Matthew Rakel
Christopher Mersman
Bradley Richards

Distributed Shared Memory

Introduction

As stated in Moore's Law, the processing power of our computers is growing exponentially every year. From the creation of the computer, to present day we are always just one step from the next big leap in processing. Yet even as our computers can process more data in less time, many other aspects still bottleneck the performance of modern day machines, namely memory accesses. Even without this bottleneck, users have long needed computations on data that no single computer could conceivably do alone. In this need distributed systems were born.

A solution to many problems, distributed systems also produced a new set of bottlenecks. One of the biggest bottlenecks is the communications between different computers in the distributed system. This extrapolated the memory access bottleneck, as each computer were no longer just limited by the speed that they access their own memory, but also by the rate that they could get and receive data from other parts of the network. Therefore message passing tightened the bottleneck, and hinders the overall performance of the distributed system.

From this need the idea of distributed shared memory was born. Taken from the idea of sharing memory between processes, the idea seems to be simple enough, yet the implementation is not so. Unlike shared memory between processes, distributed shared memory has a few more hurdles to overcome. Distributed shared memory does not have the luxury of being contained to one machine, instead the paths between the processor and its memory can change. Depending on these paths it can completely hinder the system.

Going to the basics you could define a Distributed shared memory (DSM) as a computer system architecture where a large number of individual computers share memory over a network. Systems that implement DSM are also called multicomputers, which is a more informative name in many respects. DSM is implemented by networking multiple computers together and creating a shared address space that is used by all computers. Basically DSM is just one really big computer that is connected wirelessly. Rather than returning to the days of single computers taking up a wall in the basement, this approach allows individual PCs to be integrated without physical connections.

As we move closer to the wall of processing speeds and transistor size, it seems that our

means of accessing memory have slowed down. The path to main memory through the a bus only allows minimal amounts of permission and speed to its contents. On such a system, only a small amount of processors may be provided for memory access. In a situation of distributed shared memory, many processors can access different memory through a message system on a network, or through their own local caches. In a high speed, tightly connected network, several nodes can “share” memory by sending packets and messages to each other that make memory abstract and can sometimes have multiple pieces of the same memory within the system. This is where problems come in.

A couple problems with these systems is the maintenance of memory and thread control. Just like synchronizing threads in programming for the access of variables, we do not want to have situations where there are race conditions on variables and places in memory that are constantly changing. These situations are more likely to happen with multiprocessor nodes within the system allowing different cores to access the same memory. An attribute that has an influence on this is the amount of writes and reads. A system that has more reads is going to be treated differently than a system that has more writes.

The purposes of these systems can be very different; however, in most they are implemented for a larger memory storage or for faster memory access systems that have nodes that may or may not be local. Shared Memory is usually implemented over a network in which memory is passed or kept within local caches. With all this distributed work, problem solving can be very fast, especially on a large scaled project. Instead of having six processors working on a local machine, you can have six processors on twenty machines working. Into the future these solutions to the shared memory problem will become ever more viable. By combining the software solution and the hardware solution into a hybrid solution, the future looks bright for distributed shared memory systems. Mixing hardware with software we can open up the current bottlenecks and have shared memory systems whose memory accesses are nearly as fast as a non-distributed shared memory system, while still remaining cost effective.

Distributed shared memory systems are the next step for our computing devices. This will open up many different possibilities, that not only make our systems all the more powerful, but also more useable. With faster Internet connections our home systems may simply stream applications instead of needing to install them. Developers can also make use of a distributed shared memory to quickly load and exchange operating systems, or applications in order to test their own code.

As we dive a little further into what Distributed Shared Memory is, we will be talking about the ways to implement such systems, the tradeoffs that will need to think about, and the solutions of real world applications. These systems are going to become the future for access and storage where we will need to implement these in faster ways. Since Moore’s law is running down we will need to find a faster solution, and this is the route humanity seems to be taking. The future of many

computing disciplines will change because of distributed shared memory. From cloud computing to consumer devices, distributed shared memory will help to increase the speed and usability of our computing devices.

Problem Characterization

Distributed Shared Memory (DSM) is a type of memory architecture that allows physically unconnected processors to access the same virtual memory space as if it is within the local system. In order to overcome the high cost of communication a DSM will move the data that needs to be accessed to the location of the requested system. Each data object is owned by a node, this ownership changes as it moves from one node to another. The DSM is responsible for keeping track of where the data object is at all times so that any process that requests it can get access to the object.

DSMs are being implemented due to the constant demand for faster processing and smaller devices. With modern technology shrinking chips and transistors to quantum-mechanical levels, it is natural for other avenues of performance enhancements to be pursued. DSM systems take advantage of networking technology to bring more computing power to machines without making them larger or building the hardware out of single particles rather than transistors.

Another reason these systems exist is for network distribution. The idea is having different locations access the same resources and interfaces. This could help in collaboration for combining the talents of users through a network on the same system. An example of this would be a company that has multiple locations that would need to share the same resources.

Issues of implementing a DSM

Problems in implementing DSMs can include concurrency, latency, network connectivity, and creation. Concurrency is an issue because of the access of memory and resources. The problem with having more processors is that there is more access to resources. There needs to be a check to make sure that there is a consistent system and that it isn't in a bad state.

The DSM has a location manager or a mapping of the data object to which node currently has the object. It must keep track of the data object as it changes nodes so that anytime a request is made for the object it can be found and accessed. There needs to be a minimization of communication when accessing data objects outside of the local system. If a data object is constantly being accessed and switching nodes performance decreases for the entire system. The implementation needs to be able to handle the cost of the communication overhead. The DSM has another issue when a process needs to access data that is not in the local system and it needs multiple pieces that are spread out across various nodes. This system must be able to handle concurrent access to data that is not only in the local node but also in remote nodes.

Memory Coherence

Mechanisms in the DSM are responsible for maintaining memory coherence for the system. There are several ways to maintain the consistency so that the memory stays coherent. Sequential consistency is the result of operations that would appear as being executed in the order specified by the program. General consistency is that all locations of a data object will eventually contain the same information. Processor consistency is where the processors define the order of operations; this can vary if a data object is locked by another process, so it can vary among executions. Immediately after an operation the memory is consistent and data can be accessed normally after all synchronized operations have been made is called Weak Consistency. Release consistency is acquiring the lock doing the operation and then releasing the lock only within the processor.

Issues With Coherence

The DSM must ensure that the location address replicas have the same information across all nodes so that they are able to access data objects upon request. Also, the nodes must access only up to date data objects. This is accomplished by having coherence protocols. One implementation is once a node writes to a data object it invalidates all other copies of the data object so that they are no longer accessible. Another is once a write operation has been done then all other copies are updated for that data object to maintain the memory coherence.

Latency Issues

When it comes to latency, DSM systems can have long wait periods while it looks for the resources it needs. This issue relates to the connectivity of the network because in order to get information from the other nodes in the system, there needs to a strong level connection. With a weak connectivity there is the possibility of losing packets and information, slowing down the whole process. We need to keep things moving because the purpose of a DSM is to increase both the speed and the efficiency of the system. Other latency problems could be related to the hardware in one machine that provides resources for other machines. In these cases issues would include a longer wait time for jobs that are running with DSM or wait for longer access to the memory itself.

Design Issues

The size of the shared memory block or page has issues as well. If it is too large then the page will move around more. If too small then there will be more traffic. The design must also account for pages that are shared, private, read only, and writable. This leads us to the problem of network and I/O latency.

Tradeoffs of Implementation

There are many reasons for implementing a DSM architecture, but the attributes of these implementations come at a certain cost. These “give and takes” are ways to decide how a DSM system

should work for yourself. If there are a large amount of requests for data objects between nodes then performance will decrease overall for the system. This typically comes from either too large of data chunks or too small. The design needs to have some foresight to minimize this constant passing of data objects. Then there is how to keep the data object's most current location. On one side if you keep all nodes up to date about where the current location is then there needs to be that protocol that constantly updates all of the reference to the object. On the other side of it, the DSM would need to invalidate any reference that wasn't current and if some process needed that object they would need to go to the DSM to get the most recent reference.

With a large page size more address space can be used allowing the potential for easy access on the same node as the process and if a page needs to be switched then more information is switched with it allowing fewer page faults. However, with so much data on a page more processes would be requesting the page. A smaller page size would decrease the amount of requests for a page and allowing more operations to get the page they need. The consequent is that there will be more traffic as pages have less data and they need to move to the operation that is requesting them.

One potential issue that cannot be neglected is that the implementation of the DSM architecture requires fast, reliable connections between nodes in order to function well. While this is not normally a problem in the modern environment, it is still something to consider. This would be a poor environment for a DSM if the network was unreliable or had small bandwidth. Another important tradeoff is the consistency of data across the network. Much like accessing a database for reading and writing, it would be difficult to know if data was up to date across the network. This introduction of concurrency issues is something to consider when choosing whether or not to use the DSM implementation. If the system is required to have updated data at a moment's notice, this is not the appropriate implementation, unless writing data kicks all readers out of the system.

When it comes to the motion of memory, we need strategies to prevent conflicts. The use of replication within a distributed system allows each machine to have its own caches that may have several different copies of the same resources. When it comes to replication, however, we want to make sure that we are implementing a system that does not change and write over resources. In a system of replication, we would need to make sure that we do not have a race condition creating an inconsistent state where the read and write resources are fighting to be first. Once a write is committed we want all the caches to be updated, or we want to have the local node check. In these systems that use replication, we really want to have it so that we do not have to run into all these conflicts and generally want this system to have a majority of reading.

The next concept is the use of migration. With this strategy there is only one copy of the resource in the system, and it moves it to where the DSM needs it. In this case we tend to have a system that invokes more writing situations. If this system requires a lot of reads while holding its resource consistency, we can use multi-reader, single-writer algorithms. To make a very consistent

system would be containing a single-reader, single-writer algorithm; however, this would be allowing for sleeps and more latency compared to the multi-reader because of the bottleneck that it creates.

The solutions that these algorithms give can be played in many roles. An example would be a centralized manager algorithm. This is where you have a manager that requests have to go through to obtain memory. This allows for more concurrency so that we do not have any resource issues. A node talks to this manager for a requested segment of memory, which locks the memory to that node and then the manager waits for a response from that machine. This is made faster by having the list of ownership of memory segments held within each node instead of the manager. The decentralized synchronization allows for the local machine to act faster but still be managed through the centralized manager.

A risky implementation of a system is allowing a multi-reader/multi-writer system. This is where every time that a resource is accessed or written, the node has to inform the rest of the nodes. This algorithm can tend to be pretty heavily weighted and needs to have a sequencer for its writes. This sequencer acts as a queue of writes so that it acts in the order that is received and does not allow for race conditions. This is just another strategy that is used in the namespace of DMS. All these different strategies can help serve different purposes in situations necessary for Distributed Shared Memory. Many actual systems use these strategies to solve jobs that are specific to their creators.

Solutions

The problems that DSM solve are the minimization of access time over the network, the maintenance of logical view of distributed memory, and the actual process of distributing data across devices. These problems are solved by a set of algorithms designed to optimize particular areas over others. For example, single-reader/single-writer algorithms have a performance bottleneck at the central server. Multiple-reader/single-writer algorithms face the possibility of writer starvation, but provide much faster access to readers, and it is this implementation that is most widely used. Other solutions can be held within the changes in hardware as well.

Network-wide organization of data is also a problem, both for consistency and for presenting users with a clean, logical layout to work with when accessing information. Each cluster in the network topology, whether that cluster is a group of computers or an individual machine, keeps a directory of where to look for data. There are no real constraints for this implementation, but there are performance issues to consider on the extreme ends. One extreme is to store all table data in a central node that communicates will all others, turning the architecture into one large cluster. This generates a bottleneck at the node holding the table, as data can only be fetched by going through it. The alternative is to store table data at each node, which presents more diversity in implementation. If each node stores the full table, lookups only take two calls at the maximum and lookup queries can

start in any node. However, each node now has a huge amount of data to store and sort through for lookups. The performance issue here is the requirement for both well-organized data and large storage on each machine. The happy medium seems to be organizing the DSM into smaller clusters and storing lookup data in each cluster. This would mean no centralized bottleneck for the entire system and no massive table on every machine, so it seems the most logical implementation.

Dominant Approaches

With distributed shared memory it comes down to one of two strategies. The first strategy is replication. With replication, when a computer needs shared data, it creates a copy of said data in its local resources. This allows the computer to have faster access time for each subsequent access on that data. The second strategy is migration. Each time a computer needs a piece of shared data, the data is moved to the computer's local resources, where the computer can work exclusively on the data. With these strategies, it allows for three different implementations for a distributed shared memory system.

The first implementation is by using software. Some software implementations include Ivy, Mermaid, and Munin. Ivy is a shared virtual memory solution, that can be implemented on current architecture. Like Ivy, Mermaid uses a page-based protocol, however Mermaid is designed to be on a heterogenous system. Finally, Munin is the last software implementation that I will discuss. Unlike the other software implementations, Munin uses multiple consistency protocols and is able to support multiple writers. (13)

Ivy is split between the user-level programs and the operating system support. On the user-level side Ivy has three primitives that can be used by programs. They are process management, memory allocation, and initialization. The operating system primitives are remote operation, and memory mapping. Besides of these primitives, Ivy's address space is split between the shared memory space, and the private memory. Using an invalidation approach, Ivy invalidates all the read-only copies of a page before other processes can write to it. With Ivy it shows that you can implement a shared memory system on an existing system, however it can create a lot of overhead and therefore its performance may not be acceptable. (13)

Mermaid is a first approach to a heterogeneous shared memory system. As with Ivy, Mermaid works on the user-level and is page-based, but Mermaid is unable to handle large amounts of memory. The pages in Mermaid are dynamic as only one data type is allowed per page. Mermaid helps keep the overhead caused by heterogeneity low. Heterogeneity causes the need for data conversion. This is simple for integers, and grows more complex as the object becomes more complex. For user defined data types, it is imperative that you provide your own mapping to the data structure. Therefore, it causes the developer to do extra work so that they can implement Mermaid on a system. (20)(13)

Munin works to make distributed shared memory more viable, by reducing the

communication required in keeping the data consistent. By using the multiple consistency model, it allows Munin to break from Ivy and Mermaid, and instead use multiple writers with multiple consistency protocols. Also unlike the previous versions of the software implementations Munin is not a user-level implementation, but runs at the system level. Depending on the shared-data type, Munin employs different coherence protocols. The biggest limiting factor in Munin, is the overhead from the operating system. (4)

Along with software implementations, there has also been work on hardware implementations. By supporting fine-grain sharing, hardware allows the distributed shared memory system to have improved performance. With systems like Memnet, Dash, and Merlin, hardware systems are able to quickly share among their local connection. Yet for its higher performance, certain hardware implementations are not scalable, and are not cost effective. (13)

Memnet is one of the earlier hardware implementations, with the main goal being to decrease the amount of interprocessor communications. Memnet abstracts the shared memory directly to the network of multiprocessors. Mapping onto the local address space, Memnet splits the memory into the shared memory and the local memory. When a processor needs data that is not currently in its local memory, it sends a request into the shared memory space, and gets the closest valid copy of that data. When a processor needs to write a piece of data, it sends it to every node with a valid copy and updates their values. By sending the data only to each processor with a copy it allows for multiple writes, and thus increases performance. (13) A major problem with Memnet is that it is not scalable, and it's storage directories grow at too high of a rate. (17)

Dash stands for Directory Architecture for Shared Memory. As the name suggests it uses a directory architecture on the scalable multiprocessors network. As with Memnet each have their own local memory caches, however each shared memory block is only shared between every four processors. This allows for more parallelism as each memory block is able to satisfy the requests within itself. When a memory block is not able to satisfy this request, it sends a message to the home cluster and then the system acts accordingly. Dash allows for greater performance, however it requires hardware support for synchronization. (13)

Merlin is the solution to the scalability issues surrounding bus-based systems. Upon each write to memory, Merlin not only updates the local memory, but also sends the update request through the multiprocessor network. However, there are two types of updates for the multiprocessor network; synchronous and rapid. By representing a reflective memory-based interconnection system, Merlin is also able to handle a heterogeneous environment. (13)

Currently there are a number of hybrid implementations. Since neither a software nor hardware approach is perfect, and both require some kind of support from the other, by using a hybrid implementation you can take reduce the cost while still maintaining performance. (13) This makes the hybrid implementation to be the most viable solution to distributed shared memory in

the future. By using lazy hybrid we are able to take the benefits of different protocols, and have significant memory accesses on medium-grained applications.(6)

Insights

When it comes to addressing performance issues that come from memory accesses, it only makes sense to see a distributed solution. However, a distributed solution is not quite as simple as it may seem. Instead it offers up its own set of issues that could be just as bad if not worse than the time it takes to access the memory. This can be seen with the central-server. Yes it is the easiest to implement and can ensure that you are working with the most up to date version. However it can not scale as all the traffic in and out of the central server produces a bottleneck even greater than the memory access.

With the central-server our idea of a distributed system is inside the box. With only a single reader and a single writer the distributed system is instead held in a state quite similar to a commodity machine. Distributed systems are in reality more flexible. By increasing the readers and/or writers the system can be abstracted from a central-server and instead have a network of machines all talking to each other.

Yet even with an abstraction at a software level the system may still suffer from congestion. This is where hardware support comes in. By developing a distributed system at a hardware level the bottlenecks and congestion can all but be eliminated at the network traffic. This also allows greater freedom at the software level, and can greatly increase the system. By having a connected shared address space the system can have increased performance, and if done right increase parallelism.

But just like software implementations, hardware implementations are not perfect for implementing a distributed shared network. By only relying on a hardware implementation we once again think of a distributed system as a commodity machine. This in turn can affect the scalability of the network. It may be near impossible to add or remove a machine from the network without causing serious issues that the system may not be able to come out of.

This leaves us with only one other option in implementing a distributed system, and this is with a hybrid implementation. By thinking outside the box we can bring together the software and hardware implementations. With certain adaptations to hardware we can alleviate the bottlenecks and congestion that remains from software implementations. Together the system can have the scalability that software allows and the performance that hardware demands.

How will the problem space transform in the future

Like all modern computing infrastructure, DSM implementations will be affected by the widespread adoption of quantum computing. Quantum computing is known for being small, fast, and

able to solve integer factorization problems that currently support our best encryption algorithms. Quantum computing will greatly improve the performance of DSM architectures, but there are always tradeoffs, some of which will be outlined here.

Quantum computing replaces the traditional bits in a computer with qubits. Traditional bits are either 1 or 0 based on either a magnetic dipole or stored charge, depending on the storage medium. Qubits are single particles that have a certain spin, termed up or down to represent 1 or 0. What makes quantum computing so effective is the Heisenberg Uncertainty Principle, which allows particles to act as though they are both spinning up and down at the same time. Not only does this allow for greatly improved computational power, it also opens up entirely new ways of solving traditional computing problems. For example, modern hash functions are rendered ineffective by these qubits.

Another important aspect of quantum computing is called quantum cryptography, which deals with the changes in networking and cryptography that quantum computers will prompt. While currently in use only as a key distribution system, the field has generated interest in research that uses quantum entanglement to pass information. Quantum entanglement is, as described by Albert Einstein, “spooky motion at a distance.” Two particles interact physically and share measurable properties, meaning that the state of one particle effects the other, regardless of their proximity to one another. Applying this concept to networking eliminates the need to transmit data over a physical medium, ensuring the security of the data. It also eliminates the need for data to be localized. When the state of one particle in an entangled pair is changed, the other particle exhibits an exactly opposite change at almost exactly the same time. While no speed has been well-defined for quantum entanglement, it has been shown that the connection speed is at least 10,000 times faster than the speed of light. This means there is no real difference between asking the other machines in the rack for data and asking the machines in another datacenter for it. This represents a huge positive for systems implementing the DSM architecture, but it does not solve every problem.

There are several problems with quantum computing, one of which is any physical interaction disrupts the entanglement relied upon by the qubits. If one of the particles in the entangled pair experiences an outside force, the entanglement is broken and another entangled pair would need to be distributed for communication.

Trade-off space and solutions in the future

In order to implement a network based solely on quantum entanglement for communication, all machines would need to be connected by fiber optic cables to distribute the photons. The network would also require a way to reliably distribute entangled pairs to select machines. In our modern network topologies, routers would fill this role well. Instead of passing messages, at the beginning of each conversation, one computer would request a connection to a second computer.

The router would acknowledge the request and distribute the entangled pair, which would then be used by the two computers to communicate directly.

Modern quantum cryptography systems have demonstrated quantum entanglement systems working at up to several hundred meters away. Extrapolating this to the size of a datacenter does not seem to be a huge leap, but over every datacenter seems like a much greater challenge. Assuming that all datacenters are connected by fiber lines, the bottleneck in accessing data from other centers would be the distribution of entangled pairs to each machine. This could be cut down by having each router possess an entangled particle or bank of particles that connect to other routers. This means information would hop from router to router in exactly the same pattern as it does now, but the transmission of data would be instantaneous. This would also be an effective way to implement the system if quantum entanglement cannot be reliably extended to cover the globe.

The development of non-volatile memory using qubits is integral to the widespread acceptance of quantum computing, which is something still being researched. One tradeoff of utilizing quantum computing would be the reliance on older, non-volatile memory storage methods. DSM would benefit immensely if all data could be stored in non-volatile memory that could be accessed quickly, so qubits may not replace common bits if it means losing data every time a machine loses power. Besides that, there is no significant downside to moving to quantum computing and quantum cryptography, especially in a datacenter.

Quantum entanglement models of communicating over long distances instantly makes data localization unnecessary. This means not only can racks communicate instantaneously, but datacenters can as well. Reliable methods of keeping two particles entangled over long distances are still being developed, but the future promises great returns on investment. The increased speed offered by qubits and the instantaneous, lossless, and secure communication between datacenters offered by quantum entanglement will make the DSM architecture much more seamless and efficient.

Conclusions

Distributed shared memory, though not perfect, is the solution to many of our computing problems. In our growing world everyone is dealing with larger and larger sets of data, and our memory cannot keep up with our processors. Whether software, hardware, or a combination of both, distributed shared memory is the future of our devices. By abstracting away the need for all things to be local, the possibility of our devices becomes endless.

Yet distributed systems have their own set of bottlenecks and issues. Without a proper connection between nodes in the system the whole model can go up in smoke. With certain models one node can be a hub (as in the server of a central-server model) which can implode the system if said node goes down.

Distance and latency are two of the biggest issues with systems that rely on distributed shared memory. These variables impact every aspect of a DSM system, and are the most serious bottlenecks that need to be addressed. Because of these bottlenecks many DSM systems are held relatively close to one another so as to minimize them. Hundreds and thousands of computers are grouped together in datacenters all to make use of this simple solution.

But our understanding of the universe is ever changing. The way we look at time and space is mutable. Cause does not have to happen before effect, a request does not have to happen before the process begins. Nor does it mean that two objects are not right next to each other just because they are miles apart. Into the future our whole world of computing technology will change from because of the leaps of quantum computing.

DSM is still young. Though being adopted into many practices and in recent years becoming the core of many different industries such as cloud computing, DSM still has some maturing to do. Until such time that messages can be sent and received instantaneously across vast distances DSM will be held in tight circles across the globe. But for DSM the future looks very bright. Whether the change in technology creates one giant web of memory connecting everyone in their daily lives, or merely makes it easier for users to look up cat pictures. DSM is the inevitable next step in all things computing.

References:

1) Baker, M., Apon, A., Buyya, R., & Jin, H. (2002). Cluster computing and applications. *Encyclopedia of Computer Science and Technology*,*45*(Supplement 30), 87-125.

2) Bell, G., & van Ingen, C. (1998, December 1). *DSM Perspective: Another Point of View.* Retrieved from Microsoft Research: http://www.google.com/url?sa=t&rct=j&q=theoretical%20challenges%20with%20distributed%20shared%20memory&source=web&cd=20&ved=0CGgQFjAJOAo&url=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F69681%2Ftr-98-72.doc&ei=XcshUdqlIeuGyQHh7oGwDQ&usg=AFQjCNFKmWQYHG3n4v

3) Carter, J. B. (1994). *Efficient distributed shared memory based on multi-protocol release consistency.* Retrieved from dpace.rice: http://www.dspace.rice.edu/bitstream/handle/1911/16717/9514164.PDF?sequence=1

4) **Carter, John B.** Design of the Munin Distributed Shared Memory System. *Research Gate*. [Online] [Cited: February 24, 2013.] http://www.researchgate.net/publication/2278762_Design_of_the_Munin_Distributed_Shared_Memory_System

5) Chai, C. (2002). *Consistency Issues in Distributed Shared Memory Systems.* Retrieved from CSE 6306 Advance Operating System: http://crystal.uta.edu/~kumar/cse6306/papers/Chingwen.pdf

6) Cox, A.L.; Dwarkadas, S.; Keleher, P.; Honghui Lu; Rajamony, R.; Zwaenepoel, W., "Software versus hardware shared-memory implementation: a case study," *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on* , vol., no., pp.106,117, 18-21 Apr 1994 doi: 10.1109/ISCA.1994.288157
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=288157&isnumber=7173

7) **Dwarkadas, Sandhya, et al., et al.** Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. CiteSeerX. [Online] [Cited: February 24, 2013.] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.138.7551

8) Ethakota, L. B., & Ganta, M. S. Distributed Shared Memory and Case Study of The Midway Distributed Shared Memory System.

9) Jelica Protic, M. T., & Milutinovic, V. l Distributed Shared Memory: Concepts.

10) Johnson, K. L., Kaashoek, M. F., & Wallach, D. A. (1995). *CRL: High-performance all-software distributed shared memory* (Vol. 29, No. 5, pp. 213-226). ACM.

11) Krzyzanowski, P. (1998). Distributed Shared Memory and Memory Consistency Models. *Rutgers University–CS*, *417*.

12) Lee, C. (2002, February). Distributed Shared Memory. In *Proceedings on the 15th CISL Winter Workshop Kushu, Japan¢ February*.

13) **Li, Kai.** IVY: A Shared Virtual Memory System for Parallel Computing. *Google Books*. [Online] 1988. [Cited: February 24, 2013.] http://books.google.com/books?id=YgXP-sumHu4C&pg=PA121&lpg=PA121&dq=distributed+shared+memory+ivy&source=bl&ots=hl2DFLMIer&sig=nA19_tIGH-405Hu9YJ51A7CkRuI&hl=en&sa=X&ei=CEgpUaufDYTYqQGqs4A4&ved=0CEMQ6AEwAg#v=onepage&q=distributed%20shared%20memory%20ivy&f=false

14) Mellor-Crummey, J. M., & Scott, M. L. (1991). Synchronization without contention. *ACM SIGPLAN Notices*, *26*(4), 269-278.

15) Minnich, R. G., & Farber, D. J. (1993, February). *Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory.* Retrieved from Repository UPenn: http://repository.upenn.edu/cgi/viewcontent.cgi?article=1477&context=cis_reports&sei-redir=1&referer=http%3A%2F%2Fwww.google.com%2Furl%3Fsa%3Dt%26rct%3Dj%26q%3Dtheoretical%2520challenges%2520with%2520distributed%2520shared%2520memory%26source%3Dweb%26cd%3

16) Nitzberg, B., & Lo, V. (1991). *Distributed Shared Memory: A Survey of Issues and Algorithms.* Retrieved from CDF Toronto: http://www.cdf.toronto.edu/~csc469h/fall/handouts/nitzberg91.pdf

17) **Proctic, Jelica, Tomasevic, Milo and Milutinovic, Velijko.** A Survey of Distributed Shared Memory Systems. CS RIT EDU. [Online] 1995. [Cited: February 24, 2013.] http://www.cs.rit.edu/~pns6910/docs/Distributed%20Shared%20Memory%20Systems/A%20s

urvey%20of%20distributed%20shared%20memory%20systems.pdf

18) Protic, J., Tomasevic, M., & Milutinovic, V. (1996). Distributed shared memory: Concepts and systems. *Parallel & Distributed Technology: Systems & Applications, IEEE*, *4*(2), 63-71.

19) Zhang, L., & Parashar, M. Shared Memory Multiprocessors. *Wiley Encyclopedia of Computer Science and Engineering*.

20) **Zhou, Songnian, et al., et al.** Heterogeneous Distributed Shared Memory. *EECG Toronto edu.* [Online] 1991. [Cited: February 24, 2013.] http://www.eecg.toronto.edu/~stumm/papers/Zhou-IEEE-TRPDS91.pdf