architecture, a translation table is allocated for each process. It contains physical addresses for the process' virtual pages that have been pinned in physical memory. The translation table is not visible to the user process. However, the user process can tell the operating system where in the table to store the physical translations for a given virtual buffer.[5] To specify a virtual page in a data transfer operation, the user process tells the network interface the index into the translation table where the physical

[4]A null `ioctl` call takes 12 μs on 300-MHz Pentium II PC running Windows NT. Within 12 μs, the Myrinet network interface can transfer at least 600 bytes between two computers' memory with a conservative 50 MB/sec end-to-end bandwidth for small messages. Future generations of network interfaces with faster data transfer rate may suffer even more from the context switching overhead.

[5]Hence, the name "User-managed" TLB.

memory address for the virtual page is stored. With the index, the network interface can directly obtain the physical address out of the translation table.

Our third idea is to construct a fast *user-level lookup* data structure that is modeled after the two-level page table of the Intel x86 architecture [11]. This is because the user process has to keep track of the mapping between the translation table indices and the pinned virtual pages. The page table contains one entry for each virtual page. Each entry can either be in the invalid state or contain the index into the translation table where the physical address for that virtual page is stored. Only two memory references are required to obtain the UTLB index for a given virtual page address.

The following pseudo-code illustrates the steps taken by the user process to send data from its virtual buffer.

```
User Program:
send_message(vaddr, nbytes)
    1) looks up (vaddr, npages) in the user-level
       table
    2) if (some pages in the range are not pinned)
          finds free UTLB translation entries
          invokes the ioctl call to
                 a) lock the pages
                 b) fill the translation entries
    3) submits the request to NI using the indices

Network Interface:
1) receives user request
2) obtains physical addresses by directly
   indexing the translation table
3) performs DMA
```

4.2 Implementation Platform

We implemented the UTLB mechanism on the Myrinet network interface [4]. The network interface has a 33MHz RISC microprocessor and 1MB Static RAM (SRAM). The network interface is attached to a 300 MHz Pentium-II PC via the PCI bus running Windows NT and our custom VMMC communication software (see Section 5.1). The VMMC device driver initializes the network interface and downloads the firmware, Myrinet Control Program (MCP), into the network interface SRAM. The driver also allocates a special command post buffer from the Myrinet SRAM and maps it into the user process' address space. The user process posts communication requests through the command buffer.

We modified the firmware for the Myrinet network interface to add the UTLB translation tables and lookup mechanism. An `ioctl()` call is added to the VMMC device driver for pinning virtual pages and storing physical addresses in the translation table. For our implementation, we did not modify the Windows NT operating system. An earlier implementation of UTLB on Linux was also done without system modification.

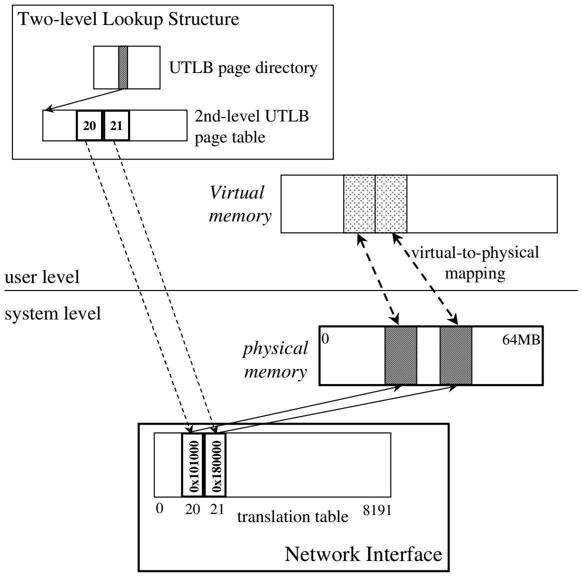


Figure 2: Structure of UTLB on VMMC

4.3 Per-Process UTLB

The initial implementation of UTLB allocated a 16K-entry translation table for each process. The translation tables reside in the network interface SRAM for fast access by the network interface. They are protected from the user processes and can only be modified by the device driver operating in kernel mode. Through a special device-driver `ioctl()` call, the user process can ask the OS to pin certain virtual pages and "install" their physical translations at specific locations in the translation table. These indices are then stored in the user-level lookup page tree. Figure 2 depicts the structure of our initial implementation of UTLB.

The device driver also allocates and pins a "garbage" page. All UTLB translation table entries are initialized with the physical memory address of the garbage page. This scheme saves the network interface from checking the validity of user-submitted indices: At worst, the network interface transfers data to and from an unused garbage page; no harm will be done to the system or other processes.

It is possible that the translation table may become full while more virtual pages need to be pinned. Such *capacity misses* are handled by *evicting* some translations already in the UTLB translation table. This results in unpinning of the virtual pages whose translations have been evicted from the UTLB. The UTLB library decides which translation table entries to evict and tells the OS to unpin the virtual pages. To reduce the frequency of capacity misses, the UTLB library monitors the virtual page usage and use a simple replacement policy such as *LRU* to select the victim entries for eviction.

On a workstation with large physical memory, a significant portion of a user process' virtual address space can be involved in data communication. The size of the UTLB translation table must be scaled up to reduce the *capacity misses*. However, the limited amount of SRAM on the Myrinet network interface places an upper bound on the size of a translation table. This upper bound is further limited by the need to set aside a significant portion of the SRAM for retransmission buffers. When multi-programming is employed, multiple translation tables may have to be present on the network interface at the same time. This is because the communication requests can be queued and processed even if the process is not running. The SMP presents an even tougher challenge: as many as four processors can be using the network interface simultaneously; only a quarter of available SRAM space can be used for one translation table.

4.4 Shared UTLB-Cache

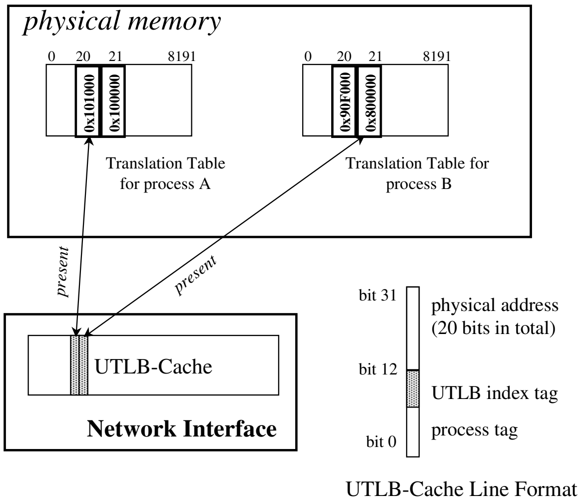


Figure 3: Structure of Shared UTLB-Cache

Our second approach is to implement a Shared UTLB-Cache on the network interface and move the *per-process* translation tables to the host physical memory (DRAM), as shown in Figure 3. The Shared UTLB-Cache caches the entries from the per-process translation tables. It is direct-mapped and indexed by the translation table index. Each Shared UTLB-Cache entry contains the process ID and part of translation table index to uniquely identify an entry from a particular translation table. When a miss occurs in the Shared UTLB-Cache, the network interface simply fetches (via DMA) the entry from the translation table in the physical memory. The cost of fetching an entry is only a couple of microseconds, making the Shared UTLB-Cache an efficient device.

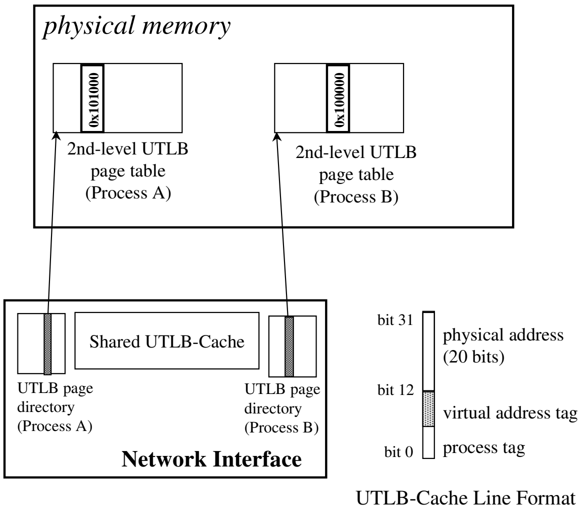


Figure 4: Structure of Hierarchical-UTLB

With the Shared UTLB-Cache mechanism, the UTLB translation tables can be large enough to cover the entire physical memory of the host.

4.5 Hierarchical-UTLB

The Hierarchical-UTLB is an enhancement of the UTLB. Instead of letting the user process search its lookup data structure for UTLB indicies, Hierarchical-UTLB directly uses the protected translation table as the lookup data structure. Under Hierarchical-UTLB, the user process submits virtual addresses to the network interface. The network interface directly searches Hierarchical-UTLB translation table for physical address translation (Figure 4.

The translation table in Hierarchical-UTLB resembles a two-level system page table tree: The first-level directory points to second-level page tables in physical memory. Each page table entry stores the physical address of a pinned virtual page. The Hierarchical-UTLB translation table differs from a real system page table by one important aspect: The entries in the second-level Hierarchical-UTLB translation table are the physical addresses of virtual pages that have been explicitly pinned by the user process.

The top-level directory of a Hierarchical-UTLB translation table is always stored in the network interface so that when there is a miss in the Shared UTLB-Cache it takes one memory reference in the SRAM to access the page directory and one DMA to access the second-level page table.

With Hierarchical-UTLB, the user-level library only needs a bit array to maintain the memory-pinning status of virtual pages. In addition, a process can directly use the virtual address to represent its buffers. Instead of indices, the network interface simply use the virtual address to lookup the physical address in the Shared UTLB-Cache or query the UTLB page table on a miss.

The Hierarchical-UTLB eliminates the need to handle UTLB fragmentation: after complex data accesses, a user buffer's translations may be scattered in the translation table. Integrating Hierarchical-UTLB with the translation table used for *receive buffers* is also straightforward. Virtual addresses can be used everywhere to represent both remote and local buffers.

To summarize, at the cost of one extra SRAM reference to process a UTLB cache miss, Hierarchical-UTLB simplifies the construction of UTLB and eliminates the fragmentation problem. In the rest of the paper, UTLB refers to Hierarchical-UTLB unless stated otherwise.

4.6 Discussion

The use of Shared UTLB-Cache introduces an interesting problem: cache misses. Handling a miss in the Shared UTLB-Cache involves DMAing a word from the host physical memory. This cost is much higher than the cost of a hit which is only a few instructions to read the SRAM on the network interface. A miss in the Shared UTLB-Cache will have perceivable impact on small-message latency.

When one process is using the network interface, both capacity misses and conflict misses may occur in the Shared UTLB-Cache. Capacity misses can be reduced by enlarging the size of Shared UTLB-Cache. Conflict misses can be alleviated by making the cache set-associative. Set-associativity can also help reduce *multi-process conflict misses*, those that result from multiple processes using the same virtual addresses, for example, in the case of Shared Virtual Memory (SVM) programs running on an SMP cluster. A simple scheme to reduce the conflict misses is to offset a translation table index by a process-dependent constant. The same index from different translation tables will be hashed into different locations in the Shared UTLB-Cache.

It turns out that a direct-mapped Shared UTLB-Cache with 8K entries is sufficient to handle 4 concurrent processes on a 4-way SMP. The majority of the misses are *cold misses* whose number is the total size of the virtual address space used in data transfer. Because most programs display spatial locality, we can prefetch extra UTLB translation entries when a miss occurs in Shared UTLB-Cache. Aggressive prefetching can reduce the frequency at which cold misses occur (i.e., cold miss rates). But it also incurs additional cost for fetching more entries. We evaluate the effect of Shared UTLB-Cache size, associativity, and prefetching in Section 6.

5 Basic UTLB Performance

The overall cost for translating a virtual page in the UTLB includes the overhead on the host processor and the overhead on the network interface. Each overhead varies depending on whether the virtual page is pinned and whether the physical address is in the UTLB network interface cache. The fastest path to translate a virtual address on the network interface is taken when the virtual page is pinned and its physical address is present in the UTLB network interface cache (a hit). The total overhead for this path is merely 0.9 μs (0.4 μs on the host and 0.5 μs on the network interface).

5.1 Experimental Environment

num_pages	1	2	4	8	16	32
check min	0.2	0.2	0.2	0.2	0.2	0.2
(μs) max	0.4	0.6	0.6	0.6	0.6	0.7
pin (μs)	27	30	36	47	70	115
unpin (μs)	25	30	36	50	80	139

Table 1: UTLB overhead on the host processor

We conducted our experiments on a cluster of PC workstations running Windows NT 4.0 operating system. Each PC consists of a 300 MHz Pentium II processor and a 32-bit wide PCI bus running at 33 MHz. A Myrinet network interface card with a LANai 4.2 microprocessor uses the PCI bus and connects the PC to the Myrinet network switch.

The time on the network interface is measured with LANai's real-time clock register, with an accuracy of 0.5 μs. Because the LANai 4.x processor has *no* instruction or data caches, averaging the total time of repeated operations gives the exact timing for an individual operation. On the host, the time is measured with the Pentium processor's cycle counter with an accuracy of a CPU clock cycle. Reading the cycle counter has an overhead of 39 cycles.

5.2 Host-side Performance

Table 5.1 lists the overhead of UTLB host-side operations: user-level lookup (as *check*), page-pinning (as *pin*), and page-unpinning (as *unpin*). The lookup procedure checks a bit map to see if the virtual pages of a buffer is already pinned in the physical memory. The cost of checking the bit map varies with the first bit's position in the bit map. The table reports both the minimum and maximum costs from all posible bit posisions.

If some virtual pages are not pinned, the lookup procedure invokes a device driver `ioctl()` call to pin these pages and store their physical addresses in the UTLB tranlation table. As the numbers suggest, page-pinning is an expensive operation. They validate the UTLB design strategy of on-demand pinning and translation caching.

Unpinning of virtual pages occurs when the UTLB translation table is full. This is a rare event since the translation table can be configured sufficiently large to cover the entire physical memory. Nevertheless, for completeness we list the cost of unpinning pages.[6]

5.3 Network Interface Performance

num_entries	1	2	4	8	16	32
hit (μs)	0.8					
DMA (μs)	1.5	1.6	1.6	1.9	2.1	2.5
miss (μs)	1.8	1.9	1.9	2.3	2.8	3.2

Table 2: UTLB overhead on the network interface

Table 5.3 shows the cost of UTLB operations on the network interface: lookup (as *hit cost*) and miss handling (as *miss cost*). The hit cost is the time it takes the network interface to look up a virtual page's physical address in the UTLB network interface cache. The number given here is for a direct-mapped cache with 8K entries[7].

The miss cost includes the time to obtain the physical address for the second-level page table (see Hierarchical-UTLB) and the time to DMA the entries into the UTLB network interface cache. Multiple entries are prefetched at once during a miss to exploit the spatial locality of a program's data access. The prefetching cost remains relatively constant with respect to the number of entries fetched because DMA setup dominates the total fetch time for a small number of words (see Table 5.3).

[6]Unpinning a virtual page is tricky because the device driver must synchronize with the network interface to make sure that no transfer is in progress on that page.

[7]The Myrinet VMMC firmware breaks down data transfer at 4KB page boundaries. The translation lookup is performed for each virtual page address. Therefore, we list the hit cost for only one entry here.

6 Trace-driven Analysis

As the results from previous section indicate, a miss in the UTLB network interface cache costs an order of magnitude more than a hit. The average UTLB cost on the network interface depends on the hit/miss ratio in the cache, which in turn depends on three parameters: cache size, cache associativity, and prefetching aggressiveness.

We used trace-driven analysis to evaluate the effect of each parameter and derive its optimal value. We obtained communication traces from an SMP cluster because the high degree of multi-processing on an SMP offered us the most stressful test on the UTLB network interface cache. The traces of communication requests, each of which was time stamped with a globally-synchronized clock, were generated from running a number of SPLASH2 [15] applications with the Home-based Release Consistency SVM protocol [16]. We ran these programs with the SMP version of the SVM protocol [14] on four 4-way SMPs. Each SMP contains four 200 MHz PentiumPro microprocessors and 256 MB of DRAM. Five processes, including a protocol process, are created on each SMP node.

6.1 Applications

- The RADIX program sorts an array of keys in parallel. The algorithm is divided into a number of radix phases. During a radix phase, each process radix-sorts a contiguous sequence of the keys. At the end of the phase, the results from each processes are combined to form a new array for subsequent radix-sort phases.

- **FFT** implements a parallel 2D Fast Fourier Transform algorithm. This program exhibits high degree of data communication.

- **LU** is a parallel LU decomposition program.

- **Barnes** implements the Barnes-Hut[Barnes] algorithm for NBody simulation. Each process gets a partition of the particles and caculates their new positions during one time step. Communication in this application is moderate as the particle partition exhibits spatial locality.

- **Water** caculates movements of molecules using an all-to-all caculation that results in an n-square complexity.

- **RayTrace** uses a task-farm model to raytrace a scene. Communication in RayTrace revolves around the task queues.

- **Volrend** uses a task-farm model to render a 3-D volume. Communication in this application also centers on the task queues.

6.2 The Effect of Prefetching

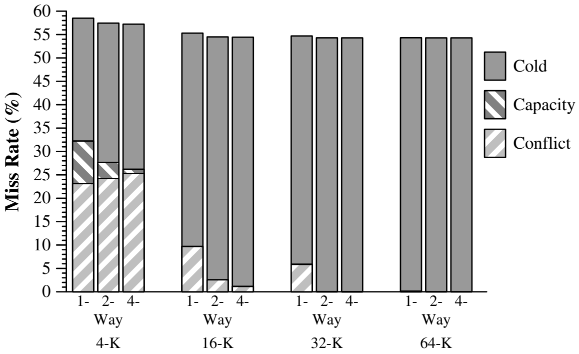
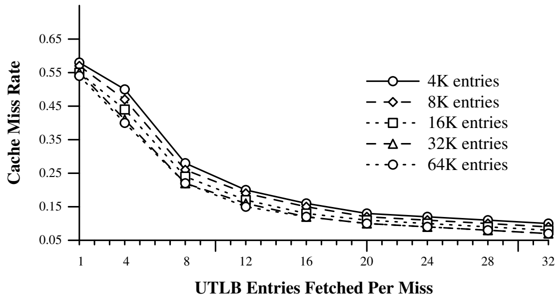


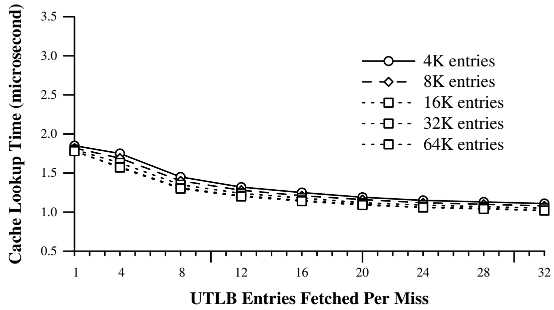
Figure 5: Miss Rates in the Network Interface UTLB Cache

Our trace-driven analysis discovered that when the size of UTLB network interface cache exceeds 4K entries the majority of the misses are cold misses. Figure 5 shows the breakdown of network interface cache miss rates when we ran the RADIX program. The graph indicates that as the size of the network interface UTLB cache grows, cold misses quickly become the dominant component of UTLB lookup misses.

To reduce cold miss rates, we let the Shared UTLB-Cache prefetch multiple translation entries when an entry misses in the network interface cache. Table 3 details the overall miss rates and overall UTLB lookup cost on the network interface for different prefetching sizes and a sufficiently large Shared UTLB-Cache size (8 KB). As one would expect, the overall miss rate decreases as prefetching becomes more aggressive. To further illustrate this point, we plot the miss rates and lookup cost from RADIX as a function of prefetching aggressiveness in two graphs shown in Figure 6. These two graphs clearly show that prefetching can effectively reduce overall miss rate. Because the cost of prefetching multiple UTLB entries increases much slower than the rate at which overall miss rate drops, the average lookup cost decreases as prefetching becomes more aggressive. In our implementation of UTLB, we chose 16 as the prefetching size.



Cache Miss Rates vs. Prefetch Size



Average Cache Lookup Cost vs. Prefetch Size

Figure 6: The Effect of Prefetching in UTLB Network Interface Cache

6.3 The Effect of Cache Size

Cache size and associativity directly affect the capacity and conflict miss rates. Table 4 shows the overall miss rates in the network interface cache for varying cache sizes and three different associativities. We also present the average cost of a UTLB lookup in the network interface in Table 5. The trace-driven analysis shows that direct-mapped cache perform surprisingly well compared with the set-associative caches. This interesting behavior is best illustrated in Figure 7, which compares the overall UTLB miss rates and average lookup cost among three set-associativity for RADIX program.

We think that the reason why direct-mapped UTLB cache performs well is due to the fact that we already offset each virtual addresses with a process-dependent constant in the network interface. When the actual cost of lookup is considered, the set-associative caches lose to the direct-map cache simply because the 2- and 4-way set-associative caches check more entries per cache line than the direct-mapped cache. Since we implement set associativity in Myrinet firmware, set-associative caches pay higher cost for cache lookup than a direct-mapped cache.

The trace-driven analysis justifies our choice of direct-mapping for the cache. From the graphs, we conclude

Prefetch Size	Associativity	**Applications**						
		Radix	LU	FFT	Barnes	RayTrace	Volrend	Water
1 entry	direct	0.57	0.50	0.47	0.05	0.44	0.26	0.04
	2-way	0.55	0.50	0.44	0.05	0.44	0.25	0.04
	4-way	0.55	0.50	0.42	0.05	0.44	0.25	0.04
4 entries	direct	0.47	0.13	0.48	0.04	0.24	0.20	0.01
	2-way	0.46	0.13	0.47	0.03	0.24	0.20	0.02
	4-way	0.47	0.18	0.45	0.03	0.24	0.20	0.01
8 entries	direct	0.26	0.07	0.50	0.03	0.17	0.20	0.01
	2-way	0.26	0.07	0.48	0.03	0.17	0.19	0.01
	4-way	0.27	0.15	0.47	0.03	0.18	0.21	0.01
12 entries	direct	0.19	0.05	0.26	0.03	0.14	0.20	0.01
	2-way	0.18	0.05	0.25	0.03	0.14	0.20	0.01
	4-way	0.19	0.14	0.25	0.03	0.15	0.21	0.01
16 entries	direct	0.15	0.04	0.26	0.03	0.13	0.19	0.01
	2-way	0.15	0.04	0.25	0.03	0.13	0.18	0.01
	4-way	0.16	0.14	0.25	0.04	0.14	0.20	0.01
20 entries	direct	0.12	0.03	0.18	0.03	0.12	0.19	0.01
	2-way	0.12	0.04	0.17	0.03	0.12	0.18	0.01
	4-way	0.14	0.14	0.17	0.04	0.13	0.20	0.01
24 entries	direct	0.11	0.03	0.18	0.03	0.11	0.19	0.01
	2-way	0.11	0.03	0.17	0.03	0.11	0.18	0.01
	4-way	0.12	0.14	0.17	0.04	0.12	0.20	0.01
28 entries	direct	0.10	0.03	0.14	0.03	0.10	0.18	0.00
	2-way	0.10	0.03	0.14	0.03	0.10	0.17	0.01
	4-way	0.11	0.14	0.13	0.05	0.12	0.19	0.01
32 entries	direct	0.09	0.03	0.14	0.03	0.10	0.14	0.00
	2-way	0.09	0.03	0.14	0.03	0.10	0.13	0.01
	4-way	0.11	0.14	0.13	0.05	0.12	0.16	0.01

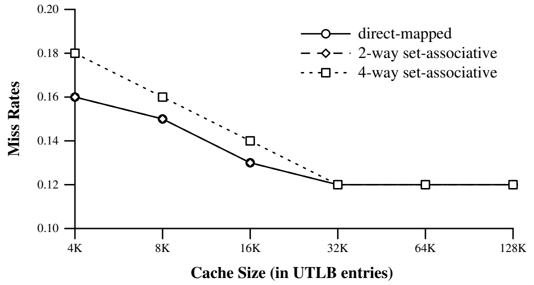
Table 3: Overall Miss Rates in Shared UTLB-Cache vs. Prefetching Size (cache size = 8K entries)

Cache	Associativity	Applications						
		Radix	LU	FFT	Barnes	RayTrace	Volrend	Water
4K	direct	0.16	0.04	0.26	0.04	0.14	0.23	0.01
	2-way	0.16	0.04	0.26	0.05	0.14	0.23	0.01
	4-way	0.18	0.17	0.26	0.07	0.17	0.25	0.01
8K	direct	0.15	0.04	0.26	0.03	0.13	0.19	0.01
	2-way	0.15	0.04	0.25	0.03	0.13	0.18	0.01
	4-way	0.16	0.14	0.25	0.04	0.14	0.20	0.01
16K	direct	0.13	0.04	0.25	0.02	0.9	0.16	0.01
	2-way	0.13	0.04	0.24	0.02	0.9	0.15	0.01
	4-way	0.14	0.9	0.23	0.02	0.12	0.16	0.01
32K	direct	0.12	0.04	0.23	0.02	0.9	0.15	0.01
	2-way	0.12	0.04	0.22	0.01	0.9	0.14	0.01
	4-way	0.12	0.04	0.21	0.02	0.9	0.15	0.01
64K	direct	0.12	0.04	0.19	0.01	0.9	0.14	0.01
	2-way	0.12	0.04	0.18	0.01	0.10	0.14	0.01
	4-way	0.12	0.04	0.17	0.01	0.10	0.14	0.01

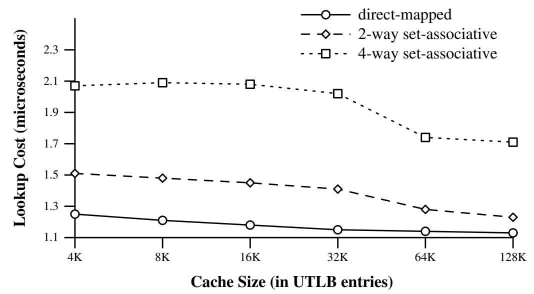
Table 4: Overall Miss Rates in Shared UTLB-Cache vs. Cache Size (prefetch=16 entries)

Cache	Associativity	Applications						
		Radix	LU	FFT	Barnes	RayTrace	Volrend	Water
4K	direct	1.25	0.92	1.53	0.92	1.19	1.43	0.82
	2-way	1.51	1.01	1.75	1.08	1.45	1.73	0.89
	4-way	2.07	2.03	2.36	1.60	2.07	2.49	1.03
8K	direct	1.21	0.91	1.52	0.88	1.15	1.32	0.82
	2-way	1.48	1.04	1.72	1.01	1.35	1.57	0.90
	4-way	2.09	1.96	2.33	1.50	2.01	2.30	0.98
16K	direct	1.18	0.91	1.51	0.85	1.12	1.26	0.82
	2-way	1.45	1.07	1.70	1.05	1.33	1.39	0.89
	4-way	2.08	1.90	2.30	1.54	1.96	2.01	0.97
32K	direct	1.15	0.91	1.45	0.84	1.11	1.22	0.82
	2-way	1.41	1.07	1.66	0.99	1.32	1.39	0.87
	4-way	2.02	1.63	2.26	1.46	1.91	1.94	0.96
64K	direct	1.14	0.91	1.34	0.84	1.10	1.18	0.82
	2-way	1.28	1.12	1.52	0.94	1.24	1.31	0.87
	4-way	1.74	1.74	2.13	1.39	1.86	1.85	0.96

Table 5: Overall Lookup Cost vs. Shared UTLB-Cache Cache Size (prefetch=16 entries)



Cache Miss Rates vs. Cache Size (Prefetch=8)



Average Lookup Cost vs. Cache Size (Prefetch=8)

Figure 7: UTLB Network Interface Cache Performance For RADIX

that either 8K or 16K entries is sufficient for the cache size. The actual size for the UTLB network interface cache is 32 KB or 64 KB.

7 Conclusions

In this paper we described the design and implementation of UTLB, a user-managed address translation mechanism for network interfaces. By maintaining a user-level lookup data structure and a protected translation table, UTLB avoids system calls in the common path of communication and completely eliminates device interrupts.

We also presented two enhancements to the original UTLB design: Shared UTLB-Cache and Hierarchical-UTLB. The Shared UTLB-Cache structure allows an arbitrarily large per-process translation table to be constructed in physical memory. We have shown with micro-measurements and trace-driven analysis that extra overhead of accessing a UTLB translation entry over the system I/O bus is low. This overhead can further be reduced by prefetching multiple translation entries upon a miss in the Shared UTLB-Cache.

The Hierarchical-UTLB design further simplifies the construction of UTLB. It also opens up an opportunity to combine the address translation for local buffers with those for receive buffers. In particular, virtual addresses can be used uniformly to represent all buffers. We plan to implement such integration soon.

Our implementation and experiments were done on a 32-bit microprocessor architecture. The design tradeoffs may be different if the host microprocessor uses a 64-bit architecture such as Digital Alpha and MIPS R10000. For example, the Hierarchical-UTLB design may require much physical memory to be pinned as second-level UTLB translation tables. Whereas the user-level lookup table in the Shared UTLB-Cache (or original UTLB) design can be paged out to the disks. It remains a research question which design is suitable in that case.

References

[1] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-net: A user-level network interface for parallel and distributed computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, Colorado*, December 1995.

[2] Anindya Basu, Matt Welsh, and Thorst von Eicken. Incorporating memory management into user-level network interfaces. http://www2.cs.cornell.edu/U-Net/papers/unetmm.pdf, 1996.

[3] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[4] 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, February 1995.

[5] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the hamlyn sender managed interface architecture. In *Proceedings of the Operating Systems Design and Implementation Symposium*, October 1996.

[6] Helen Custer. *Inside Windows NT*. Microsoft, 1993.

[7] S. Damianakis, A. Bilas, C. Dubnicki, and E.W. Felten. Client-server computing on shrimp. *IEEE Micro*, 17(1):8–18, February 1997.

[8] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. Shrimp project update: Myrinet communication. *IEEE Micro*, 18(1):50–52, January 1998.

[9] C. Dubnicki, L. Iftode, E.W. Felten, and K. Li. Software support for virtual memory-mapped communication. In *Proceedings of the 1996 International Parallel Processing Symposium*, 1996.

[10] The ipsc/860. Technical Reference Manual, 1991.

[11] Intel Corporation. *Pentium Processor Data Book*, 1993.

[12] Paul Pierce. The Paragon implementation of the NX message passing interface. In *Proceedings of Scalable High-Performance Computing Conference (SHPCC) 94*, 1994.

[13] Justin Rattner. Paragon system. In *Presentation in the DARPA High Performance Software Conference*, January 1992.

[14] Rudrajit Samanta, Angelos Bilas, Liviu Iftode, and Jaswinder Pal Singh. Home-based svm protocols for smp clusters: Design and performance. 1998.

[15] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological considerations and characterization of the splash-2 parallel application suite. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, May 1995.

[16] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Operating Systems Design and Implementat ion Symposium*, October 1996.