# 대용량 메모리 데이타 처리를 위한 범용 하드웨어 기반의 원격 메모리 시스템
# (Large-Memory Data Processing on a Remote Memory System using Commodity Hardware)

정 형 수 † 한 혁 †† 염 헌 영 †††
(Hyungsoo Jung) (Hyuck Han) (Heon Y. Yeom)

**요 약** 본 논문에서는 대용량 메모리 데이타 처리를 위한 범용 하드웨어 기반의 원격 메모리 시스템을 제안한다. 느린 디스크와 상대적으로 대단히 빠른 접근 속도를 보장하는 메모리 사이에 존재하게 되는 새로운 메모리 계층을 구현하기 위해, 본 논문에서는 다수의 일반적인 범용 데스크탑 PC들과 원격 직접 메모리 접근 (이하 RDMA) 기능이 가능한 고속 네트워크를 최대한 활용하였다. 제안된 새로운 계층의 메모리는 합리적인 응답시간과 용량을 제공함으로서 비교적 적은 양의 성능 부담으로서 대용량의 메모리 상주 데이타베이스를 구동할 수 있게 되었다. 제안된 원격 메모리 시스템은 원격 메모리 페이지들을 관리하게 되는 원격 메모리 시스템과, 원격 메모리 페이지의 교체를 관리하게 되는 원격 메모리 페이저로 구성되어 있다. 범용으로 쓰이는 MySQL과 같은 데이타베이스를 이용한 TPC-C 실험 결과로 볼 때 제안된 원격 메모리 시스템은 일반적인 대용량 메모리 데이타 처리 시스템에서 요구하는 다양한 요구조건을 만족시킬 수 있을 것이라 생각된다.
**키워드** : 대용량 메모리 데이타 처리, 원격 메모리 시스템, 인피니밴드, RDMA

***Abstract*** This article presents a novel infrastructure for large-memory database processing using commodity hardware with operating system support. We exploit inexpensive PCs and a high-speed network capable of Remote Direct Memory Access (RDMA) operations to build a new memory hierarchy between fast volatile memory and slow disk storage. The new memory hierarchy guarantees a reasonable response time, and its storage size enables us to run large-memory database systems with little performance degradation. The proposed architecture has two main components: (1) a remote memory system inside the Linux kernel to manage other computers' memory pages efficiently and (2) a remote memory pager responsible for manipulating remote read/write operations on remote memory pages. We insist that the proposed architecture is practical enough to support the rigorous demands of commercial in-memory database systems by demonstrating the performance of publicly available main-memory databases (e.g., MySQL) on our prototyped system. The experimental results show very interesting results from the TPC-C benchmark.
**Key words** : Large-memory data processing, Remote memory system, InfiniBand, RDMA

## 1. Introduction

The attractive feature of random access memory, that its access time is tens of thousands times faster than that of disk storage, forces existing database architectures to be redesigned so that all database components can fit in the volatile memory space. Much research effort has been made to develop high-performance in-memory database systems that are orders of magnitude faster than disk-based database systems. These efforts are

· This study was supported by the Seoul Research and Business Development Program, Seoul, Korea. The ICT at Seoul National University provides research facilities for this study.
† 정 회 원 : 서울대학교 컴퓨터공학부
login.root@gmail.com
†† 학생회원 : 서울대학교 컴퓨터공학부
hhyuck@gmail.com
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
yeom@snu.ac.kr
논문접수 : 2007년 6월 13일
심사완료 : 2007년 7월 24일

driven by the decreasing price and increasing capacity of volatile memory, which makes building computers with large main memories not only possible, but affordable. Soon, it will not be difficult to see a terabyte of volatile memory provided as a buffer pool for a very large database.

Despite the performance advantage, in-memory database systems are rarely used commercially and have many issues that need to be addressed for them to be usable in large 64-bits systems. To fully utilize the fast random access memory feature, the indexing structure and data layout should be designed differently from that of disk-based database systems to maximize database processing throughput. This is the problem currently being addressed by database researchers. However, with a naive setting of MySQL, we were able to obtain processing time some magnitutes faster than its disk-based counterpart, from which we can see the superior performance of in-memory database systems.

The most important issue aside from database design for in-memory databases to be used commercially is to find a cost-effective way to build large memory computer systems at an affordable price. This is the tradeoff between a single expensive, large memory mainframe and clustered, inexpensive new memory hierarchy systems. This tradeoff is ascribed to the innate limitation of in-memory database systems - that is, the in-memory database must always reside in the volatile memory lest its performance decline drastically under overcommitted situations. Henceforth, the high cost of large memory systems which can support up to a terabyte of memory at the cost of millions of dollars is the only option remaining for clients to choose from when they want to run large in-memory databases.

In this article, we propose a new memory hierarchy infrastructure using inexpensive cluster computers interconnected by a multi-gigabit network interface capable of RDMA operations in order to satisfy the large quantitative memory requirement of in-memory database systems. A new memory hierarchy is constructed on a farm of cluster computers connected by a very high-speed network, InfiniBand, which can exert a maximum bandwidth of 10 Gbps. The main software part, especially the virtual memory and swap system, is substantiated inside the Linux kernel using various RDMA operations supported by the InfiniBand network system.

The reason we instrument the operating system is that current general purpose operating systems are somewhat inappropriate for supporting large memory database systems. Given the general purpose services of current OSes, it is worth noting that the database system itself is a very complex system, which already includes numerous components that overlap with some of the operating system's services, such as the buffer management service for managing memory resources efficiently. Even worse, the key service to share another machine's memory is not implemented successfully in general purpose operating systems. It, therefore, seems reasonable to conclude that designing a specialized operating system is meaningful to support large memory database systems.

The key design goal of our system is to construct a new memory hierarchy residing between volatile memory and swap disk. The access time of the new memory hierarchy is faster than that of disks but slower than that of local memory. Our new memory hierarchy currently supports static configuration among cluster systems, and for efficiency, we employ a delayed bulk write policy when we write memory pages to the new memory hierarchy system. Since the new memory hierarchy is dependent on the network, it is vulnerable to network failures, but we assume recovery from such failures is supported by the database system.

We show that if the size of the new memory hierarchy is sufficient, for sufficiently large relations, the performance of in-memory database systems on the proposed infrastructure is comparable to that of pure in-memory database systems. We demonstrate its performance using two well-recognized in-memory database systems, MySQL with the in-memory option. The rest of the article is organized as follows. Background materials, including a brief introduction of the InfiniBand architecture, are presented in Section 2.

Section 3 gives a detailed explanation of a new memory sharing architecture. Section 4 describes implementation details, and Section 5 reveals very interesting results of our rigorous experiments.

## 2. Related Work and Background

### 2.1 Memory sharing

Memory hierarchy and memory sharing has been a traditional research topic among many researchers. Closely related work on memory management for distributed architectures includes page placement strategy for distributed shared memory architectures, and shared virtual memory systems. The cost of a local memory access is significantly lower than accessing remote memory. Research work[1-4] in this area have shown that dynamic page replacement is an effective solution to the problem. However, it is not recommended to use this technique in cluster architectures which rely heavily on explicit message communications unless it has special hardware for direct access to remote memory.

There have been also numerous research work on a remote memory sharing. Feeley's work[5] presents global memory management in a workstation cluster. The system employs a single, but distributed memory management algorithm to manage all cluster-wide memory. The global memory manager can be regarded as global paging system, and it thus has global aging mechanism in applying page replacement policy. But, it involves explicit inter-node communication to exchange various control messages.

Comer[6] described a remote memory model in which the cluster contains workstations, disk servers. and remote memory servers. The remote memory servers were dedicated machines whose large primary memories could be allocated by workstations with heavy paging activity. No client-to-client resource sharing occurred, except through the servers.

Franklin et al.[7] examine the use of remote memory in a client-server DBMS system. Their system assumes a centralized database server that contains the disks for stable store plus a large memory cache. Clients interact with each other via a central server. On a page read request, if the page is not cached in the server's memory, the server checks whether another client has that page cached; if so, the server asks that client to forward its copy to the workstation requesting the read. Franklin et al. evaluate several variants of this algorithm using a synthetic database workload.

Dahlin et al.[8] evaluate the use of several algorithms for utilizing remote memory, the best of which is called N-chance forwarding. Using N-chance forwarding, when a node is about to replace a page, it checks whether that page is the last copy in the cluster; if so, the node forwards that page to a randomly-picked node, otherwise it discards the page. Each page sent to remote memory has a circulation count, N, and the page is discarded after it has been forwarded to N nodes. When a node receives a remote page, that page is made the youngest on its LRU list, possibly displacing another page on that node; if possible, a duplicate page or recirculating page is chosen for replacement.

### 2.2 InfiniBand network

InfiniBand is an architecture and specification for data flow between processors and I/O devices that aims to provide greater bandwidth than the Peripheral Component Interconnect (PCI) shared-bus approach used in most of today's personal computers and servers. Offering throughput of up to 2.5 gigabytes per second and support for up to 64,000 addressable devices, the architecture also promises better sharing of data between clustered processors.

On InfiniBand, data is transmitted in packets in the form of a communication unit called a message. A message can be an RDMA read or write operation, a channel send or receive message, a transaction-based operation, or a multicast transmission. The main RDMA interface in a new memory hierarchy was implemented on IB Gold 1.8.0 provided by Mellanox Technologies, which consists of drivers, protocols, and management applications from the open source OpenIB software suite in a simple, ready-to-install package. Of the software components included in the Mellanox InfiniBand Gold Distribution, the OpenIB.org InfiniBand Driver was modified.

## 3. Architecture

As mentioned in the previous section, architectures of new memory hierarchies to exploit the concept of remote memory sharing have been researched for quite a long time. The fundamental framework we assume in this article is based on distributed shared memory systems. The proposed architecture consists of two main components: (1) a *remote memory system* and (2) a *remote memory pager*. First, the remote memory system can be viewed as a framework which contains a remote managing functionality. The management functions include registrating memory and maintaining a *memory pool*, which is a set of remote memory pages. Second, the remote memory pager mainly performs the role of page allocator for remote memory pages. This leads us to make a *global page table* for that purpose. In this section, we describe the architecture and internal mechanism of our system in detail and explain practical matters down to the minutest details.

### 3.1 Memory hierarchy

The starting point to describe the memory sharing architecture is the *memory hierarchy*. Our intention is to create a new *memory abstraction* which is larger but slower than local random access memory. A memory hierarchy including a newly created memory level is shown in Figure 1. Unlike traditional memory hierarchy diagrams, we create an intermediate level between random access memory and disk storage. We call it *remote memory*, and the entire framework is called the *remote memory system*. The new memory level has numerous interesting characteristics compared to both random access memory and disk storage.

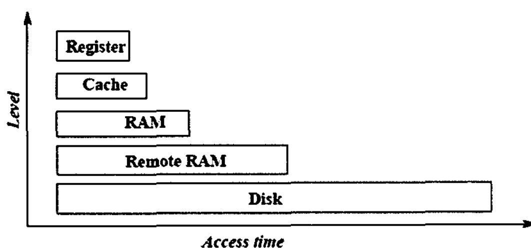



Figure 1 Various levels and access time of the new memory hierarchy

First, the access time of remote memory is a thousand times slower than that of random access memory, but it is more than a thousand times faster than that of disk storage. The slow access time is obviously due to the extra work required to access remotely located memory pages. The extra work consists of two main parts - a page reclamation mechanism and RDMA read/write operations on remote pages. The details of these mechanisms will be described later.

Second, although remote memory has slower access time than RAM, its access time behavior is very much like that of RAM in that it can provide almost uniform access time to its remote pages. This is due to the InfiniBand network, which has high bandwidth and low delay. It is thus possible to access any remote page in a given time bound with high fidelity, and this is an important characteristic because it enables us to build a remote memory system with low delay. In addition to the above feature, the remote memory system has a trait that the larger the data transferred, the higher the net bandwidth. This is a traditional feature of any network interface, and we exploit the above feature by adopting a delayed bulk write policy when we have large data to be sent to a remote memory space. This enhances the performance of the remote memory system significantly.

### 3.2 Remote memory system

This section looks further inside the core software part of the *remote memory system*. The schematic structure of the remote memory system is drawn in Figure 2. Figure 2 shows a snapshot of the remote memory system. Except for the machine that runs the in-memory database, all machines are configured to provide a part of their physical memory area to the remote memory system.

3.2.1 Memory registration

For each machine, to access another machine's memory whenever it is required, (1) each machine has to register its physical memory space as DMA-able memory to the InfiniBand network interface card (NIC), (2) the NIC must export its local memory to all machines and gather other machines' memory information by exchanging con-

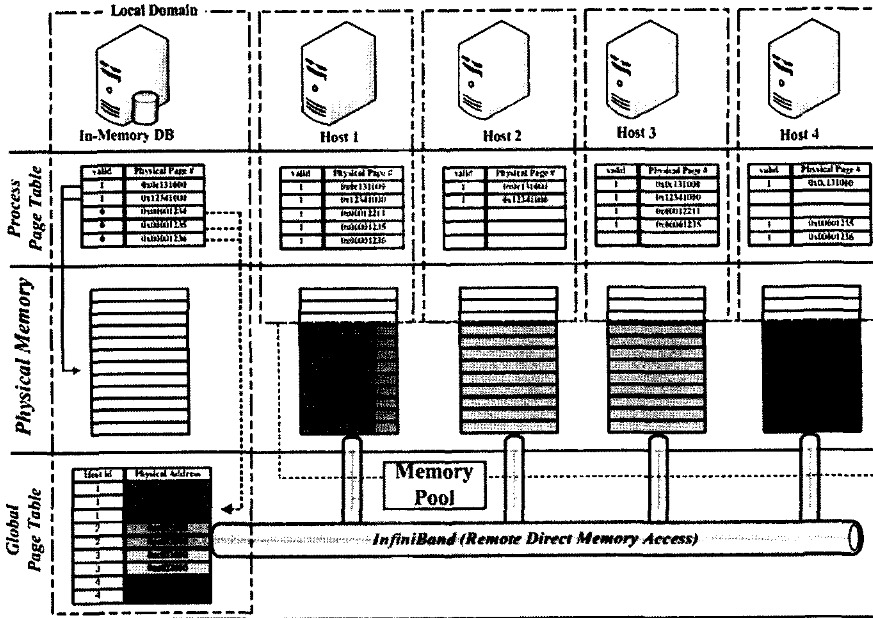



Figure 2 Structure of the remote memory system

trol messages, and (3) the NIC must support RDMA mechanisms, and its driver should provide proper RDMA operations to both user processes and kernel components to access another machine's memory directly.

Once all machines complete exporting their memory to the remote memory system, the in-memory database machine can immediately use the remote memory through a well-defined interface. Since the remote memory system relies heavily on RDMA operations, we define a common interface for the Linux kernel to access another machine's physical memory more easily without interfering with the remote machine. This zero-interfering access interface can only be made possible by exploiting a raw RDMA operation, which is exported as a kernel API by InfiniBand.

**Improvement.** Transferring data using an RDMA operation between two machines requires the kernel to register two memory spaces, the source and destination areas, lest the CPU become involved. Because the InfiniBand network heavily exploits RDMA operations, it is obvious to see that the registering overhead cannot be ignored. Therefore, it is much more efficient for the Linux kernel to register its entire memory space as DMA-able memory to the InfiniBand's network interface card at once than to register a candidate memory page on demand.

To minimize the modification of the Linux virtual memory system, we rarely modify the main virtual memory component, i.e., the *zone-buddy allocator*. The only part we instrument in the VM system is the *paging_init* function and a couple of related header files in order to set up a physical memory area to be exported to the remote memory system. The remaining part of the remote memory system is implemented as a kernel module. Detailed explanations of each part of the remote memory system are given below.

3.2.2 Memory pool

In the remote memory system, a *memory pool* serves as a remote page pool to fulfill remote page allocation. When an in-memory database machine needs pages from the remote memory system, the remote memory system allocates available pages from the memory pool by looking up a *global page table*. Therefore, all cluster computers that participate in the remote memory system must be known to initiate the remote memory sharing

mechanism. Having exchanged the information, we can see that pages of each machine that are exported to the remote memory system are regarded as the actual remote *memory pool*. As previously stated, our primary goal is to build a remote memory management system, and security issues are not within the scope of this article. So we assume that the cluster machines do not need to authenticate each other for remote paging to work, but a machine may crash.

Since we designed the remote memory system symmetrically, all machines run the same algorithm and attempt to make choices that are good both globally and locally. So we divide a machine's physical memory space into two segments at kernel's boot time. Both segments are configured as contiguous memory regions, and they are recognized as the kernel's available memory space at initial time. The colored region of each memory space in Figure 2 indicates the second contiguous memory fragment. We, however, instrumented the second memory region as non-manageable memory space by isolating it from the kernel's accessible zone list, so that the kernel can see its existence but cannot use the second contiguous memory space as available page frames. When the kernel finishes its initial page frame construction from raw memory space, all pages on a machine are classified as being either local pages, which can only be accessed by a local machine, or global pages, which are exported to the remote memory system as available remote page pool managed by another machine's remote memory manager on behalf of itself.

### 3.3 Remote memory pager

Having described the remote memory system, it is now time to focus on the remote memory pager, particularly as it is articulated with greater details. In defining the role of the remote memory pager, it may be useful to begin with an explanation about some related mechanisms in page allocation. While we describe each mechanism, we also give the role of the remote memory pager as well.

#### 3.3.1 Page reclamation

*Page reclamation* occurs when the virtual memory management (VMM) system faces a shortage of available page frames. In a traditional VM system, when the kernel undergoes such an urgent case, it usually activates a *swapper* to initiate a page reclamation mechanism. Originally, page reclamation was used to maintain a minimal amount of free page frames so that the kernel can safely handle out-of-memory situations. In the event of this critical situation, the main work is performed by the *swapper* process. The main job of the swapper is to find the most adequate candidate pages to reclaim to satisfy the current memory allocation demand. If the swapper gathers enough page frames from various sources such as dirty buffers or disk caches, it tries to write them to the swap space. It then returns the corresponding pages to the VM system to recover from the memory shortages.

However, in the remote memory system, the swapper is replaced to the remote memory pager. It first demands available pages from the remote memory system rather than allocate valid swap space from local swap device, then it writes old pages to remote pages instead of writing them to the swap device. If we observe more carefully, we can easily note that the remote memory pager may activate two page transmissions, a *page-in* and *page-out*. Logically, page-outs are mostly done in advance to page-ins because available page frames are obtained easily only if old page frames go out to the swap space. We, therefore, replace both paging activities with our common RDMA interfaces to enable the swap in/out to be redirected to the remote memory page.

When the remote memory pager needs available remote memory space, it should look up the global page table maintained by the remote memory system, which is responsible for managing metadata for all remote pages exported from other cluster machines. The remote pager then gets valid remote page identifiers for available remote pages. Once It obtains enough page idetifiers, it starts to send local page contents to the remote memory pages directly.

#### 3.3.2 Global page table

The *global page table* shown in Figure \ref {global_pte} is the only interface in which the

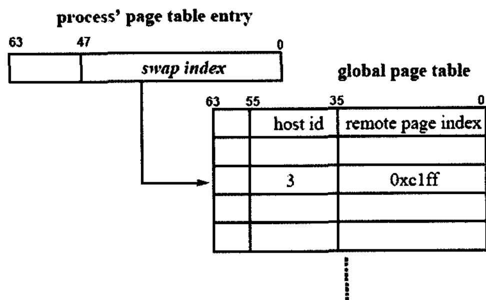



Figure 3 Index translation in the global page table

kernel can check for the existence of remote pages and obtain available remote pages. The role of the global page table is to maintain the location information of all remote pages registered to the memory pool, and this location information is retrieved by the remote memory pager whenever it needs available remote pages.

The global page table shown in Figure \ref {global_pte} is a table structure implemented in the x86_64 architecture. The table consists of 64 bits-wide entries. Each entry has three partitions: (1) 8 bits for various flags, (2) 20 bits for a host identifier, and (3) 36 bits for a remote page index. Flags represent various states of the corresponding remote page, such as a valid bit, usage count, and page size. A host identifier is initialized at registration time, and translation from host identifier to InfiniBand's physical address of a remote host is done in our driver routine. The last field is a remote page index. By default, each index value is created by four kilobyte units, which is the standard page size. We can change the unit of size by setting the page size bit in the flags. The initialization of the global page table is performed at boot time and maintained by the remote memory system. When a new memory space is registered as remote memory space, the kernel driver dynamically allocates proper entries and fills the required information for the new pages.

To find an available remote page identifier, the remote memory pager uses the *swap index* as it did before. The translation of the swap index to the appropriate global page table entry is performed in a straightforward way. The position of global page table entry is indicated by the value of the swap index. So, it is not hard to extract a page identifier that contains metadata such as various flags, a host identifier, and the remote page index of the remote host.

## 4. Implementation

The prototyped remote memory system is implemented for the Linux operating system [9,10]. Most components are implemented in a straightforward way.

### 4.1 Problem in the Linux Swap System

From a top-down view, the entry point to the remote memory system starts at the *swap index* generation routine. The unique way to find the location of a remote data page is the swap index. Sadly, the current implementation of the index generation, even in the latest Linux, i.e., RedHat kernel-2.6.13-15, is inappropriate to use as it is. The current implementation, once contiguously available index slots are running out, searches a free index slot linearly from the lowest free slot position in an index slot array. Therefore, if the search reaches the end of the swap index array, the search time might be very sensitive to the memory access behavior of the running program.

Figure 4 shows the result of our initial experience with the original index lookup algorithm, and we can observe very drastic performance degradation when running in-memory databases for long time. Even if the total amount of used swap space (2.7GB) is slightly more than half of the total swap size (3.5GB), once the searching mode is switched to the linear scanning mode after 1200 sec, the algorithm might consume significant amount of time in searching a free index slot, that is expected to take constant time. This unexpected pitfall happens when all free index slots are positioned at both ends of the index array with a high hit ratio, and used slots are spread around the middle of the array with a very low hit ratio. This phenomenon emphasizes that this optimistically designed linear algorithm can be easily broken by the program whose memory access behavior has high spatial locality.

### 4.2 Enhancement

Hierarchical bit-vector compression. For efficiency,

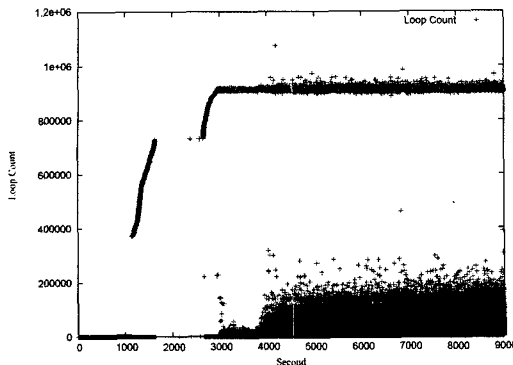



Figure 4 **Trace of Loop Count of the Linear Index Lookup Algorithm.** The loop count is traced at the point where the free swap index slot is searched. Once the index lookup algorithm is switched to the linear scanning mode, the trace has a huge amount of unexpected spikes due to the memory access behavior of the MySQL database (on the remote memory system), that shows strong spatial locality. (0-1700 sec: a data insertion period, 1600-2600 sec: a delimitation period, and 2600-9000 sec: under the TPC-C benchmark test)

we rewrote Linux's swap index generation routine in a space and time efficient manner. We implement a hierarchical bit-vector compression algorithm for searching the free index slot or inserting the one after using it. Each bit represents a single index slot which points to a single page. Therefore, to cover maximum 1 terabyte, we implement the algorithm having 7 hierarchies. The bit-vector compression algorithm consumes less than 4MB to cover 64GB index slots and takes constant time to flip a used index slot or to find the free index slot irrespective of the memory access behavior. This eliminates the problem we met and plays a critical role in improving overall efficiency.

## 5. Experiment

In this section, we analyze the performance of our system using MySQL Cluster. MySQL Cluster is a technology which enables clustering of in-memory databases in a shared-nothing system[11]. Core components of the MySQL Cluster are *mysqld, ndbd,* and *ndb_mgmd. mysqld* is the process which allows external clients to access the data in the cluster. *ndbd* stores data in its memory and supports replication and fragments. *ndb_mgmd* manages other processes in the cluster.

Two different experiments were conducted based on the type of machines that execute the *ndbd* processes. In the case of 32-bit machines, the maximum database size is between 2GB and 4GB, while the size of a database in a 64-bit machine is not limited. We executed a microbenchmark and a TPC-C-like benchmark on small databases in 32-bit machines.

### 5.1 Experiments using 32-bit machines

#### 5.1.1 Experimental Setup

The performance of query execution was measured with a simple Java program using the Java Database Connectivity (JDBC) interface. The database for this test has one table. An index is built on its primary key field of type BIGINT. Seven experiments were conducted with varying record sizes (512 bytes, 1kB, 2kB, 4kB, 8kB, 16kB, and 32kB). The following queries were tested:

- Insert a record
- Retrieve a record with a given primary key and return all columns of the selected record
- Update a record matching a given primary key
- Delete a record matching a given primary key

To compare the performance of our system with that of a system with sufficient memory, we divide the experimental setting into *Memory Not Shared* and *Memory Shared.* In the *Memory Not Shared* environment, memories are not shared between machines, i.e., the front-end node can use only the 4GB of RAM installed locally. On the other hand, in the *Memory Shared* environment, memories are shared between the front-end node and back-end nodes, i.e.,the front-end node can access 3GB of RAM owned by the back-end nodes as well as 1GB of local RAM. Linux 2.4.30 is modified to share memory between nodes.

Figure 5 shows the experimental environment of *Memory Shared.* Three 32-bit machines are used to execute MySQL Cluster, and one 32-bit machine executes client programs. Core components of MySQL Cluster are installed on the front-end machine (MySQL Server), which is equipped with 1GB RAM,

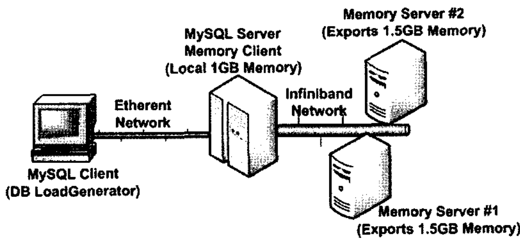



Figure 5 Experimental Setup for 32-bit machines

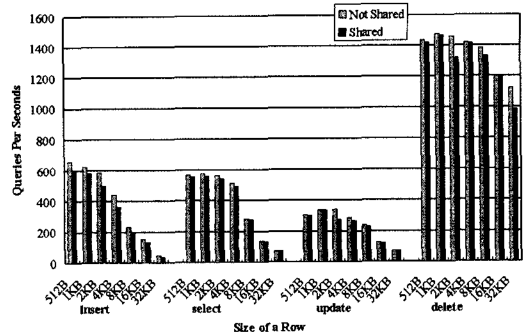



Figure 6 Results of the Micro Benchmark

two hyperthreading-enabled Xeon 3.0GHz CPUs, and one 80GB IDE disk. Two back-end machines play the role of memory servers, which export memory to the MySQL Server node. These machines have 3GB of memory and one hyperthreading-enabled Xeon 3.2GHz CPU. The memory servers export 1.5GB of 3GB memory to the MySQL Server. The machine that executes MySQL client programs has the same specifications as the memory servers. The MySQL Server, which functions as a memory client, is connected to the memory servers by an Infiniband network, which provides 10Gbps bandwidth, while the MySQL Client is connected to the MySQL Server by 100Mbps Ethernet.

To measure the impact on the Online Transaction Processing (OLTP) performance of the database, another experiment is conducted using BenchmarkSQL [12], a TPC-C-like benchmark with a Java Swing/ JDBC client. The number of warehouses was set to 15, which required 1.4GB of memory storage. The number of client sessions was set to 5. This experiment was also performed in both the *Memory Not Shared* environment and the *Memory Shared* environment.

5.1.2 Results

Figure 6 shows the query processing performance when memory is shared and when it is not shared. Five clients create JDBC connections, then generate and request queries. Values in the figures represent the number of queries that each client requested to process. As shown, *Memory Not Shared* outperforms *Memory Shared* because the MySQL Server node has plentiful memory. In the case of *Memory Shared*, if data related to requested queries do not exist in the local memory, the kernel must fetch the data from the remote memory to process the queries. This is the main overhead of our system, but we think that this overhead is relatively small.

Table 1 shows the performance ratio between *Memory Not Shared* and *Memory Shared*. Regardless of the size of rows, the main memory database management system built on our system executes a minimum of 80% of the queries executed by its counterpart system. This implies that although the amount of remote memory is three times larger than that of local memory, our system can guarantee 80% of the maximum performance. Main memory database management system built on Infiniband network and general PCs is a considerable configuration. Values in the table greater than 95% imply that *mysqld* or the Ethernet network between the MySQL Server and Client is the bottleneck in processing queries.

To measure the performance of OLTP, BenchmarkSQL is executed using same configuration as the prior experiment. To analyze the impact of our system on the performance of storage, we use the following three types of storage engines:

• *NDBCLUSTER* enables clustering of in-memory

Table 1 Performance Ratio (Memory Shared vs. Memory Not Shared)

| | 512Byte | 1KB | 2KB | 4KB | 8KB | 16KB | 32KB |
|---|---|---|---|---|---|---|---|
| Insert Query | 91.581% | 93.701% | 85.500% | 82.654% | 85.280% | 87.077% | 81.590% |
| Select Query | 97.511% | 96.570% | 96.821% | 96.202% | 97.377% | 96.380% | 99.931% |
| Update Query | 98.223% | 99.319% | 79.642% | 92.812% | 97.629% | 96.812% | 98.947% |
| Delete Query | 98.826% | 99.546% | 90.492% | 99.697% | 96.451% | 99.858% | 87.892% |

databases

- *MyISAM* is the default storage engine in MySQL for disk-based relational databases
- *HEAP* creates tables with contents that are stored in memory

The difference between the *NDBCLUSTER* and *HEAP* engines is the process that provides memory for storage. *NDBCLUSTER* uses the memory of *ndbd*, while *HEAP* uses that of *mysqld*.

Table 2 shows the average tpmC values measured after BenchmarkSQL is executed for 20 minutes. In the case of the *NDBCLUSTER* engine, the ratio of the measured tpmC values is about 90.5%, which is the result when the storage for 1.4GB database is split into 700MB of local memory and 700MB of remote memory. This means that when the percentage of the local memory is the same as that of remote memory, there is only a 10% performance degradation from full usage of sufficient local memory. This also shows that the *NDBCLUSTER* built on our system outperforms the *MyISAM* regardless of its base system. In the case of the *MyISAM* storage engine, *Memory Not Shared* outperforms *Memory Shared* because the amount of memory for read or sort is configured largely. The performance of *HEAP* storage engine falls behind that of *NDBCLUSTER* because it is used not for high-performance main memory databases but for temporary tables.

Figure 7 shows the amount of memory that pages in or out during the execution of BenchmarkSQL. An average of 10MB of data is paged in and out, with a maximum value of 16MB. This explains that when the main memory database management system uses the same amount of local and remote memory, the required network bandwidth is 1% of the storage space during the

Table 2 Results of the TPC-C-like benchmark using BenchmarkSQL

| Storage Engine | Measured tpmC (Memory Not Shared) | Measured tpmC (Memory Shared) | Ratio |
|---|---|---|---|
| NDBCLUSTER | 2369 | 2145 | 90.5% |
| MyISAM | 1080 | 818 | 75.7% |
| HEAP | 58 | 49 | 84.4% |

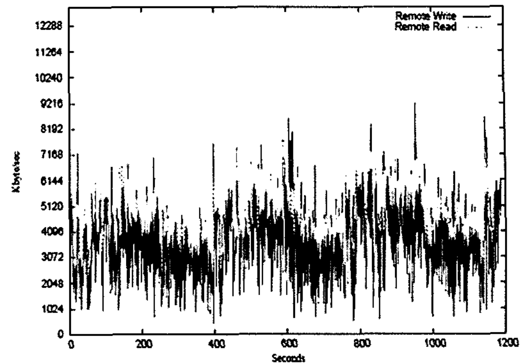



Figure 7 Trace of Memory Page In/Out During the Execution of the BenchmarkSQL

execution of the TPC-C workload. We can estimate the scalability of our system with this result, which will be discussed in the next section.

### 5.2 Experiments using 64-bit machines

This section describes experiments that test scalability using 64-bit machines. Since main memory database management systems built on 32-bit machines lack address space, it is not appropriate for large databases. Therefore, it is very important to build main memory database on 64-bit machines and 64-bit OSes.

#### 5.2.1 Experimental Setup

Figure 8 shows the configuration of the *Memory Shared* environment. The front-end node is equipped with two 1.8GHz Dual Core AMD Opteron(tm) Processor 265 CPU, 2GB RAM, and one 250GB SATA disk. MySQL Cluster software is installed in this node. The six back-end nodes have same hardware specification as the back-end nodes in Section 4.1.1. The only difference is that back-end nodes export 2GB RAM to the front-end node in this experiment. The front-end node runs a modified Linux-2.6.13 to use the remote memory, and the back-end nodes run a modified Linux-2.4.30 to export local memory. In the *Memory Not Shared* environment, the front-end node is equipped with 8GB RAM and runs an unmodified Linux-2.6.13.

#### 5.2.2 Results

To measure the scalability of OLTP performance, four experiments were conducted with varying storage sizes. To increase the need for storage size,

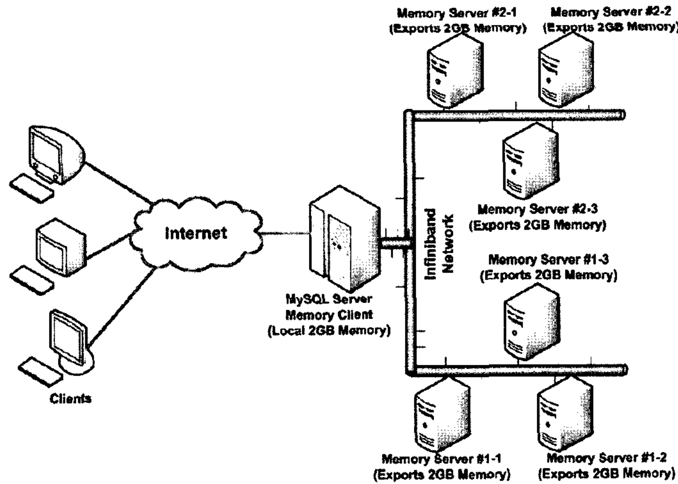



Figure 8 Experimental Setup for 64bit/32bit machines

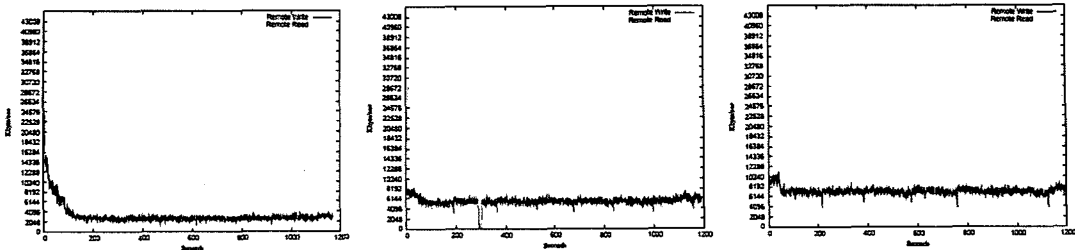
our experiments set to large number of warehouses. This is appropriate to test scalability because usage of the remote memory is raised. The front-end node has 2GB of local memory, and the database takes up 1.8GB of the local memory. When the number of warehouses are set to 50 and 75, the amount of remote memory are 6.0GB and 9.0GB, respectively. Unfortunately, since the front-end node has only 8GB of physical memory in the *Memory Not Shared* environment and this amount is less than the amount needed in the case of 75 warehouses, we could not perform the test. Therefore, we only compare the values measured in the *Memory Not Shared* environment and 50 warehouses.

Table 3 show the performance ratio. Since less remote memory used means better performance, the measured tpmC is best when the number of warehouses is smallest and the environment is *Memory Not Shared*. In both *Memory Shared* environments, OLTP performance is at least 87% of that in the *Memory Not Shared* environment. This result implies that *ndbd* of MySQL Cluster software has a locality of data access pattern, and that our system exploits the locality. In other words, frequently accessed data such as indices are always in the local memory, while less frequently accessed data such as records are transmitted to the remote node. We measured tpmC using the *MyISAM* engine in the Memory Not Shared environment, and the measured value is between 350 and 400. This means that the main memory database built on our system outperforms disk-based databases by a factor of 6 to 8.

Figure 9 shows the amount of memory that pages in or out during the execution of BenchmarkSQL. Clearly, more remote memory for data storage means that more data are transmitted between

Table 3 Remote Memory Usage and Performance Ratio. (It is noted that all results of MyISAM and performed under the Memory Not Shared environment

| # of warehouses | 50 (Memory Not Shared) | 50 (Memory Shared) | 50 (Memory Shared) | 50 (Memory Shared) |
|---|---|---|---|---|
| measured tpmC (NDBCLUSTER) | 3163 | 2940 | 2726 | 2610 |
| measured tpmC (MyISAM) | | 352 | 378 | 393 |
| Amount of remote memory | | 6.4GB | 8.4GB | 10.6GB |
| Performance ratio | | 92% | 87% | 82% |



(a) 50 warehouses (b) 75 warehouses (c) 100 warehouses

Figure 9 Trace of Memory Page In/Out with Varying the Number of Warehouses

client and server nodes. Therefore, it is natural that an average of 14MB and 24MB of data are paged in and out in the case of 50 and 75 warehouses, respectively. Although 1.8GB storage is needed when the number of warehouses is increased from 50 to 75, network bandwidth of only 10MB is required. This also implies that spatial and temporal locality storage allow the required bandwidth to be minimized.

Table 4 shows the results of the experiment where the number of clients, not the number of warehouses, were varied to show that our system supports increases in various categories. The results show that transaction processing becomes saturated before the number of clients reaches 15. This behavior is similar to that displayed in the *Memory Not Shared* environment. A trace of the memory page in/out while varying the number of clients is displayed in Figure 1.

Table 4 Results of the TPC-C-like benchmark (# of warehouses = 50, *Memory Shared*)

| # of clients | 5 | 10 | 15 | 20 |
|---|---|---|---|---|
| measured tpmC | 2940 | 3770 | 4095 | 3920 |

## 6. Future Work

Although the present article offers an initial contribution to the architecture of operating system concerning large memory database processing, more research is needed to enhance the performance of the in-memory database. What remains to be determined by future research is how the existing operating system should be redesigned to support such a large memory database perfectly. Having seen only a small part of the kernel in this article, there is still much to delve into to design a genuine architecture for the database operating system.

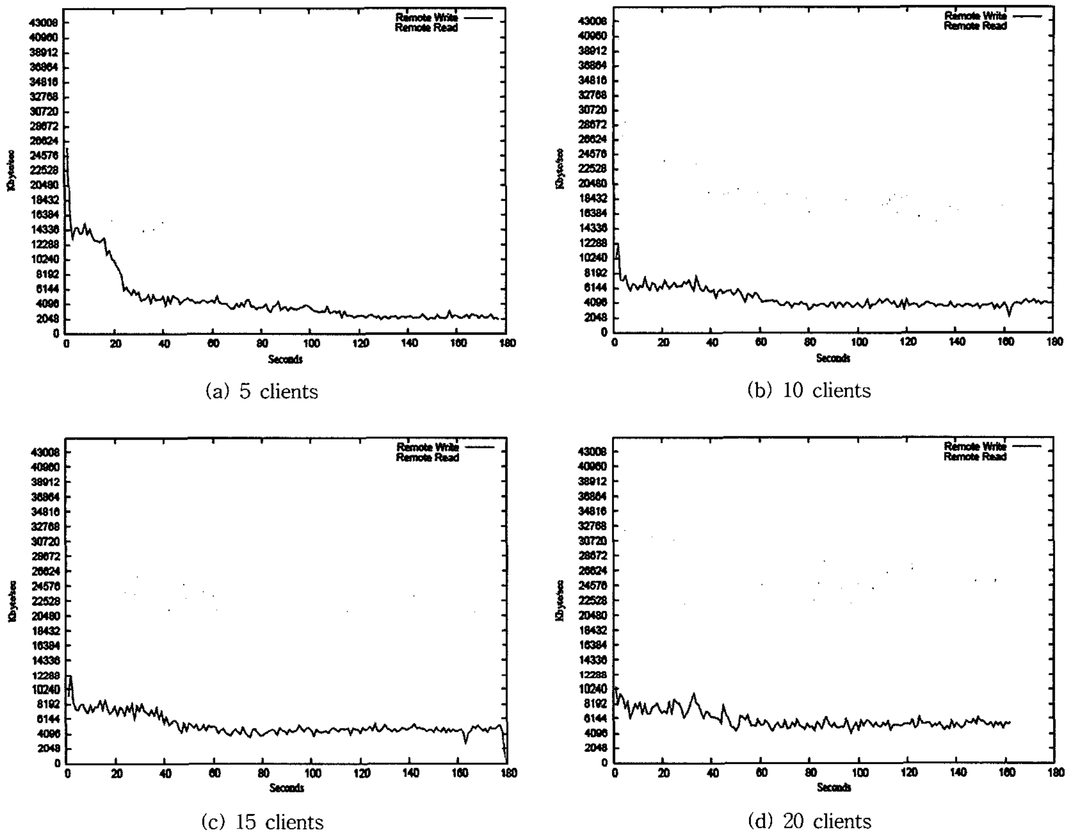
Another issue to be considered is to utilize other high performance commodity hardware to improve or to encompass the limitation of exploiting only the given resources. There are quite many cutting-edge technologies to use in many industries. Therefore, it should be viewed with a more open mind to utilize them.

## 7. Conclusion

This work presents the architecture of the remote memory system and indicates that the architectural change of existing OS using high-performance commodity hardware makes it possible to build very large in-memory database systems. Even though, tentative results of this article leaves more to be researched and analyzed to commercialize high-performance in-memory databases, it is obvious to see the potential of the proposed architecture. The conclusion which can be drawn from the results of this article are these: (1) general purpose operating systems are not suitable to be used in large memory database systems, (2) using high performance commodity hardware, it is possible to materialize large memory database systems in an efficient manner, and (3) there is much enhancements to be made in current operating system architectures to improve the in-memory database system.

## References

[1] D. Black, A. Gupta, and W-D Weber, "Competitive Management of Distributed Shared Memory," In Sprint COMPCON 89 Digest of Papers, Febu-




(a) 5 clients (b) 10 clients

(c) 15 clients (d) 20 clients

Figure 10 Trace of Memory Page In/Out with Varying Number of Clients (# of Warehouses = 50)

rary 1989.

[2] W. Bolosky, M. Scott, and R. Fitzgerald, "Simple but Effective Techniques for NUMA Memory Management," In Proceedings of ACM Symposium on Operating System Principle, December 1989.

[3] M. Holliday, "Reference History, Page Size, and Migration Daemons in Local/Remote Architectures," In Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems, pages 104-112, April 1989.

[4] W. Bolosky, M. Scott, and R. Fitzgerald, R. Fowler, and A. Cox, "NUMA Policies and their Relationship to Memory Architecture," In Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems, pages 212-221, April 1991.

[5] Michael J. Freeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, "Implementing Global Memory Management in a Workstation Cluster," In Proceedings of ACM Symposium on Operating Systems Principles, December 1995.

[6] D. Comer and J. Griffoen, "A new design for distributed systems: The remote memory model," In Proceedings of the Summer 1990 USENIX Conference, June 1990.

[7] M. J. Frankling, M. J. Carey, and M. Livny, "Global memory management in client-server DBMS architectures," In proceedings of the 18th VLDB Conference, August 1992.

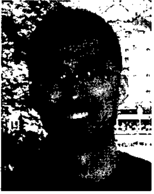
[8] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Paterson, "Cooperative caching: Using remote client memory to improve ¯le system performance," In Proceedings of the USENIX Conference on Operating Systems Design and Implementation, November 1994.

[9] D. P. Bovet and M. Cesati, "Understanding the Linux Kernel," O'REILLY, 2003.

[10] M. Gorman, "Understanding the Linux Virtual Memory Manager," Prentice Hall, 2004.

[11] MySQL AB, "MySQL," http://www.mysql.com.

[12] PostgreeSQL Development Group, BenchmarkSQL, http://pgfoundry.org/projects/benchmarksql.



정 형 수

2002년 고려대학교 기계공학과 학사
2004년 서울대학교 컴퓨터공학부 석사
2004년~현재 서울대학교 컴퓨터공학부 박사과정. 관심분야는 분산시스템, 알고리즘, 데이타베이스 시스템

한 혁

2003년 서울대학교 컴퓨터공학부 학사
2006년 서울대학교 컴퓨터공학부 석사
2006년~현재 서울대학교 컴퓨터공학부 박사과정. 관심분야는 분산시스템, 결함내성시스템, 운영체제, 알고리즘

염 헌 영

1984년 서울대학교 계산통계학과 학사
1986년 Texas A&M Univ. 전산학 석사
1992년 Texas A&M Univ. 전산학 박사
1993년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 분산시스템, 결함내성시스템, 운영체제, 데이타베이스 시스템