Comparing the Performance of the PastSet Distributed Shared Memory System using TCP/IP and M-VIA

John Markus Bjørndalen, Otto J. Anshus, Brian Vinter[1], Tore Larsen
Department of Computer Science
University of Tromsø
[1]Department of Mathematics and Computer Science
University of Southern Denmark
johnm@cs.uit.no, otto@cs.uit.no, vinter@imada.sdu.dk, tore@cs.uit.no

Abstract

Using TCP/IP or M-VIA, the performance of the structured distributed shared memory system PastSet is measured and compared to a reference single-node implementation (excluding all intra-node communication). The latencies of PastSet-operations are measured using several micro-benchmarks. For the experiment setup used, M-VIA latencies are shown to be between 1.4 and 2.2 times lower than the comparable latencies using TCP/IP. For a data size of 31KB, this corresponds to a difference of more than one millisecond. Depending on the thread-allocation policy applied in the PastSet server, PastSet latencies using TCP/IP may exhibit increased variance compared to the corresponding latencies using M-VIA. The increased latency and variance may mask the performance characteristics of the PastSet system.

1. Introduction

The application-to-application performance of a Distributed Shared Memory (DSM) system depends on the performance and interaction of the DSM and the underlying network subsystems. The key challenge [16] is to preserve the performance characteristics of the physical network (bandwidth, latency, QoS) while making effective use of host resources. Network bandwidths have been increasing and latencies through these networks have been decreasing. Unfortunately, applications have not been able to take full advantage of these performance improvements due to the many layers of user level and kernel level software required to use the network. A detailed breakdown of hardware and software costs of remote memory operations is discussed in [5]. The Virtual Interface Architecture was developed to significantly reduce the software overhead between a high performance CPU/memory subsystem and a high performance network.

In this paper, we study the application-to-application performance of the PastSet DSM system using either M-VIA, a software VIA implementation for Linux [6], or TCP/IP. The report describes the functionality of PastSet, the organization of the implementation; and the interactions between PastSet user-level components, M-VIA, and a PastSet-modified Linux kernel. An experiment configuration with micro-benchmarks and metrics is described before presenting and analyzing benchmark results.

2. PastSet

2.1. Model

PastSet was first introduced as a structured shared memory in [1]. Early partial implementations of PastSet include [2, 3]. [4] develops and demonstrates an extended PastSet programming model, shows how parallel applications are written using PastSet, and documents the performance of these applications on the authors implementation of PastSet. The PastSet paradigm resembles that of Linda [18], but with added structuring of the shared memory and different functionality of the operations provided. Comparable, efforts within industry include IBM TSpaces [15] and JavaSpaces [14].

All PastSet operations are synchronous, returning only when the operation is completed, or an error has been detected. Processes using PastSet dynamically generate tuples based on tuple templates that may also be generated dynamically. A tuple template specifies a list of data types. A tuple is a list of values matching the data types specified in the template upon which the tuple is based.

The ‘elements’ of PastSet are lists of tuples; one list per unique template used. The `enter` operator takes a tuple template as parameter and establishes a binding from that template to the associated element in PastSet. If the template is unique (i.e. no identical template has been specified for previously executed `enter`-operations), a new element is created. If the template is not unique, the binding is established with the element already associated with the identical templates. As with Linda, PastSet sup-

ports writing (called `move`) tuples into PastSet and reading (called `observe`) tuples that reside in PastSet. A tuple that is moved into PastSet is added to the associated element in PastSet and remains in that element as a unique tuple. For each element, tuples are added and observed in FIFO or program-specified order as described in [4]. Two identifiers `First` and `Last` are associated with each element in PastSet. `First` refers to the elements oldest unobserved tuple. `Last` refers to the tuple most recently added to the element. A parameter, `DeltaValue`, associated with each element in PastSet defines the maximum number of tuples allowed between `First` and `Last` for that element. A process may change `DeltaValue` at any time. `Move` and `observe` update `First` and `Last`, and obey the restrictions imposed by `DeltaValue` for each element in PastSet. PastSet preserves the sequential order among `move` and `observe` operations on tuples based on identical templates. There is no ordering among operations on tuples based on different templates.

A combined `move-observe` operation, `mob`, is provided to support efficiently the commonly used sequence of a `move` immediately followed by an `observe`. `Mob` takes two tuples as parameters and operates on two elements if the tuples are based on different templates; if not, `mob` operates on one element.

Contrary to similar systems, PastSet `observe` does not remove tuples from PastSet, observed tuples are tagged 'observed' but remain in PastSet and may be observed again later. A mechanism is provided to coarsely truncate PastSet on a per element basis, permanently removing all tuples that are older than a given tuple. There is no mechanism to remove individual tuples from PastSet.

2.2. Organization

One or more nodes may host PastSet. Each hosting node runs a PastSet kernel, a server, and an application library (Fig. 1). Currently, each element in PastSet is stored on one host only. There is no distribution, replication, or migration of individual elements. Nodes that use PastSet without hosting it will run the application library and TCP/IP or M-VIA only.

The organization supports PastSet operations on elements that are located on the same node as the initiating process (local operations), as well as operations on elements that are located on other nodes (remote operations). Remote operations are wrapped and communicated to the PastSet server on a remote node. `Mob` operations work on one or two elements, each of which may be local or remote.

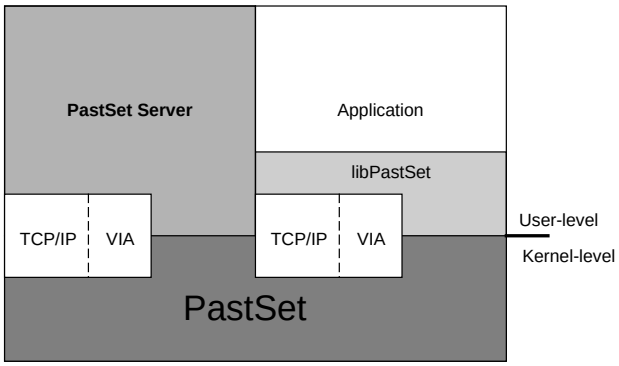

Fig. 1. Layout of a node that supports PastSet

THE PASTSET APPLICATION LIBRARY handles access to PastSet, including multiplexing between local and remote execution paths.

All PastSet operations check to determine whether the element that is to be operated on is hosted locally or remotely. If the element is hosted locally, the operation is executed using the local PastSet Kernel. If the element is hosted remotely, the operation is redirected to the PastSet Server on the appropriate host. The server, upon completion, returns the answer through the application library to the initiating process.

New tuples with additional space for communication headers are also allocated via the application library.

THE PASTSET SERVER executes remotely issued PastSet operations on local elements using the local PastSet Kernel. Since operations are blocking, the server must be able to service several operations concurrently.

THE PASTSET KERNEL is a modified Linux kernel that stores PastSet elements and services operations issued by the local PastSet Server or the application library.

3. Implementing the PastSet Server and Application Library

Two approaches to handling communication were used in the PastSet server. The *Single Thread* approach spawns a new thread for each new client connection. The thread is given a pointer to the new connection and handles that connection exclusively.
The *Thread Pool* approach uses a pool of threads, which multiplexes handling of multiple connections.

The synchronous nature of PastSet necessitates a reply with the result of an operation before the client can issue a new operation. Consequently, there are two messages for each remote operation.

The current implementation of the PastSet Application Library can not handle multithreaded clients.

3.1. TCP/IP implementation

The client library uses ordinary TCP/IP connections to remote servers. If an element is located on a remote node, the operation and its parameters are sent over the TCP/IP socket to the remote PastSet server. The client library then blocks waiting for the reply from the server.

When using the Single Thread approach, the PastSet server creates a new thread when a new socket arrives. The newly created thread is given the file descriptor of the socket and immediately tries to read data from the socket. It blocks if there is no data available.

In the Thread Pool approach, when a connection is accepted, the file descriptor is added to the list of active connections. A set of threads use the *select()* system call to multiplex themselves between the active sockets. To avoid race conditions, the *select()* call and the subsequent read of a socket with data is protected with a mutex.

We disable the Nagle algorithm on all sockets to ensure that data is sent immediately.

3.2. M-VIA implementation

The PastSet server and the application library were implemented using the M-VIA 1.0 [6] implementation of the VIA API. We have used the message passing model of VIA since the port from the TCP/IP implementation was straightforward.

The NICs used do not support the doorbell mechanism, and M-VIA has to emulate this in software. This results in traps to the Linux kernel.

The Thread Pool model was implemented using VIA Completion Queues. Because M-VIA is not thread safe per VI we protected the calls to the Completion Queue and the per VI operations using mutexes.

To reduce the CPU use of the PastSet server we used blocking calls to wait for completed descriptors. M-VIA implements these blocking calls by first spinning a few times with the respective non-blocking functions to avoid going to the kernel if the descriptor is already completed.

Tuples used in the application are allocated in memory registered with the VIA NICs to reduce copying on send and receive.

4. Methodology

This section documents the hardware and software details of the experiments, how the timing measurements were done, the micro-benchmarks, and the metrics used.

4.1. Hardware and Software

All experiments were done using one, two, or three HP LX-Pro Net-servers, each having four 166MHz Pentium Pro CPUs and 128MB main-memory, and dual peer 33MHz, 32 bit PCI buses. The level 2 cache size is 1MB per processor. The computers were interconnected using either Intel 82255 or Trendnet TE100-PCIA (with Tulip chip set) NIC-cards connected to a hub. Both NICs, and one 100VG NIC connected to the outside network, were connected to PCI bus no. 0.

Linux v. 2.2.14 with PastSet functionality added to the kernel was installed on each node participating in the experiments. M-VIA version 1.0 with a minor patch to the connection management was used. The benchmarks and the PastSet Application Library were compiled with gcc 2.95.2 using optimization flags "-O6 -m486 -mjumps=2 -malignloops=2 –malignfunctions=2." M-VIA, The PastSet Server, the PastSet Kernel, and the Linux operating system were compiled with egcs 1.1.2 using default flags.

For some experiments we could not get M-VIA running on the SMP-configurations. To circumvent this problem, we had to resort to compiling the Linux kernel to run as a single processor system rather than using all four processors in a node.

4.2. Time Measurements

The Intel Pentium Pro RDTSC (read time-stamp counter) instruction and the Linux gettimeofday system call were used to determine PastSet operation latencies.

Using RDTSC, as in [17], the cycle count was recorded for every `move` and `observe` operation. Elapsed time in microseconds was calculated by dividing the registered cycle count by the specified processor frequency of 166 MHz. No attempts were made to verify the actual frequency of each individual computer, leaving open the possibility that the computed time may deviate slightly, but consistently, from the performance measured in cycles spent. Care was taken to avoid potential problems with register overwrites and counter overflow.

The *gettimeofday()* system call was used for aggregate measurements over many operation calls. Checks were made to ensure that RDTSC and *gettimeofday()* measurements were consistent.

Cache effects are not eliminated, but measurements are averaged over 1000 iterations.

4.3. Micro-benchmarks and Metrics

Two micro-benchmarks that measure operation and ping-pong latencies of the PastSet system were designed:

- *`Move latency (mvlat)`*, *`observe latency (oblat)`*: Time to invoke, complete and return from a `move` or `observe` operation.
- *Ping-pong latency (pplat)*: Time to exchange data between two processes using `moveobserve`.

The benchmarks were executed inside client processes running both on the same computer as PastSet *("Local Latencies")* and on remote computers. When using more than one computer TCP/IP or M-VIA were used for communication.

When doing the performance measurements each node supported no other workload except for the operating system and its various artifacts.

5. Micro-benchmark Results

5.1. Operation Latency Experiment Design

PastSet "operation latency" is defined to be the time elapsed from a `move`, `observe`, or `mob` operation is called until it has completed and returned successfully. For `observe` operations it is assumed that enough tuples are available in PastSet to prevent the operations from blocking for lack of tuples. All necessary initializations are done before starting time- or cycle measurements.

```
for(i=0; i<1000; i++)
{
    save_timestamp;
    mv();
}
save_timestamp;
```

Fig. 2: The Mvlat benchmark

The client process executes `move` or `observe` operations. Data size per operation call is varied from zero to 31KB. The elapsed time is measured for each operation call. Each call is repeated 1.000 times. This is repeated five times, and the average is computed.

Due to space constraints, results are shown only for the move operation. The observe operation exhibit slightly different behavior, but is close in performance.

Three experiments were designed to measure the latency of the move operation:

- Local Move Latency: The client process and PastSet are on the same node.
- TCP/IP Move Latency: The client process and PastSet are on different nodes. TCP/IP is used for intra-node communication.
- M-VIA Move Latency: The client process and PastSet are on different nodes. M-VIA is used for intra-node communication.

The latency experiments were conducted on the following configurations, using both SMP and uniprocessor versions of Linux:

- A pool of threads is used in the PastSet server to serve all connections ("Thread Pool").
- A single thread is used in the PastSet server per connection ("Single Thread").

Because of problems experienced with M-VIA, the M-VIA Move Latency experiment was conducted only for the "single thread" version .

The configurations used for the experiments are summed up in table 1.

Table 1: Configurations

	Local	**Using M-VIA**		**Using TCP/IP**	
SMP	(Fig. 3 & 4)	Intel NIC, Thread Pool (Fig.3 & 4)		Intel NIC, Thread pool (Fig. 3 & 4)	
Uni-processor	(Not shown)	TREN Dnet NIC, Single thread (Fig. 5)	TREND-net NIC, Thread Pool (Not done)	TREND-net NIC, Single thread (Fig. 5)	TREND-net NIC, Thread pool (Fig. 5)

5.2. Move Latency Results

Fig. 3 shows move latencies for local (one node only) communication, and for intra-node communication using TCP/IP or M-VIA. Tuple sizes are varied from one byte to 31KB. The results are measured using Linux in SMP mode, a thread pool in the PastSet Server for data in/out servicing, and using the Intel NICs.

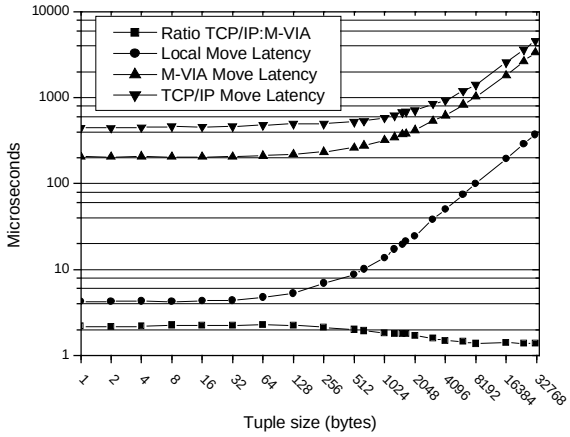

Fig. 3. *The mvlat benchmark*: Operation latency for `move`

Fig. 3 shows that local move latency (clients and server on same node) changes from about two to one order of magnitude better than M-VIA or TCP/IP as tuple size is varied from 1 byte to 32 KB. The difference in performance is due to the extra overhead caused by network communication and using the PastSet Server.

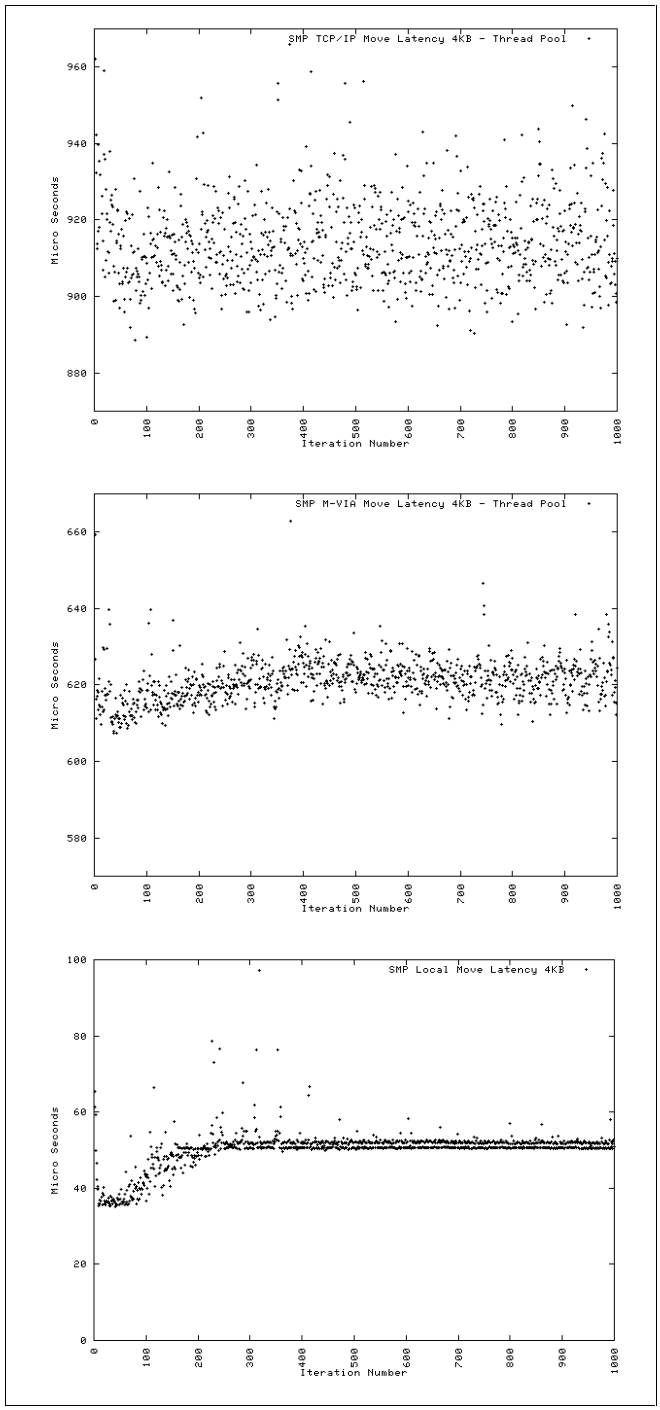

Fig. 4. *The mvlat benchmark*: `move` latency for tuple size 4KB for 1000 measurements, SMP, Thread

For small tuple sizes M-VIA `move` latency is less than half the latency when using TCP/IP. With increasing tuple size, the M-VIA advantage decreases to about 2/3 for 31 KB tuples. This is a large difference in absolute numbers. For one byte tuples, the difference between using M-VIA and TCP/IP is 242 microseconds, while at 31KB the difference is 1256 microseconds. We explain the advantage of M-VIA over TCP/IP partly by the user level communication used by M-VIA and the faster traps to the operating system. Other contributing factors are that M-VIA does not compute checksums of the incoming packets, taking advantage of the properties of a local net. However, TCP/IP uses the operating system much more heavily, and does more copying than M-VIA. Fig. 4 shows the `move` latency of 1000 `move` operations for a single tuple size, 4KB. The results are measured using Linux in SMP mode, using a thread pool in the PastSet Server for data in/out servicing, and using Intel NICs.

We have plotted three cases: local, TCP/IP and M-VIA. The TCP/IP measurements include a few very high (factor 10) values that we have removed from the plot. We believe that these values are the result of 10ms time slice events.

The results for the local `move` latency clearly show an effect coming from the way PastSet implements receiving and storing of tuples. The storage structure of PastSet seeks to make it efficient to access the newest tuples. Three levels of indirection are used to achieve this ("reverse I-nodes"). When a stream of tuples are sent to PastSet the cost of inserting them grows steadily until we are at level three in the datastructure. This effect can be seen in the local move latency plot, and to a lesser extent also in the M-VIA plot.

The measurements show that M-VIA has a lower variance while TCP/IP gives a much more unpredictable latency. We believe that the TCP/IP latency is long enough to include relatively more events (including interrupts and scheduling) happening in the total system resulting in a high variance. Also, the thread pool in the PastSet Server uses the *select()* system call when using TCP/IP. This seems to be more expensive than using the VIA completion queue mechanism.

The results suggest that M-VIA and the way we use it, even when using a slow 100Mbit network, is just fast enough to let the basic behavior of the PastSet system become visible in the measurements. This is due both to the better latency of M-VIA and less variance.

TCP/IP does not reveal the behavior of PastSet, while M-VIA does.

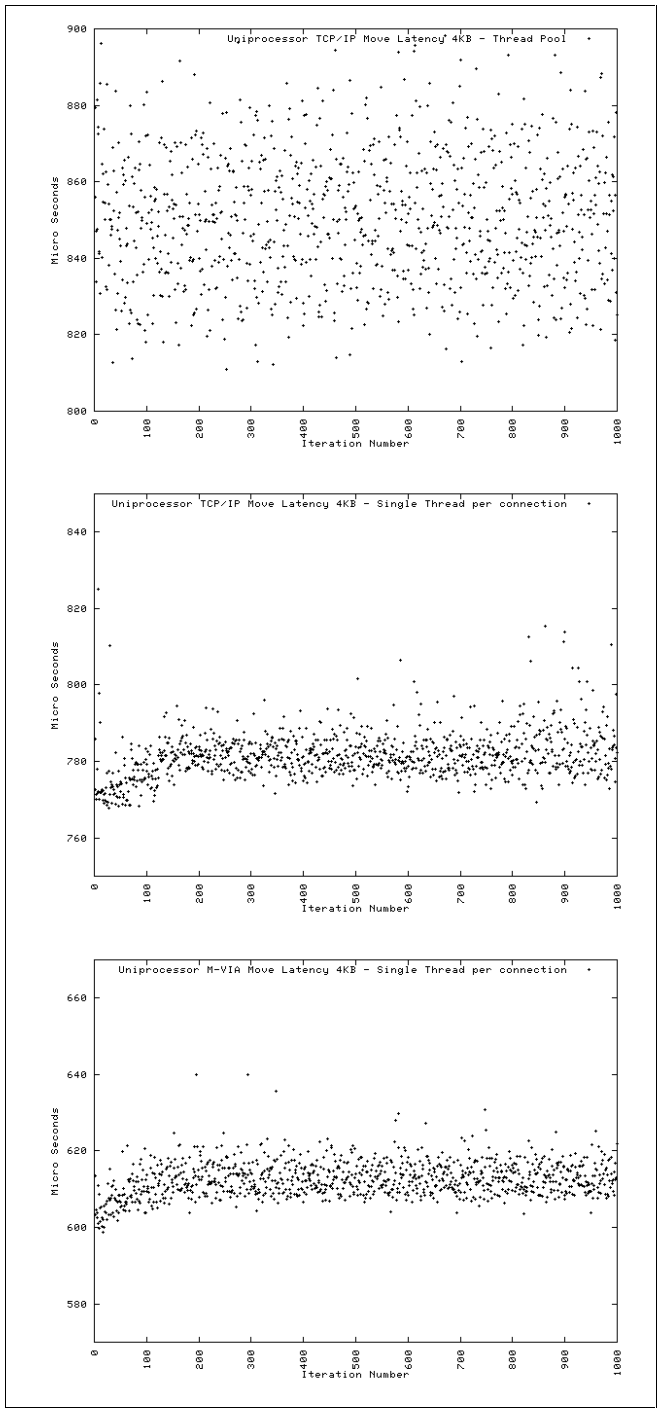

Fig. 5: *The mvlat benchmark*: `move` latency for tuple size 4KB for 1000 measurements, Uniprocessor and SMP, Single Thread and Thread Pool, TRENDnet NIC.

Fig. 5 shows the move latency of 1000 move operations for a single tuple size, 4KB. The results are measured using Linux in uniprocessor mode, and TRENDnet NICs are used. The PastSet Server uses a single thread for each connection for the M-VIA, and a single thread or a thread pool for the TCP/IP. We have not been able to use a thread pool for the M-VIA measurements using TRENDnet NICs due to problems with M-VIA.

The results from fig. 5 show that TCP/IP improves significantly when using a single thread to handle the benchmark connection as compared to using a thread pool. TCP/IP is still slower than M-VIA, but the variance has improved, and is just slightly worse than for M-VIA. We explain this with the way the PastSet server handles TCP/IP connections. In the single thread per connection configuration there will almost always be a thread ready to read incoming data, and this thread will be the same every time. In the thread pool configuration a new thread will serve each incoming packet. We believe this has impact on the cache footprint giving more variance and worse results. Also, the thread pool configuration executes more instructions.

We have not been able to measure the move latency when using M-VIA, TRENDnet NICs, and a thread pool on a Linux uniprocessor configuration. If we assume that M-VIA will behave about the same or better than TCP/IP, then we can conclude that M-VIA is less influenced than TCP/IP on whether a single thread or a thread pool is used in the PastSet server. TCP/IP is much more sensitive to this as can be seen by comparing the two TCP/IP results in fig. 5. We explain this difference in sensitivity to the M-VIAs better utilization of user level communication.

Generally, TRENDnet NICs are faster than the older Intel NICs.

5.3. Ping pong latency

PastSet "ping pong latency" is the time period elapsed between the repeated lock stepped exchanges of a value between two processes. What we measure will include potential waiting by the two processes for each other to rendezvous. All initializations have been done before we start counting time.

```
gettimeofday();
for(i=0; i<1000; i++){
  mob(); // Exchange a value with
the other process
}
gettimeofday();
```

Fig. 6. The PPlat Benchmark

For the ping-pong latency performance measurements, we execute the micro-benchmark using two processes. Both processes loops doing a given number of mob() operations. One process moves data to element e_1, and tries to pick up data from element e_2. The other process moves data to element e_2 and tries to get data from e_1. In this way, the processes exchange data in a lock step fashion.

The data units range in size from zero to 31KB. The elapsed time to exchange all data is measured, and the time per exchange is computed. The step locked operations are repeated 1000 times to eliminate noise.

We did five ping-pong experiments (in the legend tag, L means local, R means remote relative to the location of PastSet):

- LL: The two processes and PastSet are on the same computer.
- M-VIA LR: One process is on one computer, the other process and PastSet are on another computer. VIA is used for communication between the two computers.
- TCP/IP LR: One process is on one computer, the other process and PastSet are on another computer. TCP/IP is used for communication between the two computers.
- M-VIA RR: The two processes and PastSet are all on different computers. M-VIA is used for communication between the three computers.
- TCP/IP RR: The two processes and PastSet are all on different computers. M-VIA is used for communication between the three computers.

The results presented in fig. 7 show that the ratio between using TCP/IP vs. M-VIA when the communication processes are on two different nodes behaves as the same ratio for the move latency. The advantage of using M-VIA decreases when the tuple size increases. However, M-VIA has a significantly better latency at all tuple sizes, and especially for tuple sizes between 0-256 bytes.

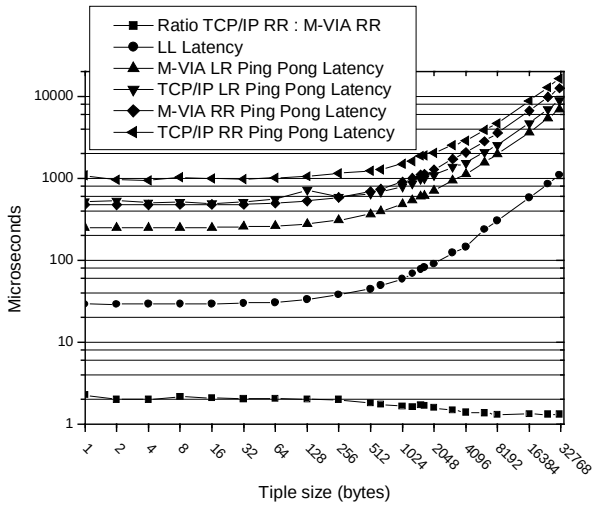

Fig. 7. *The Pplat Benchmark*

Fig. 7 also shows that for tuple sizes up to 2KB, using M-VIA on a three-node configuration (RR) gives about the same latency as when using TCP/IP on a two-node configuration (LR). Thus, M-VIA is fast enough to make up for the extra communication taking place.

6. Related work

Much effort has been put into cluster communication using either a shared memory model or an explicit communication model.

When comparing our results with the results reported in [19] we find that the ratio between TCP/IP vs. M-VIA latency is about the same and around 2.25-2.26.

Distributed Shared Memory implementations include Princeton Shrimp SVM [8] and Rice TreadMarks [9]. Object based Distributed Shared Memory systems include Orca [10]. Noteworthy examples of message-passing systems include Message Passing Interface, MPI [11], and Parallel Virtual Machines, PVM [12]. Less work has been done using Structured Distributed Shared Memory. The most well known systems include Linda [18], and more recently, Global Arrays [13].

7. Conclusions

Porting PastSet from TCP/IP to M-VIA proved straightforward. However, we have identified several bugs or limitations of the M-VIA implementation we used, and we still do not have a stabile system available.

When using a DSM system a predictable performance is desirable, and in addition to being faster, the latency of M-VIA is more predictable than the latency of TCP/IP. Operations on tuples of 256 or fewer bytes are twice as fast when using M-VIA.

By using a standard API such as VIA, system designers can achieve the benefits of user-level communication, while still maintaining portability.

8. Acknowledgements

Ken Arne Jensen and Jon Ivar Kristiansen provided invaluable support under the preparation of this paper.

9. References

[1] Anshus, O.J., Larsen, T.: “MacroScope: The Abstractions of a Distributed Operating System”. Norsk Informatikk Konferanse 1992, October 1992.

[2] Helme, A., “Scheduling of Processes in a Distributed System using a Multi Dimensional Algorithm” (in Norwegian), Master Thesis, Dept. of Computer Science, University of Tromsø, Tromsø, Norway, 1992.

[3] Stabell-Kulø, T., “A Partial Implementation of the MacroScope Distributed Operating System (in Norwegian)”, Master Thesis, Dept. of Computer Science, University of Tromsø, Tromsø, Norway, 1992.

[4] Brian Vinter, “PastSet a Structured Distributed Shared Memory System”, Dr. Scient. Thesis, Tromsø University, 1999.

[5] Bilas, A., Iftode, L., Singh, J. P., “Evaluation of Hardware Support for Automatic Update in Shared Virtual Memory Clusters”. 12th ACM International Conference on Supercomputing, July, 1998
[6] http://www.nersc.gov/research/ftg/via/
[7] Brian Vinter, Tore Larsen and Otto J. Anshus, “Improving Cluster Performance using a Causally Ordered Structured Distributed Shared Memory System”, Norsk Informatik Konferense ‘.99, 1999.
[8] Blumrich, M., Li, K., Alpert, R., Dubnicki, C., Felten, E., Sandberg, J.: “A virtual memory mapped network interface for the shrimp multicomputer”. In Proceedings of the 21st Annual Symposium on Computer Architecture, pages 142–153, Apr. 1994.
[9] Keleher, P., Cox, A., Dwarkadas, S., Zwaenepoel, W.: “TreadMarks: Distributed shared memory on standard workstations and operating systems”. In Proceedings of the Winter USENIX Conference, pages 115–132, Jan.1994.
[10] Bal, H.E., Kaashoek, M.F., Tanenbaum, A.S.: ‘Orca: A Language For Parallel ProgrammingOf Distributed Systems”. IEEE Computer 25(8), pp. 10-19, Aug. 1992.
[11] Walker, D.W.: “The Design of a Standard Message-Passing Interface for Distributed Memory Concurrent Computers”. Parallel Computing, Vol. 20, No. 4, pages 657-673, April 1994
[12] Sunderam, V.S.: “PVM: A Framework for Parallel Distributed Computing”. Concurrency: Practice and Experience, Vol. 2, No. 4, Dec. 1990.
[13] Nieplocha, J., Harrison, R.J., Littlefield, R.J.: “Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers”. Proceedings of the conference on Supercomputing ‘94, pages 340-ff., 1994
[14] Eric Freeman, Susanne Hupfer, Ken Arnold, “JavaSpaces(TM) Principles, Patterns and Practice”, SUN MicroSystems
[15] http://www.almaden.ibm.com/cs/TSpaces/
[16] Druschel, Peter and Peterson, Larry L. “Operating Systems and Network Interfaces,” In Foster, Ian and Kesselman, Carl (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 1999
[17] Chen, J. B., Endo, Y., Chan, K. Mazieres, D., Dias, A., Seltzer, M., Smith, M.D.: “The Measured Performance of Personal Computer Operating Systems”. ACM Transactions on Computer Systems, February 1996.
[18] Carriero, N., Gelernter, D., “Linda in Context”, Commun. ACM, (April 1989), Vol. 32, No. 4, pp. 444-458]
[19] Speight, E., Abdel-Shafi, H., Bennett, K., “Realizing the Performance Potential of the Virtual Interface Architecture”, International Conference on Supercomputing, June 1999
[20] Vinter, B., Anshus, O.J., Larsen, T., “Data Distribution Models for a Structured Distributed Shared Memory System”, Proc. Of the international conference on Parallel and Distributed Programming Techniques and Applications, PDPTA 99, Las Vegas June 1999