DLM: A Distributed Large Memory System using Remote Memory Swapping over Cluster Nodes

Hiroko Midorikawa #1#2, Motoyoshi Kurokawa #3, Ryutaro Himeno #4 , Mitsuhisa Sato #2

#1 *Department of Computer and Information Science, Seikei University*
3-3-1 Kichijouji Kita-machi, Musashino-shi, Tokyo, 180-8633, Japan
1 `midori@st.seikei.ac.jp`

#2 *Graduate School of Systems and Information Engineering, University of Tsukuba*
Tennodai, Tsukuba, 305-8577, Ibaraki, Japan

#3 *Advanced Center for Computing and Communication,* #4 *Research Program for Computational Science, RIKEN*
2-1 Hirosawa, Wako-shi, Saitama, 352-0198, Japan

Abstract—**Emerging 64bitOS's supply a huge amount of memory address space that is essential for new applications using very large data. It is expected that the memory in connected nodes can be used to store swapped pages efficiently, especially in a dedicated cluster which has a high-speed network such as 10GbE and Infiniband. In this paper, we propose the Distributed Large Memory System (DLM), which provides very large virtual memory by using remote memory distributed over the nodes in a cluster. The performance of DLM programs using remote memory is compared to ordinary programs using local memory. The results of STREAM, NPB and Himeno benchmarks show that the DLM achieves better performance than other remote paging schemes using a block swap device to access remote memory. In addition to performance, DLM offers the advantages of easy availability and high portability, because it is a user-level software without the need for special hardware. To obtain high performance, the DLM can tune its parameters independently from kernel swap parameters. We also found that DLM's independence of kernel swapping provides more stable behavior.**

I. INTRODUCTION

Emerging 64bitOS's can supply a huge amount of memory address space that is essential for new applications using very large data. Large memory space is beneficial in applications such as databases and bioinformatics. In the current x86_64 architecture, up to 256 terabytes of address space is available. However, in a conventional virtual memory system, a page is swapped off or on a disk when the size of used memory exceeds that of real memory.

Recently, the bandwidth of network devices has become larger than that of hard disks. It is expected that the memory in other nodes can be used to manage swapped pages efficiently, especially in a dedicated cluster which has a high-speed network such as 10GbE and Infiniband.

In this paper, we propose Distributed Large Memory System (DLM), which enables users to make use of very large virtual memory by using remote memory distributed over the nodes in a cluster.

The DLM system outperforms ordinary kernel swap systems using a local hard disk as the swap device. According to our experiments [1], we found that DLM performance of a 10 Gbps Ethernet cluster is about ten times better than that of a conventional kernel swap system, when it has 77 GB of virtual memory and the remote data/local data size ratio is 15%. Even on 1 Gbps Ethernet clusters, DLM performance is more than five times better than that of a conventional kernel swap system when the remote/local data size ratio is 2 to 5. Even though DLM performance depends on the remote/local memory size ratio, it is never lower than the performance of a conventional kernel swap system.

We also found the behavior of DLM is generally more stable than that of a conventional kernel swapping system. With the kernel swapping system, performance fluctuations for the same program are 2% to 60%, even if a cluster was dedicated. In contrast, the DLM performance fluctuations were less than 1%.

There are several related studies on using remote memory over cluster nodes for swapping [2][3][4]. In most research, a new block device driver is used to access remote memory and to replace the traditional swap device, which is typically a local hard disk. This scheme has the advantage of full transparency to users even when using remote memory, but several papers reported stability problems due to the lack of local memory when the kernel swap daemon accessed remote memory. Some studies were not very successful in obtaining sufficient performance with specially designed NIC hardware, protocols, and kernel modification.

The DLM system is a user-level software with no need for any special hardware or kernel modification. It also operates independently from the OS swap system. So, setting DLM system parameters is completely separate from kernel parameters, which in turn gains the highest performance of the processors and network. In this paper, the performance of application programs using remote memory on the DLM system is compared to the performance of programs using local memory in a 10 Gbps Ethernet cluster. Our contributions are summarized as follows:

- Re-evaluation of the scheme using user-level software for remote memory swapping instead of low-level schemes, e.g., the block device driver scheme.
- Design of totally new software for large-data-use sequential programs by using recent thread technology.

- Demonstration of DLM performance, which is better than that of other schemes in high-memory-access applications using only conventional TCP on 10Gbps Ethernet.
- Development of DLM independence of the conventional kernel swapping, thus providing more stable behavior than that using the kernel swapping device.
- Use of a larger page size than 64 KB, which results in better performance in application programs.

II. OVERVIEW OF DLM SYSTEM

The Distributed Large Memory System (DLM) is a system which provides very large virtual memory by using remote memory distributed over the nodes in a cluster.

A. Runtime System of DLM

The runtime system consists of one process on a local host and one or more remote processes on memory server hosts, which are automatically forked by DLM system initialization after the user inputs a program execution command. Fig. 1 shows a DLM runtime system and some command examples. The local process includes a calculation thread, *cal thread*, which is the thread invoked by a user command, and a communication thread, *com thread*, which is automatically created during DLM system initialization. This system is designed for a sequential program using large data, so user program codes are executed only in a local host.

The user-specified large data, called *DLM data*, are allocated not only in local host memory but also in remote memory on memory server hosts when the amount of local host memory is not sufficient for the *DLM data*. The *com thread* communicates between the remote memory server processes when the local *cal thread* requires *DLM data* that are partially or fully allocated in remote host memory. Data swapping between local and remote memory is done in units of *DLM pagesize*, which is a multiple of the *pagesize* defined in the OS kernel.

The automatically forked remote processes and the local *com thread* are finalized when *cal thread* code execution is finished.

The *DLM hosts file* includes a list of the host names and amount of memory size on each host available for the DLM system. Fig.1 shows a user-defined *DLM hosts file* called hostfile in this example. The hostfile includes a calculation local host name, e.g., calhost, in the first line and multiple memory server remote host names, e.g., memhost1, in the following lines.

B. Application Interface for DLM Data

A *DLM program*, which is a C program for the DLM system, is almost the same as an ordinary sequential C program, except for the replacement of malloc() with *dlm_alloc()* and attaching of *dlm* to static data declaration statements. Fig. 2 shows the API for *DLM data* static declaration and dynamic allocation in the DLM sample program for a matrix and a vector multiplication. Only user-specified *DLM data* has a chance to be placed in remote memory when the local memory does not have sufficient space. Unspecified data are guaranteed to be in local memory.

C. DLM Compiler and DLM Library

The compiler, *DLM compiler*, easily enables the use of remote memory with ordinary C programs. The compiler consists of two components: a DLM to C program translator, and a general C compiler, e.g., gcc. In the first phase, a DLM program is converted to a normal C program that includes DLM library functions. In the second phase, gcc compiles the translated C program and links it with the DLM library, *libdlm*, to create an execution file. Fig.3 shows an example before and after the first translation phase.

The DLM initialization function, *dlm_init()*, forks memory server processes in each remote host specified in *DLM hostfile* and creates a communication thread in the local host. Then it establishes sockets (TCP or UDP) and initializes a *DLM pagetable*. The finalization function, *dlm_exit()* waits for the end of the local *cal thread* calculation and then finalizes remote memory server processes and sockets.

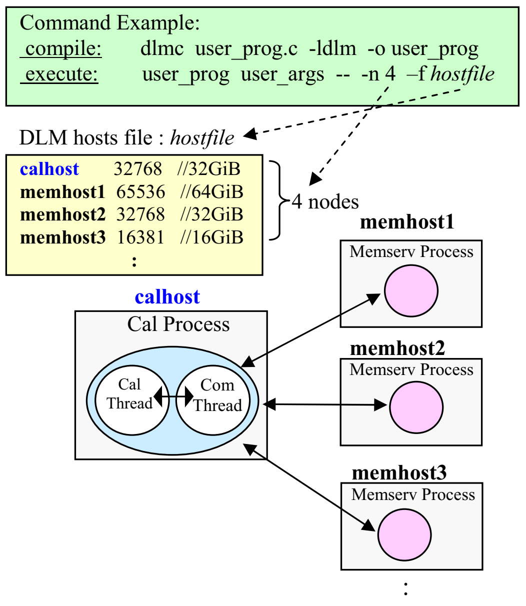
The DLM system allocates *DLM data* dynamically in run time, so the *DLM compiler* translates static *DLM data* declarations to pointer-based dynamic dlm allocations and renames all static *DLM data* array access expressions to pointer-based access expressions, as shown in Fig. 3.

In other words, the *DLM compiler* automatically translates a sequential user program to an implicitly parallel processing program which consists of one local calculation process executing user program code and multiple remote memory server processes. *DLM compiler* provides a user-transparent environment for using remote memory over cluster nodes.

III. IMPLEMENTATION OF DLM SYSTEM

A. DLM System Initialization and DLM Pagetable

DLM data are managed with *DLM pages* and a *DLM pagetable*. Each entry of the *DLM pagetable* has a host ID



Command Example:

```
compile:   dlmc user_prog.c -ldlm -o user_prog
execute:   user_prog user_args -- -n 4 –f hostfile
```

DLM hosts file : *hostfile*

```
calhost   32768  //32GiB
memhost1  65536  //64GiB
memhost2  32768  //32GiB
memhost3  16381  //16GiB
   :
```

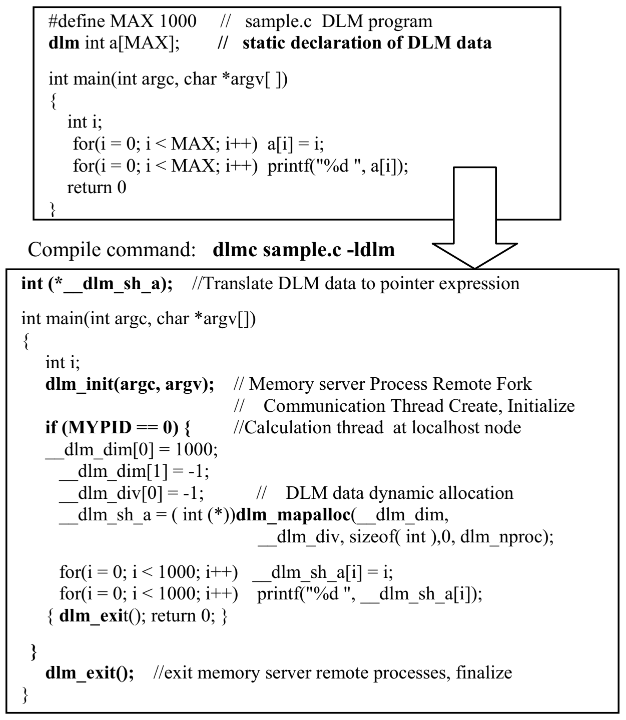
Fig. 1. DLM Runtime System, DLM hosts file and a command example

```
//DLM Program example : Matrix Vector Multiply
#include <stdio.h>
#define N 100000    //N:100K a: 80GB    x::800KB
dlm double a[N][N], x[N];   //DLM static declare

main(int argc, char *argv[])
{   int i,j;
   double *y;      // DLM dynamic alloc y:800KB
   y = (double *) dlm_alloc ( sizeof(double) *N );

   for(i = 0; i < N; i++)    //    Initialize a
      for(j = 0; j <N; j++)    a[i][j] = i;
   for(i = 0; i < N; i++)    x[i] = i;   // Initialize x
   for(i = 0; i < N; i++){        // multiply
        y[i]=0;                    // initialize y
        for(j = 0; j <N; j++)    y[i] += a[i][j]*x[j];
   }
   return 0;
}
```

Fig. 2. The API for DLM data in a sample program



```
#define MAX 1000     //   sample.c  DLM program
dlm int a[MAX];      //   static declaration of DLM data

int main(int argc, char *argv[ ])
{
   int i;
    for(i = 0; i < MAX; i++)  a[i] = i;
    for(i = 0; i < MAX; i++)  printf("%d ", a[i]);
   return 0
}
```

Compile command: **dlmc sample.c -ldlm**

```
int (*__dlm_sh_a);    //Translate DLM data to pointer expression

int main(int argc, char *argv[])
{
   int i;
   dlm_init(argc, argv);   // Memory server Process Remote Fork
                           //   Communication Thread Create, Initialize
   if (MYPID == 0) {       //Calculation thread  at localhost node
   __dlm_dim[0] = 1000;
     __dlm_dim[1] = -1;
     __dlm_div[0] = -1;          //   DLM data dynamic allocation
     __dlm_sh_a = ( int (*))dlm_mapalloc(__dlm_dim,
                                 __dlm_div, sizeof( int ),0, dlm_nproc);

     for(i = 0; i < 1000; i++)   __dlm_sh_a[i] = i;
     for(i = 0; i < 1000; i++)   printf("%d ", __dlm_sh_a[i]);
   { dlm_exit(); return 0; }

 }
   dlm_exit();   //exit memory server remote processes, finalize
}
```

Fig. 3. Before-After Translation example in DLM compiler

of where the page is allocated, the page top address and the end address of the data allocation, etc. The DLM system initialization creates remote memory server processes, the local *com thread* and sockets between them, and it allocates *DLM pagetable* with sufficient page entries for the total amount of memory size described in *DLM hosts file*. The *DLM pagetable* itself is not swapped out to memory servers.

B. DLM Data Allocation and DLM Hosts file

When *DLM data* are allocated, the necessary number of *DLM page*s is mapped in local memory first, up to the specified memory size in the first line of *DLM hosts file*. If local memory is not sufficient for mapping all *DLM data*, the remaining data are mapped in remote memory of a remote server host in the next line of the file. If the server host memory for the current allocation is not sufficient, the remains of the *DLM data* are allocated in the next line stating a memory server in the file. In this way, *DLM data* are allocated in each memory server host according to priority, determined by the sequence of descriptions in *DLM hosts file*.

C. DLM Page Swapping

When a user program accesses an unmapped non-local *DLM data* area, the SIGSEGV signal handler is invoked to retrieve a required *DLM page* from the memory server which possesses it. The handler also swaps a local *DLM page* and a remote *DLM page*, if necessary. The selection policy of the swap-out page is simplified to reduce the overhead for evaluating the importance of pages. The current implementation adopts ”First allocated page is first swap out” policy. It also can be expanded to a more sophisticated policy with kernel page information.

The difference between kernel swap and DLM swap processing is that the kernel swap causes file accesses on the swap device, which is typically a local hard disk. The DLM system unmaps a swap-out page from the local memory, sends it to the remote host, gets the required remote memory page and directly maps it to the local memory without file accesses.

The DLM system is independent from kernel swap processing, so all parameters used in the system (e.g., page size) for swapping, are separate from kernel parameters and can be set with values that gain the highest performance according to the processor and communication link characteristics.

D. Communication in the DLM System

The *com thread* in a local process communicates to both the local *cal thread* and remote memory server processes, and so it waits for two kinds of signals, the SIGIO from the external process and the SIGUSR from the internal *cal thread*. Conversely, each remote memory server process is a dedicated process for a single client, the cal process. So, it is implemented as a simple loop server process waiting for a client request.

There are no communication links between remote processes. So, when an error occurs in a remote process, it informs the *com thread* in the cal process, and the *com thread* broadcasts it to other remote processes and all processes are finalized. In normal finalization, the *com thread* waits for the end of the *cal thread* calculation and after that, it notifies the normal end to remote processes, and all sockets are closed.

IV. DLM PERFORMANCE OF 10 GBPS ETHERNET CLUSTER

A. Experimental Setting

In this study, the bandwidth of remote memory access in a DLM system is measured and the performance of DLM programs partially using remote memory is compared to conventional programs using only local memory (the local/total

TABLE I
10Gbps Ethernet Cluster (RIKEN CSLM)

Cluster	HP DL585 G2 x 5 Nodes
Node CPU	DualCore AMD Opteron(8220SE) 2.8GHz x 4 (8Cores)
Node Memory	64GiByte(67.1GB)
OS	Linux kernel 2.6.9-42 x86_64
Compiler	gcc version 3.4.6
Network	10GbE protocol (Myri-10G)
Switch	Fujitsu XG1200(10GbE Switch)
Hard Disk	SAS 147GB 10krpm 2 , RAID1 Smart array 5i, HP 431958-B21 (TransRate 300MBps, seektime 4(Ave)8(max)ms)

TABLE II
DLM PAGE SIZE VERSUS WRITE TIME WITH SWAP OVERHEAD BETWEEN REMOTE MEMORY AND LOCAL MEMORY

DLM Page Size(KB)	Write Time(usec)		Relative Time	
	1GbE	10GbE	1GbE	10GbE
4	311	103	1.00	1.00
8	369	116	1.18	1.12
16	437	171	1.40	1.65
32	559	182	1.80	1.76
64	885	245	2.84	2.36
128	1505	395	4.83	3.81
256	3179	726	10.21	6.99
512	7571	1461	24.31	14.06
1024	16495	3106	52.96	29.89

data size ratio is 100%). Table I shows the cluster used in this experiment. In this environment, the DLM system uses TCP over Myri-10G [6]. The performance of TCP on Myri-10G is comparable or better than that of UDP on Myri-10G. It should be noted here that all measured applications runs sequentially in one calhost and other nodes serve as memhosts.

B. Basic Performance of Micro Benchmarks

1) DLM Pagesize and Overhead of Remote Memory Swap: To investigate the influence of *DLM page* size on swap overhead, one write time accompanied by one page swapping is measured. Table II shows the integer write time for each *DLM pagesize* both on 1 Gbps and 10 Gbps Ethernet clusters. The table shows that the values for a page size of 1024 KB are 53 and 30 times slower than the values for a page of 4 KB in each case. These are not small values, so it means that a larger page size is not always good for all, even though a larger page size usually gives a data preload effect, depending on the data access locality of each application.

2) STREAM Benchmark measuring Remote Memory Bandwidth: The remote memory access bandwidth in the DLM system is measured by using STREAM benchmarks. STREAM [5] is a set of multiple kernel operations, that is, sequential accesses over array data with simple arithmetic, as shown in Table III. STREAM outputs sustainable memory bandwidth

TABLE III
STREAM BENCH MARK

	Kernel	Code
STREAM	COPY	a(i) = b(i)
	SCALE	a(i) = q*b(i)
	ADD	a(i) = b(i) + c(i)
	TRIAD	a(i) = b(i) + q*c(i)

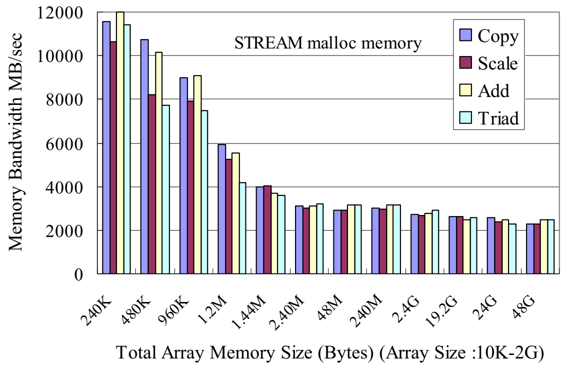


Fig. 4. Local Memory Bandwidth (STREAM) Array Size 10K-2G Local memory/Total memory Size Ratio 100%

at the application level for each kernel after iterative measurements. It is important for STREAM to choose an array size which is sufficiently large to ignore the cache effect. So, we first measured local memory bandwidth with the various array sizes shown in Fig. 4. Although STREAM originally uses static array declaration, it was modified to use dynamic data allocation because of the limitation of the static data section size in gcc. The result shows that an array size larger than 2.4 MB is acceptable for measurement. We chose 100 M elements and a 2.4 GB array size for comparison between the remote and local memory bandwidth. The local memory bandwidth of this size is around 3 GB/s, shown in Table IV. Static data allocation obtains a slightly better performance. To obtain more objective results, the local memory bandwidth of dynamic allocation is used for comparison with remote memory bandwidth in DLM, because the DLM System allocates data dynamically.

The measured remote memory bandwidth is shown in Fig. 5. The larger page size shows a better performance. The best value, 380 MB/s for page size 1024 KB, is much better than the bandwidth in the Remote Direct Memory Access (RDMA) based block device scheme [2]. This value is also comparable to 375 MB/s, which is a theoretical maximum IO performance for typical storages, SATA300 and SAS. The actual performance of storages is more degraded by additional

TABLE IV
LOCAL MEMORY BANDWIDTH(MB/S) ARRAY SIZE 100M

STREAM Array Size:100M 2.4GByte

	COPY	SCALE	ADD	TRIAD
static	2976	2804	2926	3153
malloc	2718	2694	2767	2925

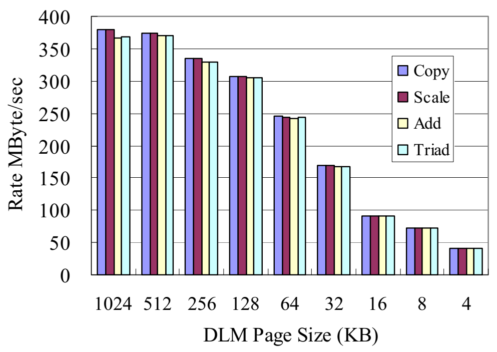


Fig. 5. Remote Memory Bandwidth (STREAM) Array Size 100M(2,4GB) Local memory/Total memory Size Ratio 8%

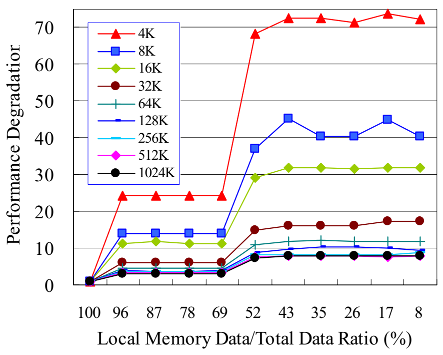


Fig. 7. STREAM TRIAD Performance Degradation, Array Size 100M(2,4GB)

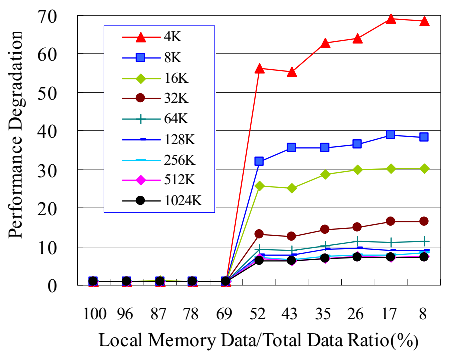


Fig. 6. STREAM COPY Performance Degradation, Array Size 100M(2,4GB)

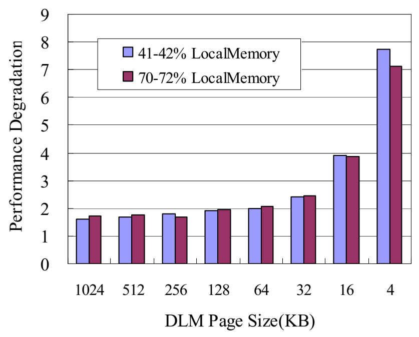


Fig. 8. NPB FT.B Performance Degradation

hardware overhead, e.g., the seek time, and software overhead, especially in random accesses.

Fig. 6 and 7 show the relative execution times of COPY and TRIAD, respectively, to the execution time using only local memory for various local/total data size ratios and *DLM pagesize*s. In COPY, only two arrays among three are used, as shown in Table III, so there is no degradation when there is more than 60% - 70% of data is in local memory. On the other hand, TRIAD degradation begins in the lower local/total data size ratio because it uses three arrays in the kernel. When using a small page size, 4 KB - 16 KB, all kernels in STREAM and STREAM2 showed 50-70 times degraded performance compared to that of the local memory execution. Using a larger page size than 64 KB decreases degradation by 10 times, even if only 8% of the total data is in local memory.

C. Performance of Application Benchmark

The performance in more real field applications than the above micro benchmark is measured with the NAS parallel benchmark (NPB) [7] and Himeno benchmark [9].

1) NAS Parallel Benchmark: The C sequential programs, FT, IS and CG in class B from NPB2.3-omni-C [8], are used for this evaluation. Fig.8 - Fig.10 show the relative execution time to the execution time using only local memory for various *DLM pagesize*s and size ratios of local/total data.

The DLM performance in these programs depends on data access locality and (calculation/memory access) ratio, etc. These three programs are known to be programs with a higher memory load among NPB suites, but they also seem to have more data access locality in comparison with the micro benchmark used above. The performance degradation in these NPB programs is 10 to 20 in the worst cases. This result is relatively smaller than that in STREAM, where the worst degradation is more than 70. The results show the performance for a larger *DLM pagesize* is better even if the data access is not limited to simple sequential access, as in STREAM.

FT.B is a 3-D FFT calculation for a 512x256x256 matrix, which requires three 3-dimensional arrays of a complex number, 1.7 GB. Iterative calculation is performed for 3 different directions of data accesses with discrete gaps, and causes the biggest number of swaps among these programs.

IS.B is an integer sort which requires three integer arrays of 2**25 elements, 384 MB in total. It is a relatively small data size, but the performance changes linearly according to the local/total data size ratio.

CG.B requires 14 different types and sizes of arrays, 510 MB in total. The *DLM pagesize* does not affect the performance when the local/total data size ratio is larger than 30%. Once the local memory becomes lower than approximately 100 MB, the performance drops drastically, as shown in Fig. 10. It is supposed that the working set of data access is near 100 MB. CG often uses an indirect access using an array of indices for data array for nonsequential access. It requires two accesses of memory to get data, so once the local memory

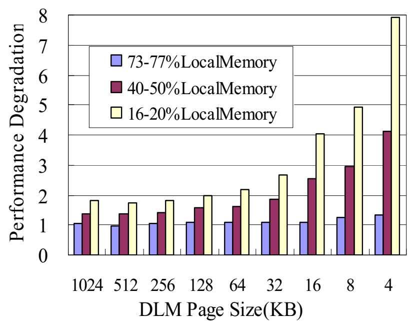


Fig. 9. NPB IS.B Performance Degradation

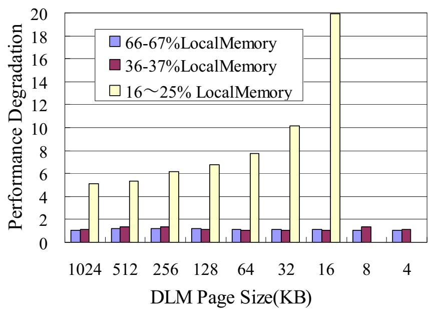


Fig. 10. NPB CG.B Performance Degradation

size ratio becomes low, the performance degradation becomes large.

2) Himeno Benchmark: This benchmark measures the speed of major loops for solving Poisson's equation using the Jacobi iteration method to evaluate the performance of incompressible fluid analysis code [9]. It uses multiple loops of iterations and is known as a program requiring heavy memory access. A large size version, 257x257x513, 1.9 GB, is used. It outputs the performance in MFLOPS. Fig. 11 shows the relative performance degradation to the execution using only local memory over various page sizes and local/total data size ratios. The result shows the worst degradation is approximately 35 at a 4 KB page size and the best is 5 at a 1024 KB page size when the local/total data size ratio is 10%.

This result means the Himeno benchmark has heavier memory access than the NPB programs used here and has lighter access than the micro benchmark, STREAM. When using a larger page size than 32 KB, the degradation is maintained at lower than 10, even at the lowest local/total data size ratio. The performance difference between 64 KB and 1024 KB page sizes is small, so it means using a page size larger than 64 KB is sufficient for good performance in the Himeno benchmark.

Our performance result in the Himeno benchmark is better than other research results that use specially designed NIC hardware for a 10 Gbps Ethernet and a device driver. The Himeno benchmark contains a wide range of data accesses that causes a higher number of swaps than the other programs, qsort, LU or SP in NPB.

Our results show that much larger page sizes than the

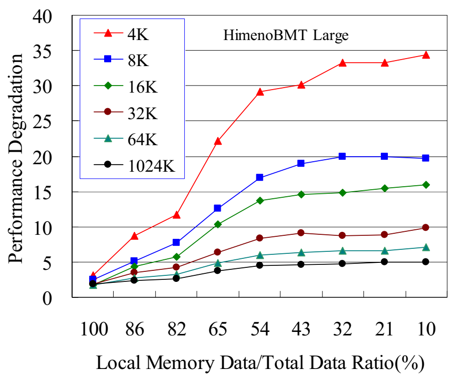


Fig. 11. Himeno Benchmark Performance Degradation

kernel page size, 4 KB, is more effective to obtain higher performance, without special network hardware, protocols, device drivers or kernel modification.

V. CONCLUSION

In this paper, we proposed the Distributed Large Memory System (DLM) which enables us to make use of very large virtual memory by using remote memory distributed over nodes in a cluster.

The results show that DLM using only user-level software achieves better performance than do other low-level remote paging schemes, which typically use a block device for swapping to access remote memory. The stable behavior and higher performance of DLM benefit by DLM's independence of the conventional kernel swapping system. The DLM can tune its parameters independently from the kernel swapping system to obtain higher performance and does not suffer form the unstable behavior of the current kernel swap daemon. It also shows that larger page size is more effective to achieve higher performance.

The advantages of DLM are not only shown in performance but also in easy availability and high portability, because it is a user-level software that needs no special hardware and kernel modification.

REFERENCES

[1] H.Midorikawa, H.Koyama, M.Kurokawa, R.Himeno, ”The Design of Distributed Large Memory System DLM and DLM Compiler”, IEICE Technical Report. Computer systems, Vol.107,No.398, pp. 29-34, 2007 (In Japanese)

[2] S. Liang, R. Noronha, and D. K. Panda,, Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device, IEEE Cluster Computing, Sept. 2005

[3] Pavel Mache, Linux Network Block Device, (1997) [Online]. http://www.xss.co.at/linux/NBD/ http://nbd.sourceforge.net/

[4] Tia Newhall et al. “ Nswap: A Network swapping Module for Linux Clusters ” , EuroPar03, 2003

[5] (2008) STREAM Benchmark web site [Online]. http://www.cs.virginia.edu/stream/ref.html

[6] (2008) Myri-10G, Myricom web site [Online]. http://www.myri.com/

[7] (2008) NPB (NAS Parallel Benchmarks) web site [Online]. http://www.nas.nasa.gov/Resources/Software/npb.html

[8] (2008) NPB2.3-omni-C web site [Online]. http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html

[9] (2008) Himeno Benchmark web site [Online]. http://accc.riken.jp/HPC/HimenoBMT/index.html