Distributed Shared Memory in Kernel Mode

Thobias S. Trevisan Vítor Santos Costa Lauro Whately Claudio L. Amorim
COPPE Systems Engineering Program
Federal University of Rio de Janeiro, Brazil
E-mail: {thobias, vitor, whately, amorim}@cos.ufrj.br

Abstract

In this paper, we introduce MOMEMTO (MOre MEMory Than Others) a new set of kernel mechanisms that allow users to have full control of the distributed shared memory on a cluster of personal computers. In contrast to many existing software DSM systems, MOMEMTO supports efficiently and flexibly global shared-memory allowing applications to address larger memory space than that available in a single node. MOMEMTO has been implemented in the Linux 2.4 kernel and preliminary performance results show that MOMEMTO has low memory management and communication overheads and that it can indeed perform very well for large memory configurations.

1. Introduction

In recent years there has been a great interest in the use of clusters of personal computers as an effective way for applications to achieve high performance at low cost. Nowadays, clusters range from some of the largest machines ever built, such as [5, 6] for large-scale parallel applications, to small systems [4] used at department level. Clusters are widely spread in both academia and commercial world, with applications in many areas such as high-performance computing, database servers, web servers, and scientific visualization.

Although clusters offer scalability regarding to the number of the CPUs as well as memory space they may be limited by node-to-node bandwidth. Thus, writing efficient distributed applications that scale can be a hard problem for cluster computing. We argue that one of the main difficulties in writing such applications is the reliance on off-the-shelf operating systems (OS). Often, each node in the cluster runs its own copy of the operating system, plus some glue code. Also, traditional operating systems were designed for different computer environments and do not take best advantage of cluster capabilities. As a result, Cluster researchers tend to work around this problem, either by working at the application level, as in TreadMarks [8], or by writing new communication drivers that operate on standard interfaces, such as M-VIA [3]. Unfortunately, such solutions may not obtain maximum cluster performance, as they do not address the mismatch between the original OS design goals and specific performance issues that modern clusters raise.

Our work concentrates on the problem of providing a shared memory abstraction while achieving maximum scalable application performance. First, we observe that many parallel applications, ranging from NAS-benchmarks [9] to Database Management systems, often need not only processor but also *memory scalability*. For instance, assume a 16-node cluster where each node has 512 MB, running either a large class-A NAS benchmark or large size database that requires 8 GB of memory. Unfortunately, many current software DSM (Distributed Shared Memory) systems [16, 8, 21, 20, 14, 11] rely on the OS virtual memory manager, and will not take full advantage of the aggregate amount of local memories for data sharing. We argue that an efficient solution to this problem involves adapting the OS virtual memory manager to the cluster.

Our second observation is that DSM systems provide sophisticated and powerful mechanisms to enforce memory consistency at the expenses of increasing system complexity. However, many applications require simpler forms of guaranteeing memory consistency, and could perform better using more appropriate mechanisms. Indeed, in many cases, it may well be worth the effort to implement specific mechanisms to support popular applications, such as database management systems and computer simulations.

Therefore, in this work we propose the following set of kernel-based abstractions for global memory management on a cluster:

- At the lowest level, we extend the OS virtual memory mechanism to encompass the cluster;
- Next, we provide a minimum set of modules to provide basic functionalities that can be implemented efficiently, such as locks, with minimum overheads.

- Lastly, other higher-level modules will support well-known DSM protocols as well as specific protocols oriented to applications such as database servers and web servers.

Our work is motivated by previous research efforts in operating systems. As in nano-kernels [12], we also think that users should have the choice of how to attain maximum performance, and that this is achievable by providing different OS interfaces. The success of open-source operating systems such as Linux and BSD variants led to widespread access to documentation and clearly defined interfaces, which contributed to the internals of kernels becoming available. Thus, many users are comfortable with patching and reconfiguring kernels towards attaining maximum performance. Ultimately, such results simplify the OS optimization tasks and provide an extra motivation to improve operating systems towards maximum application performance.

Next, we present the MOMEMTO system, *MOre MEMory Than Others* software DSM systems, which extends the Linux 2.4 kernel by including a set of mechanisms that support global shared memory. At the lowest level, the system hooks into both the Linux basic VM (Virtual Memory) and networking mechanisms and extends them by providing a very basic distributed shared mechanism at the page level. Next, we provide a set of modules that take advantage of this mechanism to support higher abstractions to the user. Preliminary results show that the MOMEMTO system can achieve good performance when compared with traditional software DSM systems, such as TreadMarks. For instance, MOMEMTO runs applications that require more memory than that Treadmarks can provide.

This paper is organized as follows. In section 2 we present MOMEMTO and its implementation. Section 3 presents our initial tests and discusses the results we obtained. Section 4 presents related work. Lastly, we draw our conclusions and suggest future works.

2. MOMEMTO

2.1. The System

MOMEMTO is designed to allow applications to access and share aggregated cluster memory as a transparent extension of local memory. At its lowest level, MOMEMTO provides a basic sharing mechanism by extending the basic kernel virtual memory mechanism. More complex mechanisms will also work at kernel level, but instead are loaded dynamically as kernel modules.

MOMEMTO works by taking advantage of the Linux VMA (Virtual Memory Area) mechanism. VMAs are contiguous ranges of linear addresses with the same permission flags, or, in other words, homogenous memory regions. Each process has a list of VMAs. Example VMAs include the text VMA, that stores the executable, a VMA for the global variables, and a VMA for the stack. All VMAs in a process are linked together in a list. Each VMA has an array of operations, kept in a structure of type `vm_operations_struct` that handle virtual-memory events, such as page-faults.

In a nutshell, MOMEMTO works by creating and managing its own VMA. A process uses MOMEMTO in much the same as way as it was doing an `mmap()` on anonymous memory. The user requests a contiguous range of addresses, and MOMEMTO creates a new range of addresses that will be inserted in the process' virtual memory. The process will then use MOMEMTO memory in much the same way as if it had performed `mmap()` or `malloc()`.

The MOMEMTO VMA allocates pages on demand. The key function is called at a page fault and is stored at the `nopage` field of `vm_operations_struct`. By default, if the field is `NULL` Linux will allocate an empty page. MOMEMTO instead calls its own code to allocate the page, as we describe next.

Figure 1 shows how MOMEMTO interacts with applications. Each application maps several virtual memory segments into its address space. Linux segments are local to a node. In contrast, MOMEMTO segments are also *shared* among nodes. From the application's point of view, the shared space is viewed as a single contiguous block of memory. From the implementation point of view, shared segments from a node are aggregated together to form the large shared segment. Each node may have mapped some, but not necessarily all, of the pages in the shared segment.

2.2. Design Issues

MOMEMTO includes a lower level module that extends the kernel's Virtual Memory mechanism, plus a collection of modules that implement extra mechanisms, such as synchronization through barriers. MOMEMTO is designed to allow for maximum flexibility by giving users maximum control. This can be achieved either through system calls that control policies or through plugging in different modules that implement different mechanisms.

At its lowest layer, MOMEMTO just shares pages among nodes. MOMEMTO currently uses a home based protocol, where each node is responsible for a fraction of the total number of pages. Each node may also cache pages owned by external nodes. Pages are cached-in at the time of first request.

MOMEMTO allows the programmer to specify the policy for which node will be the home node for a page. One possibility is simply to divide the memory from N nodes in N equally sized segments, say, by using round-robin. Another solution is to organize pages in order to optimize disk access. In this case, we could use a datum's physical loca-

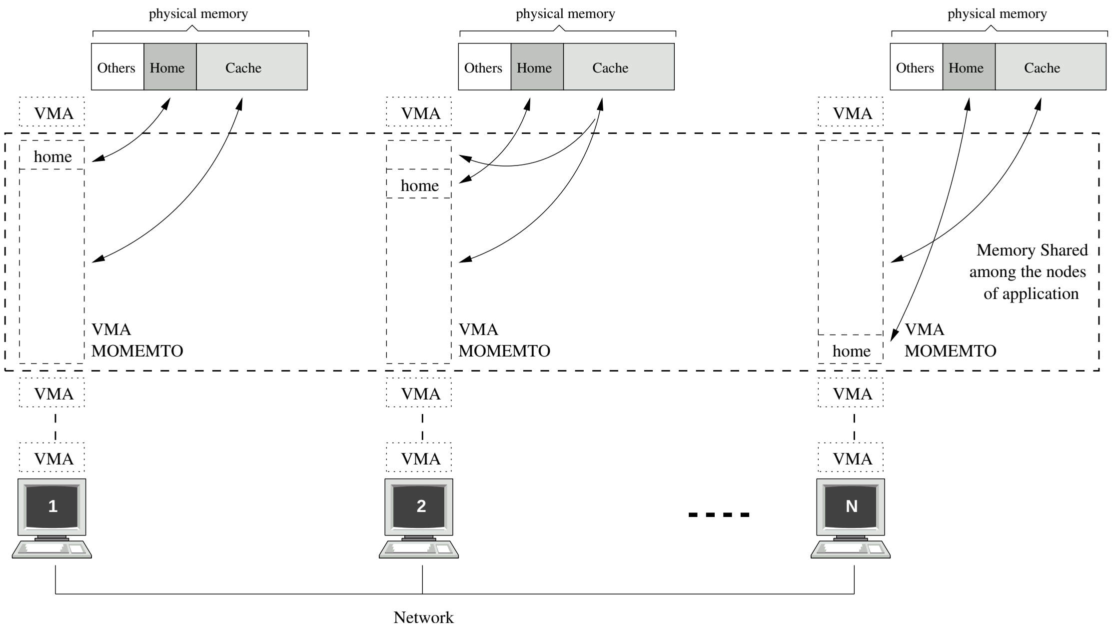

Figure 1. Basic Structure of MOMEMTO

tion to choose the corresponding page owner. An alternative, popular heuristic, is to make the home node for a page the first node that actually accesses the page.

MOMEMTO currently follows a strategy where pages are statically divided between nodes. Home nodes allocate the pages they own at initialization, simply by generating a page fault for each of the pages they own. Thus, MOMEMTO only caches pages that it is not the owner.

The lower MOMEMTO layer is not responsible for synchronization nor for page updating. Instead, this is a task for the upper layers, and possibly for application libraries. Most applications perform page updating at synchronization points, but the best update policy varies across applications. Indeed, often, the application knows which processors have copies of a page, and may easily find out the necessary requests. More general mechanisms can be supported through kernel modules or through application libraries.

MOMEMTO manages this shared area at independent way, leaving the remain of memory to the operating systems managements. It is obtained because MOMEMTO works only on your own VMA.

2.3. The Interface

The current MOMEMTO prototype provides the essential functionality for memory sharing. MOMEMTO was written over Linux-2.4. We chose Linux for several reasons, namely because it is widely used and because it is open sourced. One further advantage of Linux is a sophisticated dynamic module loading mechanism. Most of MOMEMTO was implemented using this functionality with very minimal changes to the standard kernel, that were implemented as a patch.

MOMEMTO requires three new operating system calls, named, *mmt_init*, *mmt_mmap* and *mmt_update*.

The *mmt_init* call initializes MOMEMTO. It receives as input a text file with the nodes that will share memory. All applications that use MOMEMTO must call this function first at all nodes that will collaborate. By using archives we can reuse the archives used by other systems. For example, MOMEMTO can understand the archives generated by job scheduling systems such as PBS [19] or SGE [18].

The *mmt_mmap* system call maps a virtual memory area that will be shared between the nodes involved in the previous system call. The system call is given as input how much memory is asked in bytes. All cooperating nodes must call *mmt_mmap*. The system call is also given a flag that specifies how home nodes are allocated. Currently, values for the flag are `ROUND_ROBIN`, or `SEQUENTIAL`.

The *mmt_update* provides a mechanism for actual sharing. This system call allows the user decide on when and which shared page the other nodes must receive an up-to-date copy. When the user updates a page, if the node is the page owner an up-to-date copy is sent to nodes with a copy. Otherwise, the page is first sent to the owner, which then resends to all sharers.

2.4. Prototype Implementation

MOMEMTO does not significantly change the kernel. All work is in fact performed by a plug-in module. We just needed to declare that the three symbols `vm_area_cachep`, `insert_vm_struct`, and `make_pages_present` should be visible to external modules. This way, MOMEMTO is more portable.

The implementation of the system call *mmt_mmap* is as follows:

- All nodes create a VMA with anonymous memory for the share area. VMA size is obtained from the syscall arguments. The VMA is then inserted into the list of VMAs for the process. Linux will call the MOMEMTO page fault handler for any page faults in this virtual memory ranges.

- All VMAs will have the same size, but they may be allocated at different virtual addresses. Nodes communicate using the offset of the page within the VMA, instead of the page's virtual address.

- At this point we create $N-1$ kernel threads, where N is the number of nodes. Each kernel thread i is responsible for handling page faults from node i and is also responsible for page updates sent to i.

Nodes communicate via TCP sockets. The Linux kernel provides a well-defined interface for socket access within the kernel. There is a socket for each node-node pair. The sockets are opened at startup and kept throughout execution. MOMEMTO implements a simple protocol, which is basically composed of request and reply messages over the reliable TCP network. They are used to request a remote page or update a page.

Context switches between kernel threads are much less expensive than context switches between common processes because kernel threads run only in kernel mode. When a request is received, the respective kernel thread is unblocked. The thread executes the request handling routine and returns to the blocked state in the end. The scalability of kernel threads is the same or better than user level communication mechanisms. Kernel threads demand the same resources and offer lower latency when compared with those mechanisms.

Although TCP/IP has important advantages in terms of availability and portability, it does not take best advantage of network bandwidth. We therefore designed our communication protocol in a way that it will be easy to port to other network such as VIA [1], Giganet [2], or Myrinet [17].

First accesses to a shared memory page result in a page fault to be processed by the MOMEMTO page fault handler. A page request is sent to the home node. The home node will then send a copy to the requester and set the bit in the copy array of the respective requester id.

The first MOMEMTO implementation allocates pages statically to each node. Thus each node receives $1/N$ of the total pages. The file given to *mmt_init* contains the hosts that share memory. The first node in the file has the first chunk of shared memory, and so on.

MOMEMTO also uses the replacement policy in kernel-2.4.17. We plan to allow specific memory policies per application.

MOMEMTO has a low overhead when compared to the kernel's own virtual memory data-structures. All structures required by MOMEMTO are initialized at module loading, and released at module unloading. MOMEMTO maintains the following data-structure for each node:

```
struct MOMEMTO_node {
        // number of nodes
        int NUM_NODES;
        // IP address for each node
        char name [MAX_HOST][16];
        // My name[] index
        int whoami;
};
```

MOMEMTO relies on three main data-structures, allocated as contiguous arrays:

- `home` is an array of bytes, where index i tells which node is home for page i;
- `copy` is an array of bitfields, one per each page owned by the node; each bitfield indicates which nodes are sharing the page;
- `last` indicates who last accessed a page.

MOMEMTO therefore spends:

- 6 bytes per page such that the node is the home node;
- 1 byte for other pages.

Notice that MOMEMTO did not change the kernel's VMA structure. The data-structures required for MOMEMTO are local to the module, this way, the MOMEMTO doesn't put any overhead for non-shared memory execution and kernel execution.

3. Performance

We evaluated our system in a three node cluster. Each node has a 650 MHz Pentium-III and 512MB of local RAM. Nodes are connected by a Fast Ethernet network. Each node uses an EEPro100 onboard NIC. We used the Linux operating system for our experiments, with kernel 2.4.17. Also it was used TreadMarks release 1.0.3.2 and MPI [7] LAM 6.5.3 for this evaluation.

3.1. The Vector Multiplication Application

We initially tested the system's ability to integrate cluster memory, and studied network communication patterns. We chose a simple vector multiplication (a component of a parallel scalar product) as our initial benchmark. This application displays well-understood communication patterns and can require significant memory. The application manipulates three vectors, $C = A * B$. Each node will initialize a vector. We attribute home pages contiguously and statically over the shared memory, and guarantee that each node will also be the home for one of the vectors.

Computation of the result vector C is equally divided among the three nodes. We did not attempt to optimize memory communication patterns.

In order to provide synchronization, we further implemented a centralized barrier. Barriers in our design only synchronize computation, not data. When reaching a barrier a node sends a 4 byte message to a *manager*. The manager waits for messages from nodes involved. After the manager is contacted by all other nodes, it sends a 4 byte message to release the nodes. Our implementation was performed at user-level as a library.

3.2. Results

We performed two experiments. In the first experiment (Small Load) we use 36,000 shared pages (12,000 per vector). This corresponds to a total of 141MB. In the second experiment (Large Load) we used 211,194 shared pages (70,398 per vector), representing a total of 825MB, more than the memory for each individual node.

Figure 2 shows the total running time for 36,000 pages for both MOMEMTO and TreadMarks [15]. Both systems use the same code for the actual vector multiplication.

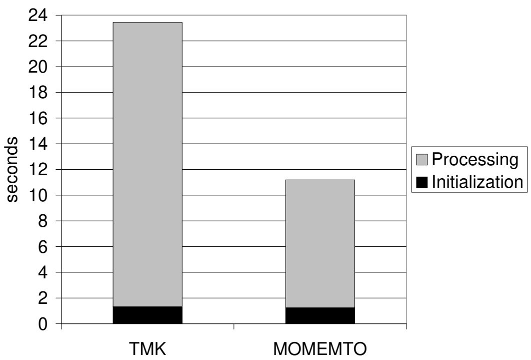

Figure 2. MOMEMTO x TMK

Figure 2 shows a improvement in the application execution time of 48% for the MOMEMTO version compared to the TMK version. This improvement is directly related to the difference in the number of page faults, system calls and interrupts issued within both systems. For the shared memory mechanisms offered by MOMEMTO are implemented in the kernel level, the number of context switches and the latency of the received interrupts in MOMEMTO are smaller than in TreadMarks. We believe it must still be true even when MOMEMTO implements a relaxed consistency protocol as the one used by TreadMarks.

Page-faults in MOMEMTO happen on demand, that is, only when a node accesses a page of the vector for which it is not the home node. Updating is performed after computing all elements of a page. The 9.96s of processing time thus include the time spent on updates, on page faults, and on two barriers. These times are shown on Table 1. Specifically, the time spent on the vector calculation was 0.55s.

	Number	Total	Network	Average
Updates	4000	2.78s	2.75s	694 μs
Page Faults	8000	6.63s	6.58s	828 μs
Barriers	2	0.0010s	0.0010s	500 μs

Table 1. Processing Time for Small Load

The $Average$ field shows the average time for each operation.

The difference between Total running time and Network time is the time actually spent running MOMEMTO code, that is, spent in the kernel. The actual MOMEMTO code execution time for update is 1% and for page fault is 1.47% of the Total running time. MOMEMTO imposes a minimal overhead compared with network latency. On the other hand, we expect that faster network technology such as Myrinet or Giganet will reduce network latency, making the contribution of MOMEMTO performance more significant.

Barrier synchronization time was quite high, showing some load imbalance. We can improve barrier time by using more sophisticated techniques as discussed in the literature [10].

In general, we believe this first benchmark argues our case that performance can be significantly improved through a protocol that uses specific mechanisms for memory management in the kernel level.

In our second test we study whether MOMEMTO could cope with a large memory load. We used 825MB of shared memory. TreadMarks does not run with this load in our evaluation environment. For this reason we show the execution time only for MOMEMTO version in Table 2.

The total execution time for MOMEMTO version was 63.2s for this load. As for previous test, total processing time includes time for updates, page faults, and two barriers. These times are shown in Table 3. Specifically, the time spent on the vector calculation was 5.35s.

	Total
Initialization	2.73s
Processing	60.47s

Table 2. Execution Time for Large Load

	Number	Total	Network	Average
Updates	23466	16.34s	16.22s	696 μs
Page Faults	46932	38.78s	38.15s	826 μs
Barriers	2	0.0012s	0.0011s	600 μs

Table 3. Processing Time for Large Load

Each node performed 23466 page updates (a third of the vector C). Each node generated 46932 page faults, corresponding to the two vectors for which the node is not home. Average time spent in MOMEMTO was about 5.3 μs for updates and 13.3 μs for page faults.

In this preliminary evaluation, a simple consistency protocol was used in MOMEMTO, allowing the application to decide on the update policy and protocol. This feature may contribute to the better performance of MOMEMTO when compared to TreadMarks. Therefore, we wanted to evaluate the kernel mechanisms offered by MOMEMTO comparing the executing time of the same application with an implementation using a message passing system. The Figures 3 and 4, show the execution time for the vector multiplication with MOMEMTO and the same application implemented with MPI [7].

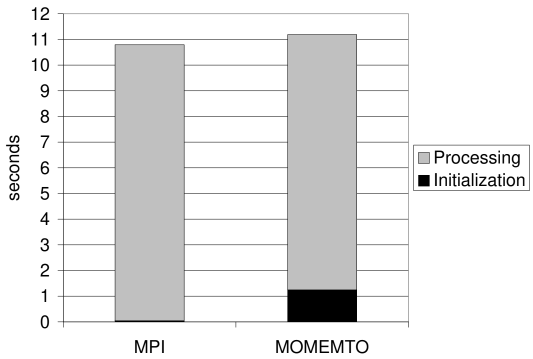

Figure 3. MOMEMTO x MPI Small Load

As in the previous tests, each node calculates a third of the vector C. Each node is the home of one vector and initializes it in the beginning of the execution. In the MPI version, messages are used to distribute equal parts of vectors A and B by their own home nodes. At the end, the home of vector C receives messages with the results of the other nodes.

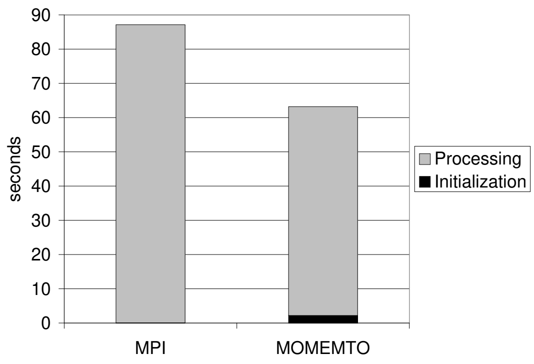

Figure 4. MOMEMTO x MPI Large Load

In the Figure 3, each vector has 12000 pages. Although MOMEMTO shows an increase of 3.5% in execution time compared to MPI, the processing time is decreased 7.5% by MOMEMTO. MOMEMTO forces a page fault in the initialization of each page the node has the homeship. This is the reason for the huge initialization time in MOMEMTO version.

The large load test, where each vector has 70398 pages, is shown in the figure 4. The execution time decreases 27% in MOMEMTO version.

MPI issues less messages than MOMEMTO. In the worst case, MOMEMTO needs 3 times more messages than MPI, as it is the case for a no-home node to get the chunk of vector C (when suffering a page fault) and to update it afterwards. A page fault, in MOMEMTO, issues 2 messages in kernel mode and the update of a page is made through a system call and a message in kernel mode, without any copy from user area. The Table 4 shows the communication time for MOMEMTO and MPI. The communication time is the time spent doing the update and handling the page fault in MOMEMTO version and the time sending and receiving the messages with the vectors A and B and sending and receiving the message with the results of vector C in MPI version.

	Small Load	Large Load
MPI	9.99s	82.53s
MOMEMTO	9.39s	55.12s

Table 4. Communication Time

In the Small Load test, MOMEMTO was 6% faster than MPI, and MOMEMTO reduced the communication time in 33% when running the Large Load test.

Again, we argue that the better performance of MOMEMTO is related to the low overhead of its operations in kernel mode. MOMEMTO version suffers less context switches and has lower interrupt latency than MPI version.

Also, from this preliminary evaluation we may conclude that MOMEMTO was indeed effective at aggregating cluster memory efficiently, allowing applications to run with large shared memory, and improving execution through providing basic synchronization primitives, with low overhead.

4. Related Work

There has been a lot of work on using remote memory for clusters. One common application is to use the network for swapping-out memory. Distributed applications can benefit from actually sharing memory between nodes. Distributed Shared Memory systems [16, 20] thus try to provide a shared memory abstraction over the network.

GMS [13], Global Memory Service, manages memory distributed over a cluster. Each node receives an area of global memory which may increase or decrease according to the node's load. GMS was designed toward providing network swapping, thus eliminating disk requirements: idle nodes will store pages for active nodes. There is no effort to implement a replacement policy or to optimize the location for shared memory.

Cashmere-VLM [11] is a DSM system that can perform swap over the network. Cashmere-VLM tries to increase the area shared by an application releasing memory from an overloaded node. In other words, Cashmere-VLM tries to use information on copies of pages on other nodes to release memory. Cashmere-VLM works at the user-level, and it implements a fixed protocol. One problem with working at the user-level in large applications is that the mapping and unmapping techniques used by the libraries will result in many small areas of virtual memory and induce significant overheads. One interesting characteristic of Cashmere-VLM is that it allows pages to dynamically change home page.

JIAJIA [14] is closest to our proposal. Again, JIAJIA is a software DSM where the workstations physical memory is combined in a large shared space. The system uses coherency protocols based on locks. Again, MOMEMTO differs from JIAJIA in that it shares kernel pages at the kernel level and does not force a coherence protocol, allowing more flexibility for the application.

5. Conclusions

In this work we introduced MOMEMTO (MOre MEMory Than Others) and performed an initial evaluation. Our system is designed towards providing maximum flexibility and allowing shared-memory systems to exploit full shared memory between nodes. In contrast with many software DSM systems, we do not propose a monolithic user-level library. We argued that applications require a flexible and modular solution. Moreover, we supported that large-scale applications require better kernel support, and that nowadays such improvements are easily available and should be taken.

MOMEMTO shows low execution overheads in our preliminary evaluation. We also show that the MOMEMTO system can indeed perform well for large memory configurations while introducing small overheads in the virtual memory management system.

We are now working on other applications that require extra MOMEMTO primitives. Our target applications include database applications, web servers, and numerical computing. We also plan to study the impact of interconnect technologies such as Myrinet and GigaNet on MOMEMTO.

References

[1] Compaq, Intel and Microsoft. VIA Specification 1.0 Available at. http://www.viarch.org.

[2] Giganet, Inc. http://www.emulex.com.

[3] NERSC. M-VIA. http://www.nersc.gov/research/FTG/via.

[4] Rocketcalc. http://www.rocketcalc.com.

[5] NCSA Urbana-Champaign. http://www.ncsa.uiuc.edu, 2001.

[6] CBRC - Tsukuba Advanced Computing Center - TACC/AIST Tsukuba. http://www.cbrc.jp/magi/, 2001.

[7] Message Passing Interface Forum. http://www.mpi-forum.org.

[8] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[9] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[10] A. G. David E. Culler, Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, August 1998.

[11] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote Memory Paging for Software Distributed Shared Memory. In *Proc., IPPS*, April 1999.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[13] M. J. Feely, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Tekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 201–212, December 1995.

[14] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In *In Proceedings of the High Performance Computing and Networking (HPCN'99)*, pages 463–472, April 1999.

[15] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[16] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *ACM Transactions on Computer Systems*, pages 321–359, November 1989.

[17] Myricom, Inc. Myrinet. http://www.myri.com.

[18] Sun Microsystems Inc. SGE. http://wwws.sun.com/software/gridware.

[19] Veridian Systems. OpenPBS. http://www.openpbs.org.

[20] L. Whately, R. Pinto, M. Rangarajan, L. Iftode, R. Bianchini, and C. L. Amorim. Adaptive Techniques for Home-Based Software DSMs. In *Proceedings of the 13th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*, pages 164 – 171, September 2001.

[21] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, 1996.