Design Aspects of Software Distributed Shared Memory Components and Implementation

A thesis submitted to the University of Hyderabad
in partial fulfillment of the requirements for the award of

Doctor of Philosophy

in

Computer Science

by

R. Sitharamaiah
Reg. No: 06MCPC06

School of Computer and Information Sciences
University of Hyderabad
Hyderabad - 500 046
Andhra Pradesh, India
August 2013

CERTIFICATE

This is to certify that the thesis entitled "**Design Aspects of Software Distributed Shared Memory Components and Implementation**" submitted by **R. Sitharamaiah** bearing Reg. No: **06MCPC06** in partial fulfillment of the requirement for the award of Doctor of Philosophy in Computer Science is a bonafide work carried out by him under my supervision and guidance.

The thesis has not been submitted previously in part or in full to this or any other University or Institution for the award of any degree or diploma.

Dr. Rajeev Wankar
Associate Professor
School of Computer and Information Sciences
University of Hyderabad
Hyderabad - 500 046

Prof. Arun K. Pujari
Dean
School of Computer and Information Sciences
University of Hyderabad
Hyderabad - 500 046

DECLARATION

I, **R. Sitharamaiah,** hereby declare that this thesis entitled "**Design Aspects of Software Distributed Shared Memory Components and Implementation**" submitted by me under the guidance and supervision of Dr. Rajeev Wankar is a bonafide research work. I also declare that it has not been submitted previously in part or in full to this or any other university or institution for the award of any degree or diploma.

Date: 01/08/2013

Name: **R. Sitharamaiah**
Reg. No. **06MCPC06**

Acknowledgement

First of all I express my profound gratitude and deep regards to my supervisor, Dr. Rajeev Wankar. I'm grateful to him for all the motivating discussions right from selection of research problem to timely guidance at every moment. My gratitude towards him is very small when compared to the support and guidance received from him. I also thank Prof C.R. Rao for his innovative research ideas and realization of concepts, which helped my research.

I gratefully acknowledge my DRC members Prof. Arun Agarwal for his continuous monitoring and valuable directions and mid-course corrections in my research progress. I also thank Dr. Atul Negi for his valuable suggestions. I learned many useful insights from their interactions in my DRCs.

I would like to express my sincere gratitude to the Dean, Prof. A.K. Pujari, School of Computer and Information Sciences, for his cooperation.

I am thankful to faculty of our school, whose interactions made my research complete. I also thank the technical and office staff of AI lab and our school for their cooperation.

My special thanks to my friends and research scholars in our school, especially Mr. P.S.V.S.Sai Prasad, Dr. N. Naveen, Mr. B.Vikranth and M.Raghava for their encouragement and support is invaluable, I will forever be thankful to them.

I express my gratitude to Management of CVR College of Engineering, Hyderabad in particular Advisor Prof.C. Mahusudhan Reddy, Hyderabad for granting multiple study leaves and encouragement, which helped in pursuing research. I acknowledge my colleagues for their cooperation.

The past seven years have not been an easy ride, both academically and personally. I truly thank my family members for their constant encouragement. It would not have been possible to write this doctoral thesis without the help and support of the kind

people around me, to only some of whom it is possible to give particular mention here. Finally, I thank all of them.

Hyderabad R. Sitharamaiah

Date: 01/08/2013

Table of Contents

List of Figures

List of Tables

List of Publications

1. Sitharamaiah Ramisetti, Rajeev Wankar, Design of Hierarchical Thread Pool Executor for DSM**,** 2011 Second International Conference on Intelligent Systems, Modeling and Simulation (ISMS2011), Kaula Lumpur, Phnompeng Combodia, January24-28,2011,PP: 284-288.

2. Sitharamaiah Ramisetti, Rajeev Wankar, C.R.Rao, Design of n-gram based Dynamic Pre-fetching for DSM, International Conference on Algorithms and Parallel Processing, September 4-7, Fukuoka, Japan ICA3PP 2012, Part II, LNCS 7440, pp. 217–224, 2012.

3. Sitharamaiah Ramisetti, Rajeev Wankar, Parallel algorithm for n-gram based Dynamic Pre-fetching for DSM. (Ready for the communication).

4. Sitharamaiah Ramisetti, Rajeev Wankar, Component architecture and design issues in HDSM. (Ready for the communication).

Abstract

In the last two decades or so, many researchers paid lot of attention to several aspects of Distributed Shared Memory systems (DSM) varying from theoretical design to implementation of several components. To the best of our knowledge no one has proposed any Unified integrated framework for Heterogeneous DSM. This thesis proposes an integrated abstract framework for HDSM. Several components of it are pluggable/user selectable, which when integrated can become operational and realizable. It has two important components Hierarchical Thread Pool Executer and Dynamic Pre-fetching strategies and its parallelization, which are important contributions of this thesis.

Thread Pool Executor (TPE) is a software component which facilitates the assignment of incoming requests in the form of tasks to the worker threads. General philosophy used in TPE is to discourage more tasks. If still more tasks are arrived TPE throws exception and tasks are rejected. Thread pool executor is an essential component of every DSM and majority of the DSM suffers with this drawback of TPE design.

If we would like to overcome these limitations, we need to have a TPE which is scalable, efficient, and instead of raising exceptions should be able to accept request and migrate it other systems (nodes). *The proposed Hierarchical Thread pool Executor (HTPE)* has several features which will take into account the parameters of the underlying operating system, hardware and dynamic load of nodes to arrive at a thread migration mechanism to exploit and use resources. Several application programs are tested with this proposed HTPE and significant speedup/performance improvement is observed and reported.

Many earlier reported works have shown that data pre-fetching can be an efficient answer to the well-known memory stalls. If one can reduce these stalls, it leads to performance improvement in terms of overall execution time of a given application. In this thesis we propose a new n-gram model based dynamic pre-fetching strategy for prediction, which is based on the concept that if we compute conditional probabilities of the stride sequences of previous n steps and predict the most probable stride. Here

n is an integer which indicates number of grams. The strides that are already pre-fetched are preserved so that we can ignore them if the same stride number is referenced by the program due to principle of locality of reference with the fact that it is available in the memory, hence we need not pre-fetch it. The model also gives the best probable and least probable stride sequences, this information can further be used for dynamic prediction. Experimental results show that the proposed model is far efficient and presents user certain additional input about the behavior of the application. The model flushes once number of miss-predictions exceed pre-determined limit. One can improve the performance of the existing compiler based Software Distributed Shared Memory (SDSM) systems using this model. Several application programs are tested with this proposed Dynamic Pre-fetcher and significant speedup/performance improvement is observed and reported.

The sequential dynamic pre-fetcher discussed above works on each node of the DSM. Since nodes of DSM can be muticore/multiprocessor, it is appropriate to propose a parallel method that can take advantage of this kind of architecture.

Looking into sequential counterpart it can be observed that the process also involves inherent parallelism for computing the stride sequences. Also it was felt that the process of learning phase of the model, where in model remembers the strides being referred can be optimized. Since the parallelism is available at finer level, the most appropriate theoretical model for parallel computation is PRAM model. The parallel algorithm based on this model is presented that shows a theoretical speedup and presented in the thesis.

Chapter 1. Introduction

1.1 Introduction

Over the last two decades many researchers have designed and implemented several categories of Distributed Shared Memory (DSM) or Heterogeneous DSM (HDSM), but to the best of our knowledge no one has proposed a generalized framework for HDSM which covers all the aspects of them. This thesis does not advocate to use any particular implementation or technology for HDSM, but proposes a generalize framework of HDSM from our perspective, and improve the efficiency of few essential components of any DSM/HDSM. It is suggested that if we take these component and plug-in software modules, we can build a reasonable HDSM. Through the journey of this component development, we realized that further study of some of the components like *Hierarchical Thread Pool Executor (HTPE)* and *Pre-fetching strategies* are very important and there is ample scope to improve overall performance (compare to the existing components available in the literature) if they are designed carefully. In this thesis, we propose efficient theoretical design and practical implementations of these two components. The work presented in this thesis is in the larger context of DSM, more specifically with reference to the Heterogeneity in it.

A shared memory system will have main memory created by combining the main memories of all the participating processors, and it is shared by all the processors. This architecture is termed as tightly coupled multi-processors system. Since the memory is common to all the processors, any processor can access any part of the memory. The scalability of such a system is limited by the amount of memory available presently and the maximum memory addressable by the any of the processors.

Shared Memory model is in general a convenient programming model with simple data sharing through a uniform method of reading and writing. The memory is connected to the processors through a shared bus. As a result it suffers from increased

contention and longer latencies in accessing the memory. The contention is due to the fact that the same memory location may be accessed by more than one processor and several processors have to access the memory through the shared bus. Since the bus can be accessed by only one processor at one time, inherent delay occurs in waiting for the bus to get free for further access by any other processor.

Though Shared Memory processing model is popular in parallel programming environment apart from the issues discussed in the last paragraph, it has its limitations: address space provided by the manufacturer and contention for memory. If the application program requires large memory that cannot be supported by the shared memory processor, then we need to divide the problem into several segments and implement the problem on two or more systems. An obvious choice for this problem is, if we are able to provide a single logical address space to the user then the application programmer can develop application logic and execute her/his program on the set of machines that provide the facility. This has further lead to the evolution and realization of DSMs.

1.2 Distributed Shared Memory

A DSM [99] system logically implements the shared memory model on a physically distributed memory system. The DSM system hides remote communication mechanism from the application writer, preserving the programming ease and portability typical of shared memory systems. DSM systems allow relatively easy modification and efficient execution of existing shared memory system applications, which preserves software investments while maximizing the resulting performance.

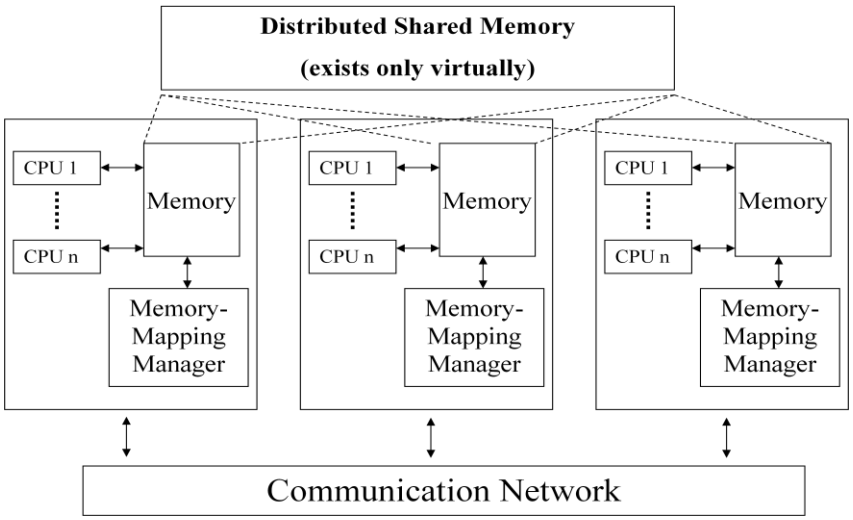


Figure 1.1 A representative diagram showing DSM

A distributed shared memory system (DSM) consists of multiple independent processing nodes with local memory modules, connected by interconnection network, providing a logical address space for all processors. The interconnection network provides the necessary communication abstraction required for the application to run seamlessly. This will alleviate the difficulty of a large program which requires considerable memory, which hitherto cannot be provided by single computer system.

Advantages offered by DSM are ease of programming; portability achieved through shared memory programming paradigm, low cost, better scalability and being free from hardware bottlenecks. The DSM provides an unbounded logical address space for the user application. As a result, the user application need not be aware of implementation details and can focus only on the application rather than spending time in looking at the runtime errors emanating from implementation issues of the underlying hardware and software.

To achieve the ease of programming, cost-effectiveness and scalability, DSM systems logically implement the shared memory model on physically distributed memories. System designers can resort to the specific mechanism in realizing the shared-memory abstraction in hardware or software in different ways. The DSM system abstracts (hides) remote communication mechanisms from application developer, preserving the programming ease and portability that is typical of shared memory systems.

The Distributed Shared Memory systems are classified into three categories: Software Distributed Shared Memory (SDSM), Hardware Distributed Shared Memory and Hybrid Distributed Shared Memory.

The Distributed Shared Memory systems are also classified into: Homogeneous Distributed Shared memory and Heterogeneous Shared Memory. Unless otherwise stated, the DSMs are considered as homogeneous DSMS. The classification is based on the hardware platforms that are chosen for implementing the DSM. If they are non-uniform it is called heterogeneous or if same hardware systems are used for realization, we call this as homogenous DSMs.

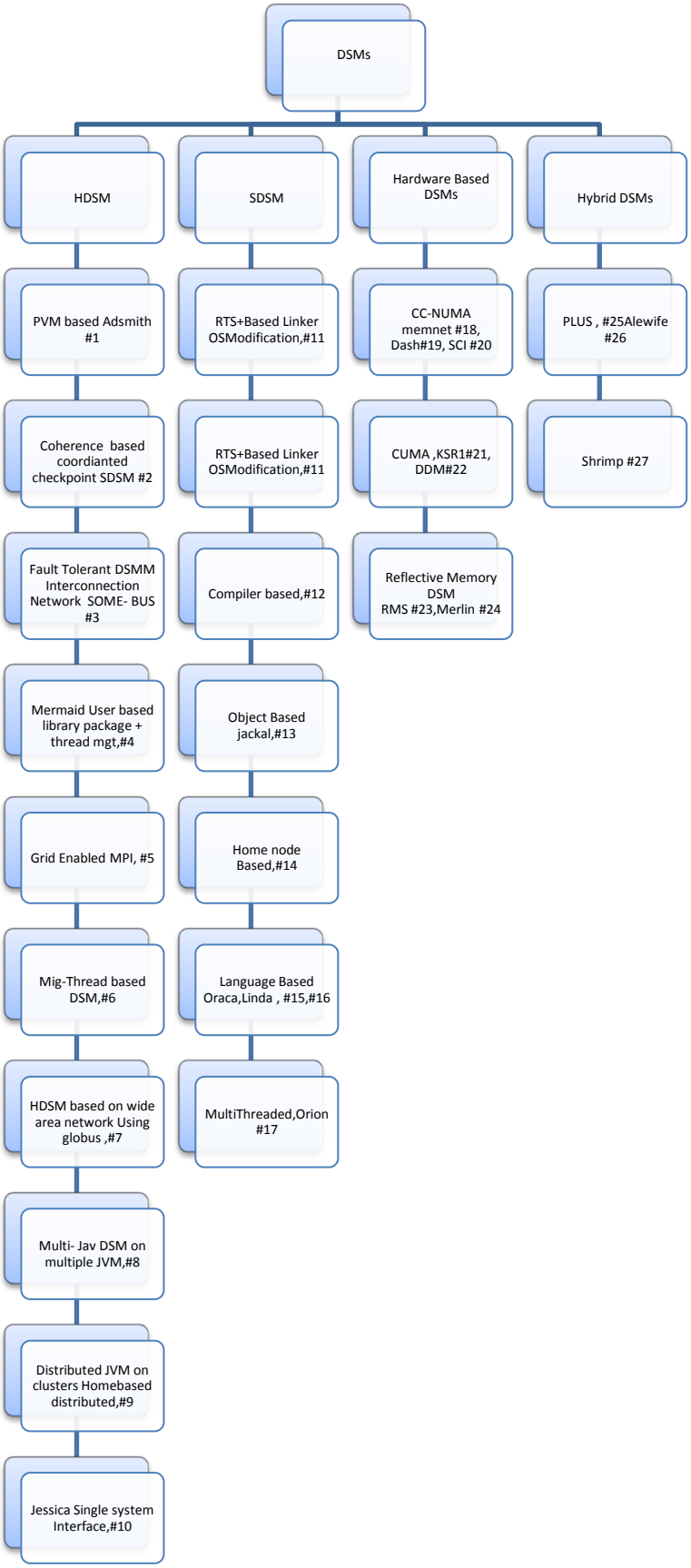


Figure 1.2 DSM taxonomy

1.2.1 DSM taxonomy references:

#1. Ce-Kuen Shieh, An-Chow Lai, Jyh-Chang Ueng, Tyng-Yue Liang, Tzu-Chiang Chang, Su-Cheong Mac, Cohesion: an efficient distributed shared memory system supporting multiple memory consistency models. *PAS '95 Proceedings of the First Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, pp: 146-152, 1995.

#2. Angkul Kongmunvattana, Santipong Tanchatchawal, Nian-Feng Tzeng, Coherence-Based Coordinated Checkpointing for Software Distributed SharedMemory Systems. *ICDCS '00 Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pp: 556, 2000.

#3. Diana Hecht, Constantine Katsinis, Fault-tolerant Distributed-Shared-Memory on a Broadcast-based Interconnection Network. *Parallel and Distributed Processing*, Volume: 1800, pp: 1286-1290, 2000.

#4. Songnian Zhou, Michael Stumm, Tim McInerney, Extending Distributed Shared Memory to Heterogeneous Environments, *10th International Conference onDistributed Computing Systems*, pp: 30-37, 1990 .

#5. Ian Foster, Nicholas T. Karonis, A grid-enabled MPI: message passing in heterogeneous distributed computing systems. *Supercomputing '98 Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp: 1-11, 1998.

#6. Hai Jiang; Chaudhary, V., MigThread: thread migration in DSM systems. *International Conference on Parallel Processing Workshops*, pp:581-588, 2002.

#7. Weisong Shi, Heterogeneous Distributed Shared Memory on Wide Area Network. *IEEE TCCA Newsletter*, pp: 71–80, 2001.

#8. X. Chen , V. H. Allan, MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines. *In Proceedings of the Conference on Parallel and Distributed Processing Techniques and Applications*, pp: 91-98, 1998.

#9. Matchy J.M. Ma, Cho-Li Wang, Francis C.M. Lau, JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing*, Volume: 60, Issue: 10, pp: 1194–1222, October 2000.

#10. Wenzhang Zhu, Cho-Li Wang, Lau, F.C.M., JESSICA2 : a distributed Java Virtual Machine with transparent thread migration support. *IEEE International Conference on Cluster Computing*, pp: 381-388, 2002.

#11. Bennett J. K., J. B. Carter J. B., W. Zwaenepoel, Munin: distributed shared memory based on type-specific memory coherence. *PPOPP '90 Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, ACM New York, NY, USA, Volume: 25, Issue: 3, pp: 168-176, March 1990.

#12. Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, Willy Zwaenepoel, TreadMarks: shared memory computing on networks of workstations. *Journal Computer*, Volume: 29, Issue: 2, pp: 18-28, February 1996.

#13. R.veldema,R.A.F.Bhoedjang,H.E.Bal Jackal, A Compiler Based Implementation of Java for Clusters Of Workstations (2001). *In proceedings of Principles & practice of parallel programming PPOP*, 2001.

#14. Weiwu Hu, Weisong Shi, Zhimin Tang, JIAJIA: A software DSM system based on a new cache coherence protocol. *High-Performance Computing and Networking, LNCS*, Volume: 1593, pp: 461-472, 1999.

#15. Henri E. Bal, Orca: a language for distributed programming. *ACM SIGPLAN*, Volume: 25, Issue: 5, pp: 17-24, May 1990.

#16. Nicholas Carriero, David Gelernter, Linda in context. *Communications of the ACM*, Volume: 32, Issue: 4, pp: 444-458, April 1989.

#17. Ng M. C., Wong W. F., ORION: an dynamic home-based software distributed shared memory system. *Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, pp: 187, 2000.

#18. Delp, G.S., Farber, D.J., Minnich, R.G., Smith, J.M., Tam, M.C., Memory as a network abstraction, *Network, IEEE*, Volume: 5, Issue: 4, pp: 34-41, July 1991.

#19. James, D.V., The Scalable Coherent Interface: scaling to high-performance systems. *Digest of Papers Compcon Spring '94*, pp.64-71, February March 1994.

#20. Daniel Lenoski, Kourosh Gharachorloo, James Laudon J., Anoop Gupta, Hennessy J., Mark Horowitz, Monica Lam, Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. *Compcon Spring '90. Intellect*, pp: 62-67, 1990.

#21. S. Frank, J.Burkardt III, J.Rothnie , The KSRl: Bridging the Gap between Shared Memory and MPPs" Compcon Sping '93, pp. 285-294.,march 1993.

#22. E. Hagersten, K. Landin, and S. Haridi, "DDM-A Cache-Only Memory Architecture," IEEE Computer**,** Vol. 25, No.9, pp. 44-54, Sept. 1992.

#23. Iftode, L., Singh, J. P., Li, K., Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Theory of Computing Systems*, Volume: 31, Issue: 4, pp: 451-473, August 1998.

#24. Maples, Creve, Wittie, L., MERLIN. A superglue for multicomputer systems. *Compcon Spring '90, Intellectual Leverage. Digest of Papers, Thirty-Fifth IEEE Computer Society International Conference*, pp: 73-81, 1990.

#25. Robert Bisiani, Mosur Ravishankar, PLUS: A distributed shared-memory system. *17th Annual International Symposium on Computer Architecture*, pp: 115-124, May 1990.

#26. Anant Agarwal, David Chaiken, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Gino Maa, Dan Nussbaum, Mike Parkin, Donald Yeung, The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor *In Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, 1991.

#27. Sun Microsystems, Inc., External Data Representation Standard: Protocol Specification. *RFC1014 by the ARPA Network Information Center*.

1.3 Heterogeneous Distributed Shared Memory

The Heterogeneous Distributed Shared Memory Systems (HDSMs) are systems that allow different hardware platforms. This is relatively more complex and implementation dependent as the way data is stored and the instruction set and hardware environment is used. Operating systems themselves pose several issues that need to be addressed in realizing HDSM. The other important aspect of HDSM is *process migration* due to the fact that the hardware of the other node, where the process needs to run is not same and the special mechanism of migration needs to take

place. In case of homogeneous SDSM, it is easy to get process to be migrated as the hardware related issues do not exist in it.

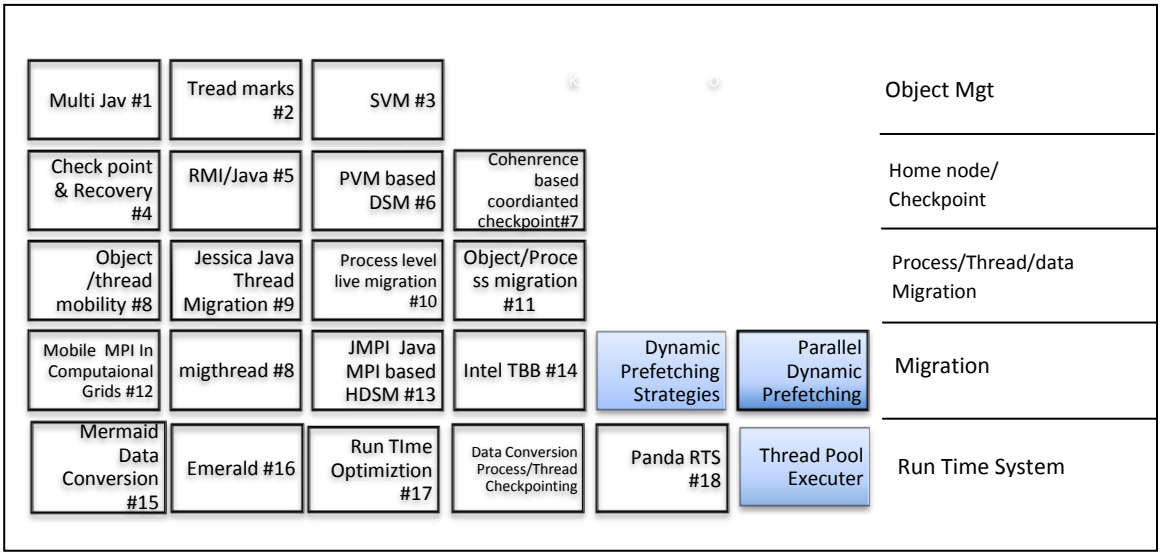


Figure 1.3 HDSM taxonomy

1.3.1 HDSM taxonomy references:

#1. X. Chen , V. H. Allan, MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines. *In Proceedings of the Conference on Parallel and Distributed Processing Techniques and Applications*, pp: 91-98, 1998.

#2. Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, Willy Zwaenepoel, TreadMarks: shared memory computing on networks of workstations. *Journal Computer*, Volume: 29, Issue: 2, pp: 18-28, February 1996.

#3. Songnian Zhou, Michael Stumm, Tim McInerney, Extending Distributed Shared Memory to Heterogeneous Environments, *10th International Conference onDistributed Computing Systems*, pp: 30-37, 1990.

#4. Balkrishna Ramkumar, Volker Strumpen, Portable Checkpointing for Heterogeneous Archtitecturesm. *FTCS '97 Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, pp: 58, 1997.

#5. Jason Maassen, Thilo Kielmann, Henri E. Bal, Efficient replicated method invocation in Java. *JAVA '00 Proceedings of the ACM 2000 conference on Java Grande*, pp: 88-96, 2000.

#6. Jack J. Dongarra, G. A. Geist, Robert Manchek, V. S. Sunderam, Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, pp: 166-175, 1993.

#7. Angkul Kongmunvattana, Santipong Tanchatchawal, Nian-Feng Tzeng, Coherence-Based Coordinated Checkpointing for Software Distributed SharedMemory Systems. *ICDCS '00 Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pp: 556, 2000.

#8. Ian Foster, Nicholas T. Karonis, A grid-enabled MPI: message passing in heterogeneous distributed computing systems. *Supercomputing '98 Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp: 1-11, 1998.

#9. Matchy J.M. Ma, Cho-Li Wang, Francis C.M. Lau, JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing*, Volume: 60, Issue: 10, pp: 1194–1222, October 2000.

#10. Dejan S. Miloji, Fred Douglis, Yves Paindaveine, Richard Wheeler, Songnian Zhou, Eleanor Rieffel, Wolfgang Polak, Process migration. *ACM Computing Surveys*, Volume: 32, Issue: 3, pp: 241—299, September 2000.

#11. Mark Nuttall, Brief survey of systems providing process or object migration facilities. *ACM SIGOPS Operating Systems*, Volume: 28 Issue: 4, pp: 64-80, October 1994.

#12. Ian Foster, Nicholas T. Karonis, A grid-enabled MPI: message passing in heterogeneous distributed computing systems. *Supercomputing '98 Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp: 1-11, 1998.

#13. Kivanc Dincer, jmpi and a Performance Instrumentation Analysis and Visualization Tool for jmpi, *First UK Workshop on Java for High Performance Network Computing*, EUROPAR-98, Southampton, UK, September 2-3, 1998.

#14. Wooyoung Kim and Michael Voss, Multi-core Desktop Programming with Intel TBB, IEEE *Software* ,pp:23-31,January/February 2011.

#15. Daniel J. S., Kourosh Gharachorloo, Design and performance of the Shasta distributed shared memory protocol. *ICS '97 Proceedings of the 11th international conference on Supercomputing*, ACM New York, NY, USA, pp: 245-252, 1997.

#16. Jul, E., Steensgaard, B., Implementation of distributed objects in Emerald. *International Workshop on Object Orientation in Operating Systems*, pp: 130-132, Oct 1991.

#17. R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, C.J.H. Jacobs, H. E. Bal, Source Level Global Optimizations for Fine-Grain Distributed Shared Memory Systems. *ACM SIGPLAN symposium on Principles and practices of parallel programming ,PoPP '01*, Volume: 36, Issue: 7, pp: 83-92, 2001.

#18. Raoul Bhoedjang, Tim Ruhl, Rutger Hofman, Koen Langendoen, Henri Bal, Frans Kaashoek, Panda: a portable platform to support parallel programming languages. *Sedms'93 USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems,* Volume: 4, pp: 11-11, 1993.

1.4 Motivation and Objective

The earlier concentration of research was focused mainly on implementation aspects and interoperability of different systems [41], [43], [47], [48]. To the best of our knowledge, no one has proposed any prototype of integrated system nor did they propose any learning based techniques to speed up the processing of components of DSM. This motivated us to design and implement superior components like Hierarchical Thread Pool Executor and Dynamic Pre-fetching Strategies and its parallelization in this work, and show how they fit into our proposed framework for HDSM.

As part of the research, we have proposed a novel integrated frame work for HDSM. The objective of integrated framework is to identify various component and later required methods of implementation, and to propose a solution for realization and integration of components of it.

We have identified that Thread Pool executor is one of the major component of any DSM implementation. The present thread pool executor is based on the usage of work stealing algorithm [102] and using JMM framework [103]. Some of the proposed modifications to the work stealing can be seen in [104]. Despite these techniques, it does not address some crucial issues like thread blocking.

We have proposed a new Hierarchical Thread Pool Executor which uses non-blocking queues. The incoming requests (tasks) are stored in non-blocking queue which are

sorted on priority. An efficient thread migration mechanism based on min-heap system hierarchy is proposed. If the Thread Pool Executor (TPE) has more worker threads than corePoolSize then the thread migration is initiated which will place the thread on the other system with minimal load. We have integrated this HTPE with standard applications for both performance improvement and implementation, and found that it gives better performance on benchmark programs.

Another very important component of DSM is pre-fetching strategies component. Pre-fetching of objects in DSM will make the object available in cache before it is accessed. One of the classical techniques proposed in the literature is a dynamic software model based pre-fetcher [86]. This will keep track of the objects present in memory and identifies most probable objects and pre-fetches these objects to be later used by the DSM. This technique which is called An Entirely Software and Dynamic Data Pre-fetcher (ESODYP) is based on Markov model.

In this work we propose an *n-gram* model based dynamic Pre-fetcher that cleverly uses the past pre-fetching history to pre-fetch next probable sequences of pages/objects. This newly proposed technique outperforms ESODYP on several benchmark programs we tested. To the best of our knowledge, the use of *n-gram* model for pre-fetching has not been used in DSM's in the past.

Although the proposed dynamic pre-fetcher is sequential in nature, we observed the inherent parallelism in the computation if we change certain parameters without affecting the outcome. This encouraged us to propose a theoretical parallel algorithm of the dynamic pre-fetcher, which is based on the PRAM model-based parallelization which is theoretically superior compare to the sequential counterpart in terms for time.

1.5 Research Problems Related to Objectives

1.5.1 HDSM Framework

Though HDSMs are implemented by different researchers, each of these implementations differs from one another and no commonality exists. We are proposing a layered framework which will be able to visualize and configure various

pluggable modules required for HDSM. The proposed HDSM framework consists of Application Layer, Object Management Layer, Migration and Consistency Layer and Multi-core Operation Layer.

Application layer is the one with which the user interacts with the HDSM system. Object Management Layer deals with the Object Management (creating an object, grouping and managing), which partitions the shared memory according to logical data structures.

Migration and Consistency Layer comprises of two sub layers. Consistency sub layer with Home node Management and checkpoint & restart. Migration sub layer consists of data migration, process migration and Thread migration. Home node Management deals with consistency and data integrity.

Multi-core Operations Layer has two sub layers namely Run Time System Layer and Interface Layer for the Multi-core. The Run Time System (RTS) Layer forms the implementation dependent actions required to be executed at the machine level. Interface Layer for the Multi-core has Network Library, Thread Pool Executor, Dynamic Pre-fetching Strategies, Garbage Collector and multi-Threaded library are modules that will be operational as per the requirement of a user application.

Finally the Operating System and Hardware Layer deals with operating system services which manages hardware and various system services provided by the operating system .The detailed description of framework for HDSM is done in Chapter 6.

1.5.2 Hierarchical Thread Pool Executor (HTPE)

In several DSMs, threads are integral part of DSM implementation (ex: object based DSM) and managing their life cycle is an important issue. The straight forward approach for building a thread based application is to create a new thread each time a request arrives and service the request in the new thread. Instead of creating and invoking a thread every time one can design for a thread pool. A thread pool executor provides a pool of worker threads which will enhance the execution of DSM. We have proposed and implemented a novel hierarchical thread pool executor which will enhance the performance of DSM which uses non-blocking queues (containing ready

threads sorted on priority) and priority principles. This is dealt in detail in Chapter 3 of the thesis.

1.5.3 Dynamic Pre-fetching Strategies:

Pre-fetching is a mechanism that will make the required datum in cache memory before it is used by the processor. This avoids memory stalls which take several mille-seconds to bring required datum into main memory. The proposed dynamic pre-fetching is an entirely software n-gram based model pre-fetching technique. The n-gram model predicts the next accessible stride (a range of addresses analogous to page in operating system terminology) and this is integrated into the DSM modules.

Dynamic pre-fetching exploits the locality of reference by the DSM modules. The locality of reference (for DSM module) says, "*When an application program module is executed on one of the nodes of DSM, the execution is confined for a small period of time around a sequence of instructions (ex: a for loop being executed by a JAVA method)*". The pre-fetching speeds up overall execution of this module as the related strides are available in processor cache due to the fact that the strides are pre-fetched ahead and are available in advance before these are referenced by the module repeatedly due to locality of reference.

This is dealt in detail in Chapter 4 of the thesis.

1.5.4 Parallelization of n-gram Model

We propose a novel parallel n-gram model, which theoretically gives better performance in terms of the parallel time compared to its sequential counterpart. For exploiting the multiple read and random access of the data structure, we use array representations. The space is not a costly matter as only *n* number of pages will be in the main memory of the home node at any given snapshot of the program execution.

Looking into its sequential counterpart, it can be observed that the process of n-gram involves inherent parallelism. Moreover if we are working on a DSM system and many processors are available, the immediate extension of sequential pre-fetching to parallel pre-fetching and can be realized, especially if the nodes of DSM's are

multicores. Since the inherent parallelism is available at finer level of the sequential algorithm, the most appropriate theoretical model for parallel computation is Parallel Random Access Machine Model (PRAM). The parallel algorithm, its implementation of various programs, parallel time and the processor's complexity all are explained in Chapter 5 of the thesis.

1.6 Research Contributions

1. Sitaramaiah Ramisetti, Rajeev Wankar, Design of Hierarchical Thread Pool Executor for DSM**,** 2011 Second International Conference on Intelligent Systems, Modelling and Simulation, pages: 284-288, January 2011. (Scopus Indexed)
2. Sitaramaiah Ramisetti, Rajeev Wankar, C.R.Rao, Design of n-gram based Dynamic Pre-fetching for DSM, International Conference on Algorithms and Parallel Processing ICA3PP-2012, September 4-7, 2012. (DBLP Indexed)
3. Sitaramaiah Ramisetti, Rajeev Wankar, Parallel algorithm for n-gram based dynamic Pre-fetching for DSM. (Ready for Communication).
4. Sitaramaiah Ramisetti, Rajeev Wankar, Component architecture and design issues in HDSM. (Ready for Communication).

1.7 Organization of Thesis

Our work is organized in to 7 chapters. The details are as given below:

Chapter 1 covers Introduction to the Heterogeneous Distributed shared memory and motivation for our research.

Chapter 2 covers extensive literature survey and review of research being carried out in the area of HDSM.

Chapter 3 deals with the proposed Hierarchical Thread Pool Executor for DSM.

Chapter 4 covers in detail the n-gram model based Dynamic Pre-fetching strategies for DSM.

Chapter 5 covers the parallelization of n-gram model and algorithm.

Chapter 6 refers to the proposed component based Heterogeneous Distributed Shared Memory architectural framework.

Chapter 7 covers the conclusion and future work.

Chapter 2. Literature Survey

2.1 Distributed Shared Memory Systems

In the last two decades or so, several researchers paid lot of attention to several aspects of DSM varying from theoretical design to implementation of several components. In this Chapter, we provide brief description of work done by them. The survey concentrates basically on the proposed techniques presented in this thesis.

2.1.1 Software Distributed Shared Memory

IVY [10] is a shared virtual memory system which provides virtual address space shared among all processors in loosely coupled multi-processors. Such a memory can solve many problems in message passing systems on loosely coupled multi-processor and is realized using an Apollo ring network. The experiments with prototype show that parallel programs using a shared virtual memory yield linear speedups. IVY consists of 5 modules namely Initialization, remote operation, memory mapping, memory allocation and process management.

A user mode prototype is implemented on Apollo Domain, an integrated system of work stations and server computers connected by a 12 Mbit /Sec base band, single token ring network. IVY is implemented on top of modified operating system Aegies of Domain environment. The implementation is not particularly efficient but is simple and tractable.

Munin [9] is a DSM system that allows programs written for shared memory multiprocessors to be executed efficiently on distributed memory machines. Munin attempts to overcome the architectural limitations of shared memory machines, while maintaining their advantages in terms of ease of programming. The core of Munin is the runtime library that contains the fault handling, thread support, synchronization and other runtime mechanisms.

Munin supports four consistency protocols conventional, read_only, migratory and w_shared plus a suite of synchronization protocols for locks, barriers and condition

variables. In addition to the consistency protocols provided, Munin has a capability for users to install User written fault handlers and use them. Munin also has a facility for users to create consistency protocols of their own.

Each Munin node interacts with the V kernel to communicate with the other Munin nodes over the Ethernet and to manipulate the virtual memory system as part of maintaining the consistency of shared memory. Each node of an executing Munin program consists of a collection of Munin runtime threads that handle consistency and synchronization operations and one or more user threads performing the parallel computation. Munin programmers write parallel programs using threads as they would on a uniprocessor or a shared memory multiprocessor.

Although the Munin prototype was implemented on the V system (V-kernel), it could be made to run on any operating system that allows user programs to manipulate its own virtual memory mappings to handle page faults at user level, create and destroy processes on remote nodes exchange messages with remote processes and access uni-processor synchronization support P and V.

Munin is unique in its use of loosely coherent memory, based on the partial order specified by a shared memory parallel program, and in its use of type-specific memory coherence. Instead of a single memory coherence mechanism for all shared data objects, Munin employs different mechanisms, each appropriate for a different class of shared data object. These type-specific mechanisms are part of a runtime system that accepts hints from the user or the compiler to determine the coherence mechanism to be used for each object.

Adaptive Software Cache Management for Distributed Shared Memory Architectures*:** Discusses an adaptive cache coherence mechanism that exploits semantic information about the expected or observed access behavior of particular data objects [27]. Munin employs several different mechanisms, each appropriate for a different category of shared data object, the technique of providing multiple coherence mechanisms is called as ***adaptivecaching. Adaptivecaching maintains coherence based on the expected **or** observed access behavior of each shared object and on the size of cached items.

PLUS[29] is a multiprocessor architecture tailored to the fast execution of a single multithreaded process. Its goal is to accelerate the execution of CPU-bound applications. PLUS support both shared memory and efficient synchronization. Memory access latency is reduced by non-demand replication of pages with hardware-supported coherence between replicated pages. The architecture has been simulated in detail, and some of the key measurements that have been used to substantiate the architectural decisions are presented. PLUS was aimed at efficiently executing a single multithreaded process by using distributed memories, hardware supported memory coherence and synchronization mechanisms.

The implementation of PLUS uses a general purpose Motorola 88000 processor (25MHz) with 32 Kbytes of cache and 8 or 32 Mbytes of main memory at each node. The memory was organized in two interleaved banks to sustain the burst bandwidth needed for cache line accesses. Global memory mapping, coherence management and atomic operations were performed by a hardware module that is implemented with Xilinx PLD's and PAL'S. In this implementation, each node can have up to 8 writes and 8 delayed operations in progress. The interconnection network uses a mesh router designed at Caltech**.** Each router has five pairs of ***U0*** links: one for the processor and one for each of its mesh neighbors. Links operate at 20 Mbyte in each direction. **SCSI** devices, audio peripherals and host computers can be attached to each node. Caching and issues such as address space structure and page replacement schemes were used to define taxonomy.

A taxonomy-based comparison of several distributed shared memory systems is given in [49]. Based on the taxonomy, three DSM efforts were examined in detail, namely: IVY, Clouds and MemNet.

CRL: high-performance all-software distributed shared memory [19**]:**The C/Region Library (CRL) is an all-software distributed shared memory (DSM) system. CRL requires no special compiler, hardware, or operating system support beyond the ability to send and receive messages. It provides a simple, portable, region-based shared address space programming model that is capable of delivering good performance on a wide range of multiprocessor and distributed system architectures.

Parallel applications built on top of CRL share data through regions. Each region is an arbitrarily sized contiguous area of memory. The programmer defines regions and

delimits accesses to them using annotations. Regions are cached in the local memories of processors; cached copies are kept consistent using a directory-based coherence protocol. CRL achieves speedups better than that of those provided by Alewife's native support for shared memory, even for challenging applications (e.g., Barnes-Hut) and small problem sizes.

Distributed Shared Memory: Where We Are and Where We Should Be Headed [20]. This work reflects on: what have been the major advances in the area, what the important outstanding problems are, and what work needs to be done. Finally, it also reflects on a modest step towards solving these problems, namely the Quarks DSM system. The umbrella of the Computer Systems Laboratory at Utah has developed a distributed shared system named Quarks3. Quarks consists of a user-level library and associated header files that support DSM on collections of workstations. Quarks was built on SunOS 4.1, HP BSD 4.3, and HPUX; native Mach and IRIX 5.2. Quarks includes a number of modern DSM features such as support for multiple consistency protocols within an application on a per page basis (e.g., a write invalidate protocol providing strict consistency, a delayed write update protocol providing release consistency, etc.). An effort was made to make adding new protocols easy, allowing the research community to experiment with new protocols and compiler writers to develop specialized protocols.

Cohesion[51] is a prototype of DSM which supports two memory consistency models, namely Sequential consistency and Release consistency, within a single program to improve the performance and supports wide-variety of parallel programs for the system. Memory that is sequentially consistent is further divided into object-based and conventional (page-based) memory; where they are constructed in user-level and kernel-level, respectively. In object-based memory, the shared data is kept consistent at the granularity of an object; it is provided to improve the performance of the fine-grained parallel applications that may incur a significant overhead in conventional or release memory, as well as to eliminate unnecessary movement of the pages which are protected in a critical section.

On the other hand, the Release consistency model is supported in Cohesion to attack the problem of excessive network traffic and false sharing. Cohesion programs are

written in C++, and the annotation of shared objects for release and object-based memory is accomplished by inheriting a system provided base class.

SCASH [2,3] is a software based DSM using PM (MPICH on PM using a cluster of systems employing myrinet switch) low-latency and high bandwidth communication library for a Myrinet gigabit network and memory management functions, such as memory protection, supported by an operating system kernel. It is implemented as a user level runtime library. To obtain high performance communication and support multi-user environments, it was co-designed PM and the run-time routine for a programming language. Several unique features, e.g., network context switching and a *Modified ACK/NACK* flow control algorithm, were provided for PM. This could be used as a platform for implementing SDSM or any other parallel applications.

MPC++ version2 Template library, called Multiple Threads Template Library (MTTL), supports local and remote thread invocation, synchronization structures and global pointers. The global pointers allow access to remote objects and its distribution to multiple processors. Such a facility is useful to implement data parallel applications in which processor's local memory is often distributed to other processors.

SCASH is a page based DSM, where consistency is maintained per page basis, utilizes Release Consistency model with multiple writer protocol. It also utilizes dynamic home node mechanism realized using concepts of diff and write invalidate and update and memory barrier synchronization mechanism.

The *Home* node of a page is the node that keeps the latest data of the page and the page directory which represents the set of nodes sharing the page. A node, intending to share a page, must copy the latest data of the page found on the ***home*** node. When the memory barrier synchronization is issued, nodes, having modified a shared page, must notify the home node of the difference between the former and the new page data. The home maintains the latest page data by applying all the differences received from those nodes.

The ***base*** is the node that knows the latest home node. All nodes know the base nodes of all pages. Even if a node does not know the home of a page, the home can be ascertained by asking the base node. Using **SCASH home selection algorithm,** the home node is moved to another node which has modified a large part of the page,

when and only when a home node does not write to the page and other nodes which share the page do write to the page. SCASH employs dynamic home node allocation.

An interesting work on Compiler and Software Distributed Shared Memory Support for Irregular Applications [8] was done by Honghui Lut, Alan L.Cox, Sandhya Dwarkadas. A modified version of TreadMarks[7] that supports prefetching and aggregation is used for analysis and performance measure of two irregular applications were carried in this work. They used another method used for comparison that use Parascope, parallel programming environment to carry out the required compiler analysis. The performance results for two irregular applications, moldyn (Moldyn is a molecular dynamics simulation) and nbf(NBF is the kernel of *a* molecular dynamics simulation) were compared.

Comparative evaluation of latency tolerance techniques for software distributed shared memory is proposed in [21]. A key challenge in achieving high performance on software DSMs is overcoming their relatively large communication latencies. Two techniques which address this problem: prefetching and multithreading. While previous studies have examined each of these techniques in isolation, in this work, these techniques were evaluated using a consistent hardware platform and set of applications, thereby allowing direct comparisons.

These were studied combining prefetching and multithreading in a software DSM and experiments on real hardware using a full implementation of both techniques. Experimental results demonstrate that both prefetching and multithreading result in significant performance improvements when applied individually. In addition, it was observed that prefetching and multithreading can potentially complement each other by using prefetching to hide memory latency and multithreading to hide synchronization latency.

JIAJIA [1] is a DSM based on Scope consistency and employing home node. The JIAJIA utilizes scope consistency and uses multiple writer protocol to update the data located in page. Each shared page has a fixed global address, which determines the home of the page. Initially, a page is mapped to its global address only by its home processor. The scope consistency used by JIAJIA gives better performance than lazy release consistency used by several other DSMs. The home nodes are maintained and the home node address is a global address space and is mapped using mmap() system

service. The referred address is first checked in the local home node. If it is not found, then the remote home pages are checked on other processor and it is acquired by using global address of the shared page. Reference to a non-home page causes the delivery of SIGSEGV signal. The SIGSEGV handler then maps the fault page to the global address of the page in local address space. JIAJIA uses fixed memory consistency model and fixed write propagation (write invalidate) strategy. Multiple write technique is used to reduce false sharing.

Since it uses scope consistency, global address space and page level mapping, the message passing is minimized which in turn gives better performance for applications such as TSP, Water and Barnes. JIAJIA can solve large problems that cannot be solved by other DSMs due to memory size limitation. JIAJIA gives better performance than CVM for many tested applications. This is one of the few home based DSM, hence gives better performance.

ORION [11] is multithreaded software distributed shared memory (DSM) system. It was developed to provide POSIX-thread (pthread) like interface. This will avoid creating another unique set of application programming interface and ease the porting of pthread programs to a distributed environment. Orion implements home-based Memory Release consistency model. It also employs two adaptive schemes for home-based DSM systems, namely home migration and dynamic adaptation between write-invalidation and write-update protocols.

An Orion program typically starts with a number of nodes or hosts specified by the user. Depending on the number of nodes specified, the master node that is ranked 0 remotely spawns the same program on other nodes and assigns them a different rank each. The Data Processing Interface(DPI) library undertakes the task of automatic remote process spawning during start up. DPI is a separate network communications library to provide the basic send, buffered send and receive functions found in MPI. Some group communication functions like barrier, all-gather and other functions are provided too.

Orion implements home-based eager release consistency (HERC) supporting the multiple writer, write-invalidate and write-update protocols. Typically in an Orion application, user codes interfaces with the Orion API, which in turn makes use of functions from pthread and *data passing interface.*

Implementation of a Software Distributed Shared Memory (DSM) over Virtual Interface Architecture (VIA) for a Linux-based cluster of PCs [17] is implemented and its performance evaluation is done. VIA is a user-level memory-mapped communication model that provides zero-copy communication and low-overhead by excluding the operating system kernel from the communication path. The VI architecture is composed of four basic components: Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers. The DSM protocol implemented on VIA is Home-based Lazy Release Consistency (HLRC) that previous studies have shown to exhibit good scalability by reducing the number of messages and memory overhead compared to the homeless counterpart.

A Distributed Shared Memory (DSM) system provides a distributed application with a shared virtual address space. Choosing a memory consistency model is one of the main decisions in designing a DSM system. While Sequential Consistency provides a simple and intuitive programming model, relaxed consistency models allow memory accesses to be parallelized, improving runtime performance. The home-based lazy release consistency (HLRC) protocol was implemented such that it supports preemptive multithreading and compare its performance with the efficient multithreaded SC protocol in a comparison of sequential consistency with home-based lazy release consistency for software distributed shared memory**.**

The runtime performance of two memory coherence protocols [42] a multithreaded implementation of the HLRC model and an efficient multithreaded implementation of the SC model that uses a MultiView memory mapping technique were compared. Comparison was carried on the same test bed environment with benchmark suite and investigated the effectiveness and scalability of both these protocols. Three consistency protocols with four sizes of coherence granularity were tested: SC, single writer LRC, and HLRC. The results show that no single combination of protocol and granularity performs best for all the applications. A combination of the SC protocol and fine granularity performed well with 7 of the 12 applications.

Home migration is used to tackle the home assignment problem in home-based software distributed shared memory systems. An adaptive home migration protocol was proposed **[54]** to optimize the single-writer pattern which occurs frequently in distributed applications. This approach is unique in its use of a per-object threshold

which is continuously adjusted to facilitate home migration decisions. This adaptive threshold is monotonously decreasing with increased likelihood that a particular object exhibits a lasting single-writer pattern. The threshold is tuned according to the feedback of previous home migration decisions at runtime.

The adaptive home migration protocol was implemented in a distributed Java virtual machine that supports truly parallel execution of multithreaded Java applications on clusters. The analysis and the experiments show that home migration protocol demonstrates both the sensitivity to the lasting single-writer pattern and the robustness against the transient single-writer pattern.

A comparison between homeless and home-based Lazy Release Consistency (LRC) protocols which are used to implement Distributed Shared Memory (DSM) in cluster computing was done in Homeless and home-based Lazy Release Consistency protocols on Distributed Shared Memory **[58].** A performance evaluation of parallel applications running on homeless and home-based LRC protocols was performed. Based on the performance comparison between Tread-Marks, which uses homeless LRC protocol, and home-based DSM system, it was shown that the home-based DSM system had better scalability than TreadMarks in parallel applications. This poor scalability in the homeless protocol is caused by a hot spot and garbage collection, but it was shown that these factors do not affect the scalability of the home-based protocol.

Some of the challenges present in providing support for OpenMP applications on a Software Distributed Shared Memory(DSM) based cluster system was carried in the work **[28]** Towards OpenMP Execution on Software Distributed Shared Memory Systems. The Treadmarks Software DSM system used as reference DSM for testing OPENMP programs wherein, shared memory sections can be allocated on-request. OpenMP workshare constructs are translated in the usual method, by modifying lower and upper bound of the loops according to the iteration space assigned for each participating process. Static scheduling was only supported. All parallel constructs are placed between a pair of barrier synchronizations. The barriers perform the dual functions of synchronization and maintaining coherence of shared data. The translation of serial program sections became non-trivial. The need to maintain correct

control flow precludes the possibility of executing the serial section by only the master process.

Detailed measurements of the performance characteristics of realistic OpenMP applications from the SPEC OMP2001 benchmarks were used for performance monitoring. The pitfalls of a naive translation approach from OpenMP into the API provided by a Software DSM system, and arrive at a set of possible program optimization techniques were proposed to overcome these.

2.1.2 Compiler Based SDSMs

A simple model of *shared data-objects,* which extends the abstract data type model to support distributed programming is **Orca [**33,35]. The model essentially provides shared address space semantics, rather than message passing semantics, without requiring physical shared memory to be present in the target system. Based on this, a new programming language Orca, based on shared data-objects is proposed and implemented. A compiler and three different run time systems for Orca are available.

The distribution of objects among the participating processors is left entirely to the implementation. This decision significantly contributes to the simplicity of the language. The design of Orca allows the compiler and run time system to deal efficiently with the distribution of objects. There are no global objects (objects have to be passed as parameter), so the run time system can keep track of which processes can access which objects. The semantics of an operation invocation do not depend on whether the object and the invoker are on the same or different processors. This location independence makes it possible to move objects dynamically. Shared data can only be accessed via a well-defined set of operations. This enables the system to dynamically *replicate* objects. An operation can access only a single object, allowing indivisible operations to be implemented without using a complicated locking or version management scheme.

Orca is also a language for implementing parallel applications loosely coupled distributed systems. Unlike most languages for distributed programming, it allows processes on different machines to share data. Such data is encapsulated in data objects, which are instances of user-defined abstract data types. The implementation

of Orca takes care of the physical distribution of objects among the local memories of the processors. In particular, an implementation may replicate and/or migrate objects in order to decrease access times for objects and increase parallelism.

Orca is intended for distributed applications programming rather than systems programming, and is therefore designed to be a simple, expressive and efficient language with clean semantics. Processes in Orca can communicate through shared data, even if the processors on which they run do not have physical shared memory. The main novelty of proposed approach is the way access to shared data is expressed. Unlike shared physical memory (or distributed shared memory**),** shared data in Orca are accessed through user-defined high-level operations.The implementation of Orca's model takes care of the physical distribution of shared data among processors.

Current parallel programming languages require advanced run-time support to implement communication and data consistency. As such, run-time systems are usually layered on top of a specific operating system, they are non portable.

Panda [36] is a portable virtual machine that provides general and flexible support for implementing run-time systems for parallel programming languages. Panda has two interfaces: a Panda interface, providing threads, RPC, and totally-ordered group communication, and a system interface which encapsulates machine dependencies by providing machine-independent thread and communication abstractions. The interfaces were integrated with Unix implementation, and provide a portable and scalable run-time system for the Orca parallel programming language on top of Panda.

TreadMarks [7]: A DSM system consisting of N networked workstations, each with its own memory. The DSM software provides the abstraction of a globally shared memory, in which each processor can access any data item without the programmer having to worry about where the data is or how to obtain its value.

TreadMarks provides shared memory as a linear array of bytes via a relaxed memory model called release consistency. The implementation uses the virtual memory hardware to detect accesses, but it uses a multiple-writer protocol to alleviate problems caused by mismatches between page size and application granularity. TreadMarks runs at the user level on Unix workstations without kernel modifications

or special privileges and with standard Unix interfaces, compilers, and linkers. TreadMarks implements inter machine communication using UDP/IP through the Berkeley sockets interface. Since UDP/IP does not guarantee reliable delivery, TreadMarks uses lightweight, operation-specific, user level protocols to ensure message arrival. Every message sent by TreadMarks is a request or a response.

Shasta[4,5] is compiler based software distributed shared memory(SDSM). Shasta supports a shared address space in software across a cluster of computers with physically distributed memory. A unique aspect of Shasta compared to most other software distributed shared memory systems is that shared data can be kept coherent at a fine granularity. Shasta achieves this by inserting inline code that checks the cache state of shared data before each load or store. In addition, Shasta allows the coherence granularity to be varied across different shared data structures in a single application. This approach alleviates potential inefficiencies that arise from the fixed large (page-size) granularity of communication typical in most software shared memory systems.

Shasta incorporates compilation checks of data before each load or store in an application. The address space is divided into lines and blocks matching with the cache lines and multiples of cache lines. The system is so organized that the block size and address generation differs for different address ranges. It also maintains a compilation checks for the loads and stores and as such this will lead to better performance. It employs exclusive shared and dirty modes of a block and the coherence is maintained with the help of these states. However, allowing processors to share memory within the same SMP is complicated by race conditions that arise because the inline state check is non-atomic with respect to the actual load or store of shared data. A novel protocol that avoids such race conditions without the use of costly synchronization in the inline checking code.

The protocol also includes optimizations such as non-blocking stores that aggressively exploit a relaxed memory consistency model. Other optimizations include detection of migratory data sharing, issuing multiple load misses simultaneously, merging of load and store misses to the same cache line, and support for prefetching and home placement directives.

Since Shasta supports shared memory entirely in software, it provides considerable flexibility in managing coherence granularity and applying protocol optimizations. The ability to support multiple coherence granularities within a single application is by far the most unique and important feature of Shasta, leading to performance improvements of as high as two times.

Compiler based prefetching for pointer-based applications in particular, those containing recursive data structures was studied **[30]** in Compiler-based prefetching for recursive data structures. Software-controlled data prefetching offers the potential for bridging the ever-increasing speed gap between the memory subsystem and today's high-performance processors. While prefetching has enjoyed considerable success in array-based numeric codes, its potential in pointer-based applications has remained largely unexplored. The fundamental problem in prefetching pointer-based data structures was analyzed and a guideline for devising successful prefetching schemes was proposed. Based on this guideline, designed three prefetching schemes, automated the most widely applicable scheme (*greedy prefetching*) in an optimizing research compiler, and evaluated the performance of all three schemes on a modern superscalar processor similar to the MIPS R10000.

These results demonstrate that compiler inserted prefetching can significantly improve the execution speed of pointer-based codes as much as 45% for the applications, compared with the only other compiler-based pointer prefetching scheme in the literature.

Cache-coherent multiprocessors with distributed shared memory are becoming increasingly popular for parallel computing. However, obtaining high performance on these machines requires that an application execute with good data locality. In addition to making effective use of caches, it is often necessary to distribute data structures across the local memories of the processing nodes; thereby reducing the latency of cache misses **[25].** A set of abstractions for controlling loop scheduling and data distribution on CC-NUMA architectures and the implementation of these abstractions within a production compiler were taken up. The unique aspects include extensive error detection features, support for separate compilation across multiple files, optimization techniques to generate efficient code for distributed arrays, and tight integration of data distribution optimizations with other loop-level optimizations

in the compiler. Networks of workstations offer inexpensive and highly available high performance computing environments.

In the work Compiler and Runtime Support for Adoptive Load Balancing in Distributed Shared Memory **[32],** the compiler provides pattern information to the run-time at the points in the code that will be executed in parallel. The run-time uses these points to gather statistics on available computational and communication resources. Based on the access patterns across phases as well as on available computing power, the runtime can then make intelligent decisions not only to distribute the computational load evenly, but also to maximize locality (based on current processor caching) and minimize communication overhead in the future. The result is a system that uniquely combines compile-time and run-time information to adapt both to changes in access patterns as well as to changes in computational power in order to reduce execution time. The TreadMarks DSM is used as a reference of study.

A critical issue for achieving good performance in any parallel system is load balancing, even more so in workstation environments where the machines might be shared among many users. This work evaluate a system that combines compiler and run-time support to achieve load balancing dynamically on software distributed shared memory programs. The information provided by the compiler is used to help the run-time system distribute the work of the parallel loops, not only according to the relative power of the processors, but also in such a way as to minimize communication and page sharing.

Jackal [18] is a Compiler Based Implementation of Java for Clusters of Workstations. Jackal is a compiler-driven distributed shared memory implementation of the Java programming language. JACKAL's goal is to efficiently execute (unmodified) multithreaded Java programs on a cluster of workstations. Jackal consists of a native Java compiler and a runtime system that implements a distributed shared memory protocol for variable sized memory regions. The Jackal compiler stores Java objects in shared regions and augments the programs. It compiles with access checks; these access checks drive the memory consistency protocol. The Jackal compiler implements several optimizations to reduce the overhead of these software access checks.

This paper describes and evaluates the use of aggressive static analysis in Jackal; a fine-grain Distributed Shared Memory (DSM) system for Java is presented in Source-level global optimizations for fine-grain distributed shared memory systems [22]. Jackal uses an optimizing, *source-level* compiler rather than the binary rewriting techniques employed by most other fine-grain DSM systems. Source-level analysis makes existing access check optimizations (e.g., access-check batching) more effective and enables two novel fine-grain DSM optimizations: *object-graph aggregation* and *automatic computation migration*. The compiler detects situations where an access to a *root object* is followed by accesses to sub objects.

Jackal attempts to aggregate all access checks on objects in such *object graphs* into a single check on the graph's root object. If this check fails, the entire graph is fetched. Object-graph aggregation can reduce the number of network roundtrips and, since it is an advanced form of access-check batching, this improves sequential performance.

Computation migration (or function shipping) is used to optimize critical sections in which a single processor owns both the shared data that is accessed and the lock that protects the data. It is usually more efficient to execute such critical sections on the processor that holds the lock and the data than to incur multiple roundtrips for acquiring the lock, fetching the data, writing the data back, and releasing the lock. Jackal's compiler detects such critical sections and optimizes them by generating single-roundtrip *computation migration* code rather than standard data shipping code. Jackal's optimizations improve both sequential and parallel application performance. On average, sequential execution times of instrumented, optimized programs are within 10% of those of uninstrumented programs. Application speedups usually improve significantly and several Jackal applications perform as well as hand optimized message-passing programs.

This paper focuses on Jackal's run-time system [53] which implements a multiple-writer, home-based consistency protocol. Protocol actions are triggered by software access checks that Jackal's compiler inserts before object and array references. To reduce access-check overhead, the compiler exploits source-level information and performs extensive static analysis to optimize, lift, and remove access checks. Optimizations for Jackal's run-time system, which mainly consist of discovering

opportunities to dispense with flushing of cached data was dealt in detail. Performance results for different run-time optimizations were carried and compared their impact with the impact of one compiler optimization. It was found that run-time optimizations are necessary for good Jackal performance, but only in conjunction with the Jackal compiler optimizations [18].

2.1.3 Hardware Distributed Shared Memory Systems

The **Stanford dash** multi-processor combines programmability of shared memory machines with capability of message parsing [6]. Dash provides a scalable shared memory machine with hardware coherent caches by using distributed directory based cache coherence protocol. Physical memory is distributed among nodes. Each processor node or cluster consists of small number of high performance processors with individual caches and part of shared memory connected by snooping bus. Dash supports release consistency model in hardware. Dash memory can be logically broken to four levels of hierarchy, processor cache level, local cluster level directory home level and physical memory level. It employs directory based protocol to access memory at all these levels through hardware maintenance of dirty bits and status flags.

The prefetch in dash bring the data in remote access cache. A subsequent access will then be serviced by cache. The Dash provides a scalable distributed shared memory at hardware level. Since the memory fetches and the access and cache coherence is achieved through hardware except for the message passing required at application level, the system is fast and leads to high performance and throughput. Dash is one of the early realized hardware DSMs.

Software versus hardware shared-memory implementation: In a case study[26] Comparing the performance of software-supported shared memory on a general-purpose network to hardware-supported shared memory on a dedicated interconnect, the authors compare a software implementation on a general-purpose network of uniprocessor nodes, a hardware implementation using a directory-based protocol on a dedicated interconnect, and a combined implementation using software to provide shared memory between multiprocessor nodes with hardware implementing shared memory within a node.

Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors [23] present a theoretical framework for automatically partitioning parallel loops to minimize cache coherency traffic on shared-memory multiprocessors. While several previous papers have looked at hyper plane partitioning of iteration spaces to reduce communication traffic, the problem of deriving the optimal tiling parameters for minimal communication in loops with general affine index expressions has remained open. This open problem was solved by presenting a method for deriving an optimal hyper parallelepiped tiling of iteration spaces for minimal communication in multiprocessors with caches. The same theoretical framework was used to determine optimal tiling parameters for both data and loop partitioning in distributed memory multi-computers. This framework uses matrices to represent iteration and data space mappings and the notion of uniformly intersecting references to capture temporal locality in array references.

Cashmere-2L [16] is a Low-latency remote-write network, such as DEC's Memory Channel, which provides the possibility of transparent, inexpensive, huge-scale shared-memory parallel computing on clusters of shared memory multiprocessors (SMPs). It had the advantage of hardware shared memory for sharing within an SMP and to ensure that software overhead is incurred only when actively sharing data across SMPs in the cluster. **CashmereSL** uses hardware to shared memory within a node, while exploiting the Memory Channel's remote-write capabilities to implement "moderately lazy" release consistency with multiple concurrent writers, directories, home nodes, and page-size coherence blocks across nodes. Cashmere, Cashmere-2L were implemented on a node, 32-processor DEC Alpha server system. Speedups range from 8 to 31 on 32 processors with standard benchmark suite, depending on the application's characteristics.

Multiprocessor system-on-chip (MP-SoC) platforms represent [31] an emerging trend for embedded multimedia applications. To enable MP-SoC platforms, scalable communication-centric interconnect fabrics, such as networks-on-chip (NoCs), was planned. The shared memory represents one of the key elements in designing MP-SoCs to provide data exchange and synchronization support.

This paper focuses on the energy/delay exploration of distributed shared memory architecture, suitable for low-power on-chip multiprocessors based on NoC. A

mechanism is proposed for the data allocation on the distributed shared memory space, dynamically managed by an on-chip hardware memory management unit (HwMMU). Moreover, the exploitation of the HwMMU primitives for the migration, replication, and compaction of shared data is discussed.

Experimental results show the impact of different distributed shared memory configurations for a selected set of parallel benchmark applications from the power/performance perspective.

2.1.4 Hybrid Distributed Shared Memory Systems

Blizzard [12] supports fine-grain memory access control, which selectively restricts reads and writes to cache-block-sized memory regions. Fine-grain access control forms the basis of efficient cache-coherent shared memory. Blizzard focuses on low-cost implementations that require little or no additional hardware. These techniques permit efficient implementation of shared memory on a wide range of parallel systems, hereby providing shared-memory codes with a portability previously limited to message passing.

Blizzard incorporates three techniques that require no additional hardware into Blizzard, a system that supports distributed shared memory on the CM-5. The first addition is software lookup before each shared-memory reference by modifying the program's executable. The second uses the memory's error correcting code (ECC) as cache-block valid bits. The third is a hybrid. The software technique ranged from slightly faster to two times slower than the ECC approach. Blizzard's performance is roughly comparable to a hardware shared-memory machine.

Blizzard, a Distributed shared memory (DSM) systems and proposed shared-memory machines **[13]** have implemented some or all of their cache coherence protocols in software. One way to exploit the flexibility of this software is to tailor a coherence protocol to match an application's communication patterns and memory semantics. Blizzard presents evidence that this approach can lead to large performance improvements. It shows that application-specific protocols substantially improved the performance of three application programs—appbt, em3d, and barnes—over carefully

tuned transparent shared memory implementations. The speed-ups were obtained on Blizzard, a fine-grained DSM system running on Thinking Machines CM-5.

Alewife[24] is an early prototype of such DSM architectures, that uses a hybrid of software and hardware mechanisms to support coherent shared memory, efficient user-level messaging, fine-grain synchronization, and latency tolerance. Alewife supports up to 512 processing nodes connected over a scalable and cost-effective mesh network at a constant cost per node. Four mechanisms combine to achieve Alewife's goals of scalability and programmability: software-extended coherent shared memory provides a global, linear address space; integrated message passing allows compiler and operating system designers to provide efficient communication and synchronization; support for fine-grain computation allows many processors to cooperate on small problem sizes; and latency-tolerance mechanisms including block multithreading and prefetching mask unavoidable delays due to communication.

By integrating mechanisms from shared memory, message passing and data flow architectures into a hybrid distributed shared memory (DSM) model, the Massachusetts Institute of Technology (MIT) Alewife machine shows that a parallel architecture can yield a rich shared memory programming environment on a scalable hardware base. The hardware, compiler, and operating system combine to achieve the goal of programmability by solving problems that traditionally burden multiprocessor programmers, namely, scheduling computation and moving data between processing elements. Features of this environment include a globally shared address space, a scalable cache coherence mechanism, a compiler that automatically partitions regular programs with loops, a library of efficient synchronization and communication routines, distributed garbage collection, and a parallel debugger.

VM-based shared memory on low-latency, remote-memory-access networks **[15]:** Cashmere and TreadMarks DSMs were implemented on a 32-processor DEC Alpha cluster connected by a Memory Channel network and performance comparison was done. Both Cashmere and TreadMarks use virtual memory to maintain coherence on pages, and both use lazy, multi-writer release consistency.

The systems differ dramatically, however, in the mechanisms used to track sharing information and to collect and merge concurrent updates to a page, with the result that Cashmere communicates much more frequently, and at a much finer grain.

They have shown that low-latency networks make DSM based on fine-grain communication competitive with more coarse-grain approaches, but that further hardware improvements will be needed before such systems can provide consistently superior performance. It is observed that Cashmere scales slightly better than TreadMarks for applications with false sharing. At the same time, it is severely constrained by limitations of the current Memory Channel hardware. In general, performance is better for TreadMarks.

Sirocco- Software fine-grain distributed shared memory (FGDSM) provides a simplified shared-memory programming interface with minimal or no hardware support **[14]**. Originally software FGDSMs targeted uni-processor node parallel machines. Sirocco is implemented on a family of software FGDSMs on a network of low-cost SMPs. Sirocco takes full advantage of SMP nodes by implementing inter-node sharing directly in hardware and overlapping computation with protocol execution. To maintain correct shared-memory semantics, however SMP nodes require mechanisms to guarantee atomic coherence operations. Multiple SMP processors may also result in contention for shared resources and reduce performance. SMP nodes also impact the cost trade-off. While SMPs typically charge higher price-premiums, for a given system size SMP nodes substantially reduce networking hardware requirement as compared to uni-processor nodes.

2.2 Heterogeneous Distributed Shared Memory Systems

The problems of building a distributed shared memory system on a network of heterogeneous machines are discussed in Extending Distributed Shared Memory to Heterogeneous Environments **[43]**. An existing algorithm that implements distributed shared memory is extended to a heterogeneous environment. An implementation that runs on Sun and DEC Firefly multiprocessor workstations connected by Ethernet is described. Related implementation and performance issues are discussed. On the basis of measurements of the applications ported to the system, it is concluded that

heterogeneous distributed shared memory is not only feasible but can also be compared in performance to its homogeneous counterpart.

The Parallel Virtual Machine (PVM), an integrated framework for heterogeneous network computing, lets scientists exploit collections of networked machines when carrying out complex scientific computations **[47].** Under PVM, a user-defined grouping of serial, parallel, and vector computers appears as one large distributed-memory machine. Configuring a personal parallel virtual computer involves simply listing the names of the machines that is read when PVM is started. Applications can be written in Fortran 77 or C and parallelized by use of message-passing constructs common to most distributed memory computers. With the use of messages sent over the network, multiple tasks of an application can cooperate to solve a problem in parallel.

This paper discusses components of PVM, including the programs and library of interface routines. It summarizes the characteristics of appropriate applications and discusses the current status and availability of PVM. In addition, the article introduces a recent extension to PVM known as the Heterogeneous Network Computing Environment (HeNCE).

Adsmith [**50]** is an object-based distributed shared memory. In an object-based DSM, the shared memory consists of many shared objects, through which the shared memory is accessed. Adsmith is built on top of PVM at the library layer using C++. PVM is used as the communication subsystem, because it is a de facto standard and encapsulates many system related details. Several mechanisms are used to improve the performance of Adsmith, such as release memory consistency, load/store like memory accesses, non-blocking accesses, and atomic operations, etc. Performance results show that even though Adsmith is implemented on top of PVM, programs running on Adsmith can achieve a performance on par with SDSM.

A Distributed Shared Memory based on Multiple Java virtual Machines[52] Multijav is a distributed shared memory based on distributed java virtual machines. Since this is implemented using standard java it can be called as HDSM in true sense. It employs changes in JVM implementation and as such does not pose any restriction on applications. This also handles all the issues related to DSM namely consistency

parallel programming, thread migration and other implementation issues in an effective manner.

JESSICA stands for Java-enabled single-system-image computing architecture [37], a middleware that runs on top of the standard UNIX operating system to support parallel execution of multithreaded Java applications in a cluster of computers. JESSICA hides the physical boundaries between machines and makes the cluster appear as a single computer to applications—a *single system image*. JESSICA supports preemptive thread migration, which allows a thread to freely move between machines during its execution, and global object sharing through the help of a distributed shared memory subsystem. JESSICA implements location-transparency through a message-redirection mechanism. The result is a parallel execution environment where threads are automatically redistributed across the cluster for achieving the maximal possible parallelism. A JESSICA prototype that runs on a Linux cluster has been implemented and considerable speedups have been obtained for all the experimental applications tested.

PM2 [48] is a high performance communication middle layer, for heterogeneous network environments. PM2 currently supports Myrinet, Ethernet, and SMP. Binary code written in PM2 or written in a communication library, such as MPICH-SCore on top of PM2, may run on any combination of those networks without re-compilation. According to authors, a set of NAS parallel benchmark results, MPICH-S Core performance is better than dedicated communication libraries.

JESSICA2 [56] a new DJVM running in JIT compilation mode, executes multi-threaded Java applications transparently on clusters. JESSICA2 provides a single system image (SSI) illusion to Java applications via an embedded global object space (GOS) layer. It implements a cluster-aware Java execution engine that supports transparent Java thread migration for achieving dynamic load balancing. Issues supporting transparent Java thread migration in a JIT compilation environment were discussed and several lightweight solutions were proposed. An adaptive migrating-home protocol used in the implementation of the GOS is introduced. The system has been implemented on x86-based Linux clusters and significant performance improvements over the previous JESSICA system have been observed.

JavaSplit [55] is a portable runtime for distributed execution of multithreaded Java programs which was presented in this work. Java-Split transparently distributes threads and objects of an application among the participating nodes. Thus, it gains augmented computational power and increased memory capacity without modifying the Java multithreaded programming conventions. Java-Split works by rewriting the byte codes of a given parallel application, transforming it into a distributed application that incorporates all the runtime logic. Each runtime node carries out its part of the resulting distributed computation using nothing but its local standard (unmodified) Java virtual machine (JVM).

This is unlike previous Java-based distributed runtime systems, which use a specialized JVM or utilize unconventional programming constructs. Since Java-Split is orthogonal to the implementation of a local JVM, it achieves portability across any existing platform and allows each node to locally optimize the performance of its JVM, e.g., via a just-in-time compiler (JIT).

A distributed Java Virtual Machine (DJVM) spanning multiple cluster nodes can provide a true parallel execution environment for multi-threaded Java applications. Most existing DJVMs suffer from the slow Java execution in interpretive mode and thus may not be efficient enough for solving computation-intensive problems.

Heterogeneous computing combines general purpose CPUs with accelerators to efficiently execute both sequential control-intensive and data parallel phases of applications. Existing programming models for heterogeneous computing rely on programmers to explicitly manage data transfers between the CPU system memory and accelerator memory **[34].**

A new programming model for heterogeneous computing, called Asymmetric Distributed Shared Memory (ADSM) [34] maintains a shared logical memory space for CPUs to access objects in the accelerator physical memory but not vice versa. The asymmetry allows light-weight implementations that avoid common pitfalls of symmetrical distributed shared memory systems. ADSM allows programmers to assign data objects to performance. Four fundamental functions an ADSM system must implement are shared-data allocation, shared-data release, method invocation, and return synchronization.

ADSM reduces programming efforts for heterogeneous computing systems and enhances application portability. A software implementation of ADSM, called GMAC, on top of CUDA in a GNU/Linux environment was realized. The applications written in ADSM and running on top of GMAC achieve performance comparable to their counterparts using programmer managed data transfers. Additional architectural support that allows GMAC to achieve higher application performance than the current CUDA model is also realized.

JAC [61] is an extended Java that introduces a higher level of concurrency, hiding threads and separating thread synchronization from application logic in a declarative fashion, java.util.concurrent package provides some excellent and powerful concurrent components. This paper introduces how to reconstruct the JAC precompiler to improve the JAC using concurrent package. The new precompiler can create Java code based on the concurrent package

2.3 Process Migration/Thread Migration

Process migration is the act of transferring a process between two machines. It enables dynamic load distribution, fault resilience, eased system administration, and data access locality. Despite these goals and ongoing research efforts, migration has not achieved widespread use. With the increasing deployment of distributed systems in general, and distributed operating systems in particular, process migration is again receiving more attention in both research and product development.

Migration may be used for dynamic load balancing purposes with the aim of gaining increased performance from a group of processors than may be gained by schemes simply allocating processes to processors at run time. Schemes providing object migration also offer object persistence, improved fault tolerance and potentially more efficient remote object invocation (RPC).

The survey **[38]** covers systems providing process migration over both modified and unmodified UNIX and various experimental operating systems. Task migration over two modern microkernel-based operating systems is followed by a description on a number of object migration facilities with objects of varying granularity.

Clusters are attractive for executing sequential and parallel applications. However, there is a need to design a cluster distributed operating system to provide a Single System Image. A cluster operating system providing both a DSM system and load balancing is attractive for efficiently executing a workload of sequential applications and shared memory parallel applications.

Survey presented in [41] reviews the field of process migration by summarizing the key concepts and giving an overview of the most important implementations. Design and implementation issues of process migration are analyzed in general, and then revisited for each of the case studies described: MOSIX, Sprite, Mach, and Load Sharing Facility.

The benefits and drawbacks of process migration depend on the details of implementation and, therefore, this paper focuses on practical matters. Survey helps in understanding the potentials of process migration and why it has not caught on.

Gobelins [39] is a distributed operating system dedicated to clusters that provides both a DSM system and a process migration mechanism to support load balancing. The implementation of Gobelins process migration mechanism which exploits Gobelins kernel level DSM system is carried out. Gobelins DSM allows implementing simply an efficient migration mechanism that can be used to move processes or threads among cluster nodes.

2.4 Checkpointing

Current approaches for checkpointing assume system homogeneity, where checkpointing and recovery are both performed on the same processor architecture and operating system configuration. Sometimes it is desirable or necessary to recover a failed computation on different processor architecture. For such situations, checkpointing and recovery must be portable. In these works [40], it was stated that source-to-source compilation is an appropriate concept for this purpose. The c2ftc compiler enables machine-independent checkpoints by automatic generation of checkpointing and recovery code. Sequential C programs are compiled into fault tolerant C programs, whose checkpoints can be migrated across heterogeneous networks, and restarted on binary incompatible architectures. Experimental results on

several systems provide evidence that the performance penalty of portable checkpointing is negligible for realistic checkpointing frequencies.

Checkpointing in a homogeneous environment, where both checkpointing and recovery are performed on the same type of machine and operating system, has been studied extensively in [45]. As heterogeneous distributed systems become pervasive, it is desirable to extend the capability of checkpointing to non-homogeneous environments. A prototype, PREACHES, that achieves portable checkpointing of single process applications in heterogeneous systems using checkpoint propagation is presented. The checkpoint propagation technique generates machine-dependent checkpoints for each architecture in the heterogeneous environment. When failure occurs, the failed process can be restarted on a specified machine with the checkpoint that is appropriate for the architecture. An implementation of PREACHES on a heterogeneous network of workstations has been successfully developed based on TCP/IP communication. PREACHES also provides automatic and fast recovery for single process programs.

Trends in high-performance computing are making it necessary for long-running applications to tolerate hardware faults. The most commonly used approach is checkpoint and restart (CPR) - the state of the computation is saved periodically on disk, and when a failure occurs, the computation is restarted from the last saved state. At present, it is the responsibility of the programmer to instrument applications for CPR. Application-level checkpointing for shared memory programs are discussed in [46].

The use of compiler technology to instrument codes to make them self-checkpointing and self-restarting, thereby providing an automatic solution to the problem of making long-running scientific applications resilient to hardware faults was derived in this study. In this, a system for shared-memory programs running on symmetric multiprocessors was discussed.

This system has two components: (i) a pre-compiler for source-to-source modification of applications, and (ii) a runtime system that implements a protocol for coordinating CPR among the threads of the parallel application. For the sake of concreteness, a non-trivial subset of OpenMP that includes barriers and locks was focused. One of the

advantages of this approach is that the ability to tolerate faults becomes embedded within the application itself, so applications become selfcheckpointing and selfrestarting on any platform.

Experiments show that the overhead introduced by this approach is usually quite small; they also suggest ways in which the current implementation can be tuned to reduced overheads further.

One potentially attractive way to build large scale shared memory machines is to use small-scale to medium-scale shared-memory machines as clusters that are interconnected with an off-the shelf network. To create a shared-memory programming environment across the clusters, it is possible to use a virtual shared memory software layer **SoftFLASH** [57]. Because of the low latency and high bandwidth of the interconnect available within each cluster, there are clear advantages in making the clusters as large as possible. The critical question then becomes whether the latency and bandwidth of the top-level network and the software system are sufficient to support the communication demands generated by the clusters. An aggressive kernel implementation of a virtual shared-memory system using SGI multiprocessors and 100Mbyte/sec HIPPInterconnects (High Performance Parallel Interface) was implemented.

2.5 Thread Pool Executor Related

The concurrency utilities in the Java library supply a feature called the executor framework that initiates and controls the execution of threads. It provides many adjustable parameters and extensibility hooks to be useful across a wide range of contexts for specifically configuring a pool of threads to execute submitted runnables depending on the application's specific needs.

The Real-Time Specification for Java (RTSJ)[60] supports asynchronous event handling (AEH), where the mapping of handlers to server threads is performed by the real-time JVM. Although the executor framework is not RTSJ-aware, it is very flexible.

This work using the executor framework to implement asynchronous event handling in the RTSJ presents the feasibility of using the framework to implement various AEH mapping models. First step is to identify the issues associated with adapting the executor framework to AEH in the RTSJ.

2.6 Prefetching Related

Stride prefetching aims at detecting sequences of data accesses whose addresses are equidistant with a certain stride. John W. C. Fu Janak H. Pate1 and Bob L. Janssens [87] have proposed a stride prefetching for scalar processors which envisages a hardware based Stride prediction table (SPT) whose configuration is given in figure 2.1.

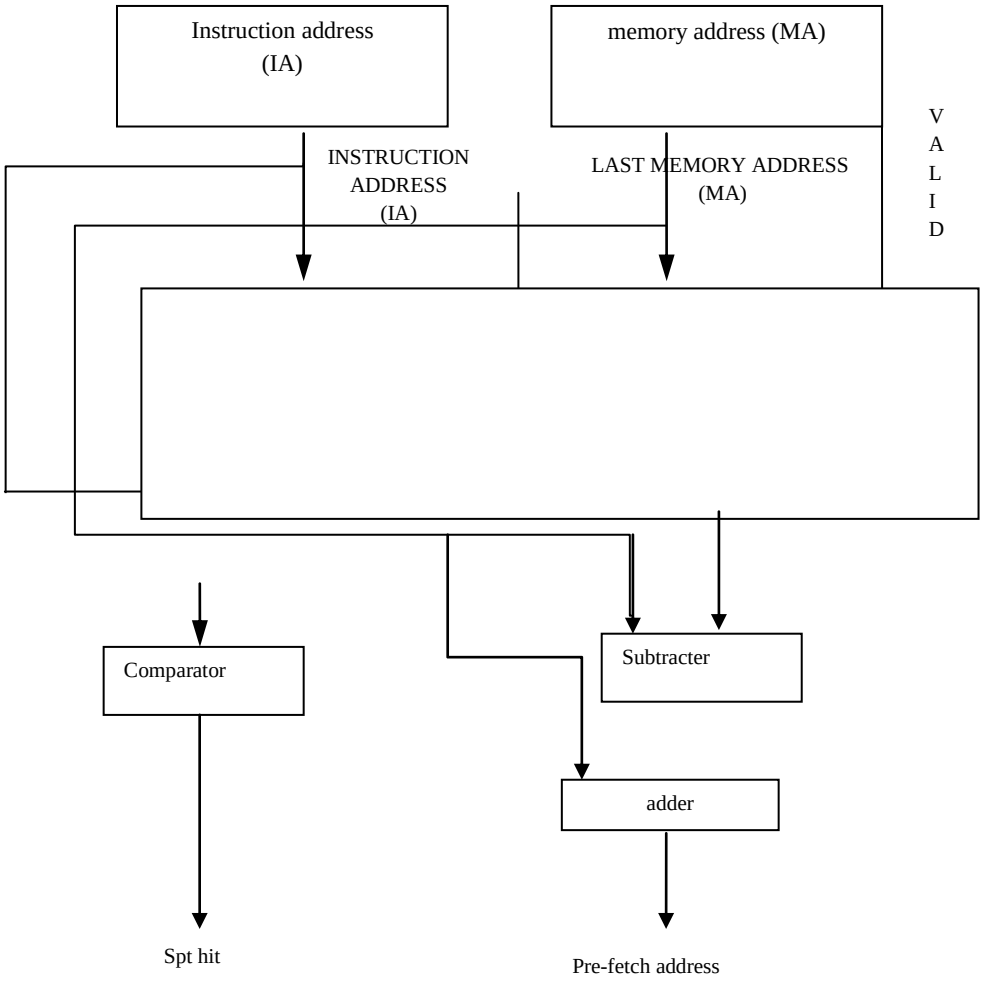


Figure 2.1 Stride Prediction Table

Figure 2.1 shows the basic SPT organization. Let *IAcur* and *MAcur* be the current instruction and data memory address pair being processed and let *IAspt* and *MAspt be* some memory address pair held in the SPT. Each memory access made by the processor presents a *IAcur* and *MAcur* address pair to the SPT. The address *IAcur is used to* access the SPT and if the SPT entry is valid, the corresponding *IAspt* and MAspt are made available. The stride is calculated as *MAcur – MAspt* and the prefetch address is *MAcur +stride.* This is a spt hit and a prefetch attempt can be made if the stride is not zero. The *MA* field of the SPT entry causing the spt hit is updated with *MAcur.* If the SPT entry is not valid, the *IAcur* and *MAcur* address pair is written into the SPT and the valid bit set. This is a spt miss.

The output of SPT is prefetch address which is prefetched by the processor; this will happen if there is a miss from the table. The number and size of the SPT will be design parameter which should be chosen for a set of applications.

Using these strides to direct prefetching into the cache, the results show that for programs that can be highly vectorized, this method can significantly reduce the number of cache misses with low overhead.

The efficiency comparison of stride prefetching and sequential prefetching, two promising hardware-based prefetching schemes to reduce read-miss penalties in shared-memory multiprocessors, was done in Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors **[59]**. Although stride accesses dominate in four out of six of the applications, it was found that sequential prefetching does as well as and in some cases even better than stride prefetching for five applications. This was because 1) most strides are shorter than the block size (we assume 32 byte blocks), which means that sequential prefetching is as effective for these stride accesses, and 2) sequential prefetching also exploits the locality of read misses with nonstride accesses. However, since stride prefetching in general results in fewer useless prefetches, it offers the extra advantage of consuming less memory-system bandwidth.

Alexander C. Klaiber, Henry M. Levy **[83]** proposed a prewrite back technique that can reduce the impact of stalls due to replacement of writeback in cache. The compiler adds the FETCH instruction to aid prefetch by the hardware. This is decoded at the decode stage and the address is checked with the contents of associative

memory for it was already available in prefetch buffer. If it was not found, then this datum is prefetched and placed in prefetch buffer which is also cache memory similar to on cache memory of CPU.

David Callahan,Ken Kennedy,Allan Porterfield [73] have presented an alternative approach called **software prefetching**, to reduce cache miss latencies. The idea was to provide a non blocking prefech instruction that causes data at a specified memory address to be brought into cache. When generated by a very simple compiler algorithm, prefetch instructions can eliminate nearly all cache misses, while causing only modest increases in data traffic between memory and cache.

At compile-time, FETCH instructions are inserted into the instruction-stream by the compiler, based on anticipated data references and detailed information about the memory system. At run time, a separate functional unit in the CPU, the fetch unit, interprets these instructions and initiates appropriate memory reads, Prefetched data is kept in a small, fully associative cache, called the fetchbuffer, to reduce contention with the conventional direct-mapped cache.

Todd C.Mowry, Monica S.Lam, Anoop Gupta [76] proposed a compiler algorithm to insert prefetch instructions into code that operates on dense matrices. Their algorithm identifies those references that are likely to be cache misses, and issues prefetches only for them. This was implemented in the SUIF (Stanford University Intermediate Form) optimizing compiler. The SUIF compiler includes many of the standard optimization and generates code competitive with the MIPS compiler. To generate fully functional and optimized code with prefetching SUIF compiler is used. By generating fully functional code, the improvements in cache miss rates, the overall performance of a simulated system was measured .This algorithm inserts prefetchs at the elements of the matrix and optimizes by employing loop unrolling as well. The algorithm significantly improves the execution speed of standard benchmark programs-some prefetches for array accesses, also eliminate many of the unnecessary prefetches without any significant decrease in the coverage of the cache misses.

Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors is studied by Fredrik Dahlgren and Per Stenstrom **[88]**

where in the stride prefetching is carried with the help of Reference Prediction Table(RPT) whose structure is given figure 2.2 , below.

Reference Prediction Table, RPT

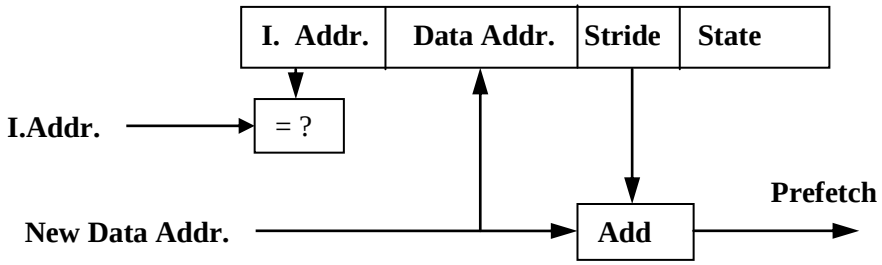


Figure 2.2 I detection scheme(RPT).

Two issues must be addressed in order for a stride prefetching scheme to be effective. First, the stride in a stride sequence must be dynamically identified which is done in the *detection phase.* Second, when a stride sequence is detected, prefetch requests must be issued early enough so that the block will be available in the cache when the processor eventually accesses it. This is done in the *preferching phase.* Note that the second issue is applicable also to other hardware-based prefetching schemes and to other access patterns than stride sequences. It is assumed that the instruction address of the load instruction that misses in the Second Level Cache (*SLC)* is matched against the entries in a *Reference Prediction Table* (RPT) which is organized as a cache.

Haiming Liu and Weiwu Hu [84] have implemented two kinds of prefetching on the home-based software DSM system JIAJIA. The first one, history prefetching (utilizes the temporal locality of the programs) predicts which pages will be accessed by scanning the access history of the pages to find if there is any periodicity in the sequence of access and invalidation. If it deduces that the next operation for a page is an access, then the page is prefetched. Multiple pages can be prefetched with one message. The second one, aggregate prefetching (utilizes the spatial locality of the programs), aggregates the fault pages between two barriers into one or more page-sets in the order they are accessed. When a page fault occurs, the process fetches both the faulty pages and the pages following the fault page in the same page set. The reason for this is that for applications with regular memory access pattern, pages accessed in sequence are probably to be accessed in that order again.

Ricardo Bianchini, Raquel Pinto, and Claudio L. Amorim [85] used Adaptive++ technique, a novel runtime-only data prefetching strategy for software-based distributed shared-memory systems (software DSMs). Adaptive++ improves the performance of regular parallel applications running on software DSMs by using the past history of memory access faults to adapt between repeated-phase and repeated-stride prefetching modes. Through detailed execution-driven simulations of several applications, the prefetching technique was very successful at reducing the data access overheads of regular applications running on the TreadMarks software DSM. Adaptive++ also reduces the overhead of applications that are not strictly regular but that exhibit periods of regularity. In terms of overall performance, Adaptive++ provided speedup improvements as significant as 34% on 16 processors.

Dahlgren,Dubios M and Strenstrom P[69] proposed a sequential hardware pre-fetching for shared multiprocessors. Sequential prefetching is a simple hardware-controlled prefetching technique which relies on the automatic prefetch of consecutive blocks following the block that misses in the cache, thus exploiting spatial locality. In its simplest form, the number of prefetched blocks on each miss is fixed throughout the execution. However, since the prefetching efficiency varies during the execution of a program, the number of prefetched blocks according to a dynamic measure of prefetching effectiveness is adapted. The advantage of such prefetcher is it reduces read misses, read penalty and execution time is also reduced to an extent. However, prefetcher needs to be incorporated at the processor level in hardware.

Tse J and Smith A.Je [80] have done extensive analysis in uni-processor environment utilizing pre-fetches in to the caches. They concluded that because of the architectural limitations such as common bus and single port to access memory the prefetch at times is degrading the performance than improving. Making memory dual ported and increasing the bus bandwidth and interleaved access incorporation for memory accessing can lead to improved performance.

Prefetching was also studied by Srinath S, Mutlu O, Hyesoon Kim and Patt Y. N. [82]. They have observed that Prefetching has the potential to improve performance, if the memory access pattern is correctly predicted and the prefetch requests are initiated early enough, before the program accesses the predicted memory addresses. Since the memory latencies faced by today's processors are on the order of hundreds

of processor clock cycles, accurate and timely prefetching of data from main memory to the processor caches can lead to significant performance gains by hiding the latency of memory accesses. On the other hand, prefetching can negatively impact the performance and energy consumption of a processor due to two major reasons, especially if the predicted memory addresses are not accurate:

• First, prefetching can increase the contention for the available memory bandwidth.

• Second, prefetching can cause cache pollution if the prefetched data displaces cache blocks that will later be needed by load/store instructions in the program.

Based on these measured parameters, if the values are acceptable the prefetch can be enabled otherwise it can be delayed.

Speculative execution via address prediction and data prefetching was proposed by Jose' Gonzalez, Antonio Gonzalez [79], a technique based on this load/store feature of the instruction, memory instructions that are highly predictable are dynamically identified and issued speculatively. Furthermore, in the case of load instructions, the address of the next execution is predicted and the value is brought from memory into a structure Memory Prefetching Table (MPT) in order to be available for next execution of the same the same load instruction. When a load arrives at the decode stage, its effective address is predicted and the entry obtain from the MPT is checked in order to know if the value has been prefetched. If the value is available, the load destination register is updated and all the subsequent dependent instruction will be allowed to be executed speculatively. Otherwise, the predicted address will be used to speculatively issue this load and the instructions dependent on it. Several cycles later, memory instructions will verify their prediction. If the prediction is correct and the load obtained the value at decode stage from the MPT, it will not require any further memory access. In the case of a misprediction, all the misspeculated instructions will be re-executed.

The above technique is effective because for the instruction, the memory access for the operand fetch as the data required is available; it can be avoided especially for loads and stores. Usually it is observed that about more than 25% of the generated

instructions will be load and stores for any typical application. Hence any saving in terms of execution time is remarkable.

2.7 N-Gram based Language Models

Predicting a word from previous n words in a sample of text, using n-gram based model on classes of words was done by Peter E.Brown, Peter V.deSouza *et.al* **[62].** The vocabulary V words into C classes using a mathematical function which maps w_i with a class C_i. These classes can be merged by means of an algorithm to reduce the number of classes.

The application is retrieval of a particular word from several words of the document. This is word retrieval from the list of words and also the word sequence as it has been found in the sentences. This application is based on n-gram model for identifying the words in question and how frequently the word was found/used in the document under reference.

But later it degrades its performance. It was also observed that the n in ngram model worked better for tri-gram model. Later, as n increases the performance improvement was negligible. The n-gram model was applied for speech recognition applications.

Steffen Bickel, Peter Haider, and Tobias Scheffer **[68]** cover the benefit that users in several application areas can experience from a "tab-complete" editing assistance function. They proposed and developed an evaluation metric and adapt *N*-gram language models to the problem of predicting the subsequent words, given an initial text fragment, using an instance-based method as baseline.

This technique was applied to study the predictability of call-center emails, personal emails, weather reports, and cooking recipes. The derived method has a *k* best Viterbi beam search decoder. The experimental results have shown that the *N*-gram based completion method has a better precision recall profile than index-based retrieval of the most similar sentence.

TaroWatanabe Hajime Tsukada Hideki Isozaki **[66]** proposed *lossless* compression of Ngram language models based on LOUDS, a succinct data structure. LOUDS

succinctly represents a trie with M nodes as a 2M + 1 bit string. The space is further reduced by considering the N-gram structure.

A compression technique is employed to compress it further for the N-gram language model structure. It also uses '*variable length coding*' and '*block-wise compression*' to compress *values* associated with nodes. When experimented with English Web 1T, a 5-gram from LDC consisting of 25 GB of zipped raw text N-gram counts. By using 8-bit floating point quantization 1, N-gram language models are compressed into 10 GB, which is comparable to a *lossy* representation.

Experimental results showed that succinct representation drastically reduces the space for the pointers compared to the sorted integer compression approach. Furthermore, This fastest implementation is as fast as the widely used SRILM while requiring only 25% of the storage. Employs most compact representation can store all 4 billion n-grams and associated counts for the Google n-gram corpus in 23 bits per n-gram, the most compact lossless representation to date, and even more compact than recent lossy compression techniques. It utilizes tries data structures for efficient representation. Tries represent collections of n-grams using a tree. Each node in the tree encodes a word, and paths in the tree correspond to n-grams in the collection. Tries ensure that each n-gram prefix is represented only once, and are very efficient when n-grams share common prefixes. Values can also be stored in a trie by placing them in the appropriate nodes. Conceptually, trie nodes can be implemented as records that contain two entries: one for the word in the node, and one for either a pointer to the parent of the node or a list of pointers to children. At a low level, however, naive implementations of tries can waste significant amounts of space. It also employs compression and hashing techniques to arrive at the required item search. This also makes use of sorted lists to have efficient search mechanism and in turn to arrive at required item. It uses n sorted arrays, one for each n-gram order. Finally to conclude, it employs novel techniques of sorted arrays, hash tables and compression to have efficient storage of data structures and search mechanism. The space of N-grams was significantly reduced by *variable length coding* and *block compression.*

Dou Shen, Jian-Tao Sun, Qiang Yang and Zheng Chen **[63]** have proposed a text classification system based on multi-gram model. Document representation refers to

the selection of appropriate features to represent documents. A bag-of-word representation scheme is widely used in text classification due to its simplicity and efficiency.

One way to address the above problem is to use features with coarser granularity, such as phrases to replace or augment single words. Lewis used syntactic parsing to create indexing phrases. These phrases correspond to pairs of words in one of several specified syntactic relationships in the original document (e.g. verb and head noun of subject, noun and modifying adjective, etc.).

Adam Pauls Dan Klein **[65]** proposed Faster and smaller *N*-gram language models.It also covers N-gram language models which are a major resource bottleneck in machine translation. The largest language models (LMs) can contain as many as several hundred billion n-grams, so storage is a challenge. At the same time, decoding a single sentence can trigger hundreds of thousands of queries to the language model, so speed is also critical.

Microsoft's Web n-gram Corpus, Kuansan Wang Christopher Thrasher, *et.al*[64], is also based on n-gram model designed, and is made publicly available as an XML Web Service. A noticeable feature is that it can be updated as deemed necessary by the user community to include new words and phrases constantly being added to the Web. The corpus makes available various sections of a Web document, specifically, the body, title, and anchor text, as separate models as text contents in these sections are found to possess significantly different statistical properties and therefore are treated as distinct languages from the language modeling point of view.

Microsoft Web n-gram provides open-vocabulary, smoothed back-off n-gram models for the three text streams using the CALM (Constantly Adaptive Language Modeling technique) algorithm [93] that dynamically adapts the n-gram models as web documents are crawled. The design of CALM ensures that new n-grams are incorporated into the models as soon as they are encountered in the crawling and become statistically significant. The models are therefore kept up-to-date with the web contents. CALM is also designed to make sure that duplicated contents will not have outsized impacts in biasing the n-gram statistics. Currently, the maximum order of the n-gram available is 5.

Whether sentence completion is helpful strongly depends on the diversity of the document collection as, for instance, measured by the entropy. A modest but significant benefit can be observed for thematically related documents: weather reports and cooking recipes.

Hatice Ulku Osmanbeyoglu, Madhavi K Ganapathiraju **[67**] have applied N-gram based methods to biological domain. Karlin et al **[94]** introduced a "genomic signature" based on dinucletiode odds ratio (relative abundance) values which appeared to reflect the species-specific properties of DNA modification, replication and repair mechanism. Campbell et al **[95]** compared dinucleotide frequencies (genomic signatures) of prokaryote, plasmid, and mitochondrial. They showed that plasmids and their hosts have substantially compatible nucleotide signatures.

It has been suggested previously that genome and proteome sequences show characteristics typical of natural-language texts such as "signature-style" word usage indicative of authors or topics, and that the algorithms originally developed for natural language processing may therefore be applied to genome sequences to draw biologically relevant conclusions.

Following this approach of 'biological language modeling', statistical n-gram analysis has been applied for comparative analysis of whole proteome sequences of 44 organisms. It has been shown by them that a few particular amino acid n-grams are found in abundance in one organism but occurring very rarely in other organisms, thereby serving as genome signatures. At that time proteomes of only 44 organisms were available, thereby limiting the generalization of this hypothesis. Today nearly 1,000 genome sequences and corresponding translated sequences are available, making it feasible to test the existence of biological language models over the evolutionary tree.

Whole proteome sequences of microbial organisms have been shown to contain particular n-gram sequences in abundance in one organism but occurring very rarely in other organisms, thereby serving as proteome signatures. Further it has also been shown that perplexity, a statistical measure of similarity of n-gram composition, can be used to predict evolutionary distance within a genus in the phylogenetic tree.

Chapter 3. Hierarchical Thread Pool Executor

3.1 Introduction

Thread Pool Executor (TPE) is a software component which facilitates the assignment of incoming requests in the form of tasks to the worker threads. There are three important parameters for TPE operations: *Core pool size* is a configurable parameter, which by default is set to number of cores available in a given system. All the tasks for execution are placed in blocking queue. The *Max pool size* defines upper limit for number of worker threads and usually is more than number of cores. It is worthwhile to mention that for every multi-threaded process a separate TPE is created.

Whenever a new task arrives for execution, and the worker threads are available, a thread is assigned to it. In case there is no worker thread is available, its request is stored in to the blocking queue. Whenever the worker thread is released, the task will be considered for execution and hence its entry is removed from the queue. The number of worker threads (*Core pool size)* are persistent and during the process execution.

In case when the Blocking queue is full and still more tasks are arrived, TPE creates new transient threads (maximum of (*Max Pool size - Core pool size)*). Instead of assigning tasks from the queue, the threads are assigned to the new tasks. General philosophy used in TPE is to discourage more tasks than core pool size to achieve parallelism. At worst it achieves concurrency if it reaches to max pool size. **If still more tasks are arrived TPE throws exception and tasks are rejected.**

Thread pool executor is an essential component of every DSM, where we have more than one system with many cores on each node and the above mentioned philosophy results in under utilization of resources.

If we would like to overcome these limitations, we need to have a TPE which is scalable, efficient, and instead of raising exceptions should be able to accept request and migrate it other systems (nodes).

In this chapter we are proposing a theoretical design and implementation of a Hierarchical Thread Pool Executor (HTPE) which achieves above mentioned goals. The proposed HTPE works in the following way.

1. We maintain a single HTPE per system.
2. Whenever a new task arrives and if the worker thread of the system it is initiated is free, a thread is assigned to it.
3. In case no worker thread is available, instead of storing them into a blocking queue, we store them into non-blocking queue, benefits of using the later data structure is explained more in detail in section 3.4.
4. The tasks are sorted on their priority of execution in non-blocking queue.
5. It checks for availability of worker threads into other systems HTPE daemon, this is accomplished by maintaining a common min-heap structure that stores the availability of worker threads.
6. Now thread migration takes place.

Java 1.5 provides thread pool executor class as part of concurrent utilities package. Applications which are primarily meant for single processor or multi-core processors seldom make use of the methods provided along with the thread pool executor.

A thread pool offers a solution to both the problem of thread life-cycle overhead and the problem of resource thrashing. By reusing threads for multiple tasks, the thread-creation overhead is spread over many tasks. As a bonus, because the thread already exists when a request arrives, the delay introduced by thread creation is eliminated.

The proposed Hierarchical Thread pool Executor (HTPE) has several features which will take into account the parameters of the underlying operating system, hardware and dynamic load of nodes to arrive at a mechanism to exploit and use resources optimally under the frame work of HDSM.

3.2 Motivation for Hierarchical Thread Pool Executor (HTPE)

In a typical HDSM, the application program is subdivided into a Process or a Thread when it comes for execution in the Run-Time System layer (RTS). If the HDSM is process based, then the operating system and RTS will provide suitable environment for smooth execution of the application.

Assuming that the HDSM is a thread-based implementation, the possible choices are based on either *Pthreads* or *Java threads*. In case of Pthreads based RTS, a library function related to thread creation/deletion is invoked, eventually thread will be created and destroyed. In case of process (MPI module), processes required for this module are created and managed by the RTS of MPI system.

Majority of the JAVA based HDSMs may be thread based or pure object based. If the HDSM is thread based, the thread can be created in one of the following ways.

Provide a `Runnable` *object*- The `Runnable` interface defines a single method, `run`, meant to contain the code executed in the thread.

The first idiom, which employs a `Runnable` object, is more general, because the `Runnable` object can be subclass or a class other than `Thread`.

The second idiom is easier to use in simple applications but is limited by the fact that our task class must be a descendant of `Thread`. Creating and deleting threads every time involves a large overhead for creating an object or interface will not be efficient. Hence if the language itself provides a facility then the RTS design will be more-efficient and flexible.

JAVA 5.0 onwards this is provided by the Java.concurrent.utilities package which is called as Thread Pool Executor service. This is proposed to be realized and updated as part of the Run time system (RTS).

Applications built with thread pools are subject to all the same concurrency risks as any other multithreaded application, such as synchronization errors and deadlock, and

a few other risks specific to thread pools as well, such as pool-related deadlock, resource thrashing, and thread leakage. These are detailed in section 3.4.3. The disadvantages of Unbounded Thread Creation are thread lifecycle overhead, resource consumption and stability are also detailed in section 3.4.3.

Apart from these issues, the TPE is designed to be facility for multithreaded application process. In case of HDSM, as part of optimization, we would like to balance the load on all the systems (nodes), such that we can arrive at increased throughput. This has prompted us to design Hierarchical thread pool executor which will enable to place the thread on the system with minimal load.

As part of thread migration mechanism of the RTS, we place all incoming threads of the application into a non-blocking queue and these will be submitted to TPE/transferred to the appropriate system's TPE with minimal load.

3.3 Thread Pool Executor

Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This type of thread exists separately from the Runnable and Callable tasks, and it executes and is often used to execute multiple tasks. Using worker threads minimizes the overhead due to thread creation. A user's request submitted to TPE for execution is termed as task. A task is assigned to a worker thread, and worker threads are executed on behalf of tasks in the TPE. Thread objects use a significant amount of memory, and in a large-scale application, allocating and de-allocating many thread objects creates a significant memory management overhead. One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue (blocking queue), which holds extra tasks whenever there are more active tasks than threads.

An important advantage of fixed thread pool is that applications using it degrade gracefully. If the fixed thread pool based TPE is used to service a typical Web Server application, where in a server thread is invoked for each HTTP request and is handled

by a separate thread. If the application simply creates a new thread for every new HTTP request(without using TPE), and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceeds the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain (performance degradation when web server need to service more clients). If the reuests are submitted to TPE, it will be able to store these requests in blocking queue, hence will be able accommodate additional requests and allow application performance degrade gracefully).This will allow the HTTP server to consider the requests and give a delayed replies than rejecting the requests(in case TPE is not used).

A simple mechanism to create an executor that uses a fixed thread pool is to invoke the new FixedThreadPool factory method from java.util.concurrent.Executors. This class also provides the following factory methods:

- The newCachedThreadPool method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.
- The newSingleThreadExecutor method creates an executor that executes a single task at a time.
- Several factory methods are ScheduledExecutorService versions of the above executors.

3.3.1 Core and maximum pool sizes

A ThreadPoolExecutor automatically adjusts the pool size according to the bounds set by corePoolSize and maximumPoolSize. When a new task is submitted in method `execute(java.lang.Runnable)`, and less than corePoolSize, while threads are running, a new thread is created to handle the request, even if other worker threads are idle.

If the number of threads running is more than corePoolSize but less than maximumPoolSize, a new thread will be created only if the queue is full.

By setting corePoolSize and maximumPoolSize the same, a fixed-size thread pool is created. Setting maximumPoolSize to an essentially unbounded value such as Integer. By setting MAX_VALUE, the pool allows an arbitrary number of concurrent tasks. Most typically, core and maximum pool sizes are set only upon construction, but they may also be changed dynamically using setCorePoolSize(int) and setMaximumPoolSize(int). Figure 3.1 gives a Thread pool Executor diagram indicating corepoolsize and maxpoolsize

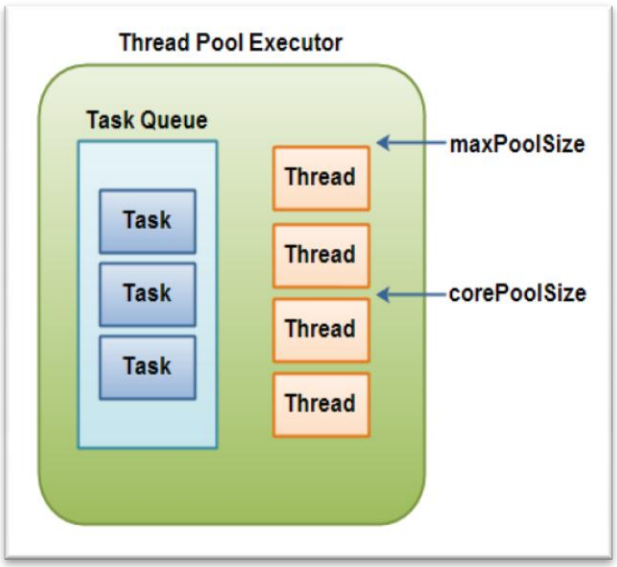


Figure 3.1: Thread pool Executor diagram indicating corepoolsize and maxpoolsize

3.3.2 Benefits of Thread Pooling

Thread pooling saves the virtual machine from the work of creating brand new threads for every short-lived task. In addition, it minimizes overhead associated with getting a thread started and cleaning it up after it dies. By creating a pool of threads, a single thread from the pool can be recycled over and over for different tasks. The thread pooling technique reduces response time because a thread is already constructed and started and is simply waiting for its next task. In the case of an HTTP server an available thread in the pool can deliver file each time that is requested. Without pooling, a brand new thread would have to be constructed and started before the request could be serviced.

All the threads are started, and then each goes into a wait state (which uses very few processor resources) until a task is assigned to it. This fixed size characteristic holds the number of assigned tasks to an upper limit. If all the threads are currently assigned a task, the pool is empty. New service requests can simply be rejected or can be put into a wait state until one of the threads finishes its task and returns itself to the pool. In the case of an HTTP server, this limit prevents a flood of requests from overwhelming the server to the point of servicing everyone very slowly or even crashing.

3.3.3 Problems of Thread Pooling

Thread pools address following problems:

Deadlock: With any multithreaded application, there is a risk of deadlock. The simplest case of deadlock is where thread A holds an exclusive lock on object X and is waiting for a lock on object Y, while thread B holds an exclusive lock on object Y and is waiting for the lock on object X. Unless there is some way to break out of waiting for the lock (which Java locking doesn't support), the deadlocked threads will wait forever. This can happen when thread pools are used to implement simulations involving many interacting objects, and the simulated objects can send queries to one another that then execute as queued tasks, and the querying object waits synchronously for the response.

Resource thrashing: Threads consume numerous resources, including memory and other system resources. Besides the memory required for the `Thread` object, each thread requires two execution call stacks, which can be large. If a thread pool is too large, the resources consumed by those threads could have a significant impact on system performance. Time will be wasted switching between threads, and having more threads than we need may cause resource starvation problems, because the pool threads are consuming resources that could be more effectively used by other tasks.

Thread leakage: A significant risk in all kinds of thread pools is thread leakage, which occurs when a thread is removed from the pool to perform a task, but is not returned to the pool when the task completes. One way this happens is when the task

throws a `RuntimeException` or an `Error`. If the pool class does not catch these, then the thread will simply exit and the size of the thread pool will be permanently reduced by one. When this happens enough times, the thread pool will eventually be empty, and the system will stall because no threads are available to process tasks.

Request overload: It is possible for a server to simply be overwhelmed with requests. In this case, we may not want to queue every incoming request to our work queue, because the tasks queued for execution may consume too many system resources and cause resource starvation.

Each Thread Pool Executor also maintains some basic statistics, such as the number of completed tasks.

Unbounded thread creation Disadvantages:

On the other hand, if we opt for unbounded thread creation: the disadvantages are

1. **Thread life cycle overhead:** Creating thread every time and destroying it and cleanup after completion is overhead every time thread is created.
2. **Resource consumption:** All the resources associated with thread are blocked exclusively for this thread.
3. Stability. We need to write handlers for every conceivable error, Implementor need to conceive every possible error which is difficult.

3.4 Design

The hierarchical thread pool executor is designed to actually increase the efficiency when it comes to the execution of threads in a multi core environment. The design of SDSM assumes that each DSM node is a uniprocessor system. The data structures used are so designed to cater to the needs of uniprocessor systems.

In the proposed HTPE framework, a new set of data structures are proposed (while maintaining compatibility with the previous design) by extending the principal data structures related to system under consideration and the multi-core architecture. The proposed data structure is able to take care of the configuration of the existing system node.

The design is such that, instead of using static data structures which may be initialized at system startup and updated whenever the node status changes from available to unavailable, we are proposing a scheme wherein the system calls (operating system) are used to obtain the information regarding present node details at runtime dynamically. This information is stored on each system node. The granularity obtained is up to the core level.

The underlying system configuration is exploited to fetch the information regarding various application reads and allotting them to certain processors with fewer loads for their execution. This allocation is done in hierarchical fashion by employing a binary tree to fetch the least load processor. The thread then executes on that processor and proceeds to completion.

The proposed HTPE works in tandem with the runtime system. The data structures form integral part of the runtime system which are obtained by HTPE for effective placement strategy where in the system with minimum load is considered for thread execution.

Various incoming application threads are first collected in a Non- Blocking Queue data structure. These threads are then sorted according to their priority, thus adhering to priority scheduling mechanism for threads. Now the non- blocking queue consists of application threads sorted according to priority from head to tail in decreasing order i.e. the highest priority thread is at the head position and the least priority thread is at the tail position. In this process all the threads related to single object and module are placed in sequence and care is taken in the design so that the related module and thread sequence are maintained as a group in the non-blocking queue.

As part of RTS, a library call is provided to obtain the number of worker threads in the TPE. If the present executable threads in TPE are less than corePoolSize, we can add requests from the non_blocking queue to the TPE on the system. If the worker threads in TPE are less than corepoolsize If the worker threads of the TPE are more than the corePoolSize then we can invoke thread migration mechanism of the RTS, which will place the threads on system's TPE (with minimal load) taking into account all the threads related to one object are placed on a single system. Whenever the TPE on the current node is unable to accept further tasks then it throws

RejectionExecutionHandler exception as part of exception handling. Consequently HDSM's RTS will invoke the thread placement mechanism to place the threads to other systems with minimal load.

The hierarchical thread pool executor will actually be part of multi-core operations layer and is invoked from the migration and consistency layer. The proposed HTPE will eventually be integrated into the proposed HDSM. However, with the use of Jackal [18], a compiler based implementation of DSM; this architecture can be deployed to operate on a cluster of workstations. This can either use JAVA's built in thread pool executor, or the proposed HTPE, which we presume to be integrated seamlessly as the proposed data structures are compatible for JACKAL DSM as well.

3.5 Hierarchical Thread Pool Algorithm:

This Algorithm gives a high level description of the hierarchical thread pool executor.

3.1 Algorithm HTPE:

Input: thread execution requests from various nodes including itself; //task to be executed//
Output: assignment of tasks to the worker threads;
Initialization:

```
core_pool_size = maximum cores in the system;
max_pool_size = user defined maximum worker threads // greater than
core_pool_size
max_number_entries_queue=max_que;
thread_pool_state=in
min_heap_structure = array of load on each system in DSM;
non_blocking_queue: double link list;
```

begin

```
While thread_pool_sate NOT(Shutt) //while thread pool is running//
{
Sort the double linked Non_blocking queue on priority with highest priority
thread using selection sort;
// While updating the Non_blocking queue use  atomic variable (semaphore) to
sort the queue based on priority;//

Get the number of active worker threads in operation in the current_threadpool
//available and not assigned to any task//

If the (number of threads < core_pool_size)
{
```

add the thread to present threadpool //assign it to the worker thread for execution//
delete the thread from non_blocking queue;
}

else if the (number of threads > core_pool_size) && (number of threads <max_pool_size)
{ obtain minimum load system *i* from the min_heap_structure
Call Threadmigration(i, tasks)
//this is done by calling migration layer of the DSM which passes the ready tasks to system i having less load and in turn the HTPE daemon running on that system using MPI calls//

delete these threads from non-blocking queue;
}
else
{
throw run_time_exception //remote possibility
}
If threadpool is idle
/"display the thread pool has no_work_to_do //
sleep 10 milli_seconds;
Threadmigration(i, tasks);
{
assign tasks to HTPE running on system i;
// this require a MPI call to system i
//thread is invoked on the designated system
return
}
}
state=shut;
//display " HTP was terminated "
Exit();
//end of non_blocking queue
End;

The following logic is used for Non_blocking queue opertations. Application using HTPE has to makes use of all the functions of Non_blocking queue.

3.2 Algorithm insert:

Input: tasks;
Output: updated queue;
begin
create a new node;
node.next_ptr=null// new end of chain//

```
//set the thread_state=current_thread//
//insert the node at the end of the chain//
      Check for the end of chain= null  then update tail.next to new_node;  and
return=true;
If rem_node=node.next_ptr;
{
// now check whether this is pointing to null
rem_node.next_ptr=new_node;
return=true;
end;
```

3.3 Algorithm delete:

```
Input: tasks;
Output: updated queue;
begin
start with head of the non-blocking queue get the first node's data;
for ;
:
if cur_node.next=null
{
 return  // exhausted no more nodes whose status is ready
else
{
        temp=cur_node.next
        cur_node.next=temp.next=cur.node.next  // Node.next=temp.next;//
        // pointer is updated hence node is deleted
        display ready node is deleted;
        return
}end;
```

3.6 Non-Blocking Queues

Non-blocking queues are just those that aren't *blocking*. *Non-blocking* data structures are those on which all operations are *non-blocking*. All *lock-free* data structures are inherently non-blocking. Non-blocking queues are built without using lock-free data structures.

3.6.1 Introduction

The traditional approach to multi-threaded programming is to use locks to synchronize access to shared resources. Synchronization primitives such as mutexes, semaphores, and critical sections are all mechanisms by which a programmer can ensure that certain sections of code do not execute in parallel, if doing so would corrupt shared memory structures. If one thread attempts to acquire a lock that is already held by another thread, the thread will block until the lock is free.

Researchers have addressed these problems by introducing non-blocking synchronization algorithm [89], which is not based on mutual exclusion. Non-blocking implementation of shared data objects is a new alternative approach to the problem of designing scalable shared data objects for multiprocessor systems. Lock-free algorithms are non-blocking [90] and guarantee that always at least one operation can progress, independently of the actions taken by the concurrent operations. Wait-free algorithms guarantee that all operations can finish in a finite number of their own steps, regardless of the actions taken by the concurrent operations.

Non-blocking implementation makes use of non-blocking algorithm. An algorithm is non-blocking if the suspension of one or more threads will not stop the potential progress of the remaining threads. They are designed to avoid usage of a critical section. Often, these algorithms allow multiple processes to make progress on a problem simultaneously without ever blocking each other. For some operations, these algorithms provide an alternative to locking mechanisms.

Non-Blocking algorithms are concurrent algorithms that derive their thread safety not from locks, but from low-level atomic hardware primitives such as Compare-And-Swap (CAS). These algorithms can be extremely difficult to design and implement, but they can offer better throughput and greater resistance to liveness problems such as deadlock and priority inversion.

3.6.2 Implementation Aspects of Non-Blocking Queues

The implementation details and other execution elements related to the modules involved in the simulation of a hierarchical thread pool executor are listed and expanded hereunder. The key modules described are defining a Non-blocking queue

structure, insertion, deletion, display and sorting operations on threads in a Non-blocking queue.

```
public class ThreadParameters extends Thread
{
  int threadPriority;
  String threadState;
  String threadName;
}
```

Code Snippet 1: Thread Parameters Class

3.6.2.1 Creation of Threads

The Runnable interface and the Thread class in Java provide support for multithreaded programming. In the current scenario, we make use of the Thread class for creating new threads. The Thread class creates a new thread of execution. In order for this to be achieved, we create a new class ThreadParameters that extends the Thread class and includes the required parameters within.

The thread classes that now intend to create a new instance and assign values for these parameters have to extend the ThreadParameters class which in turn extends the Thread class.

Code Snippet 2: A class extending Thread Parameters

3.6.3 Defining a Non-Blocking Queue Data Structure

```
private static class Node <E> //STRUCTURE OF NODE
{
 E item; //FIRST FIELD OF NODE-THE VALUE
 AtomicReference<Node<E>> next;//SECOND FIELD OF NODE-THE NEXT POINTER

 Node(E item, Node<E> next) //NODE CONSTRUCTOR
 {
  this.item = item;
  this.next = new AtomicReference<Node<E>>(next);
 }
}
```

A linked list definition of a Non-Blocking Queue Structure is specified in this section. The Non-Blocking Queue structure comprises of nodes with two fields namely the data item and the next pointer. These nodes possess a special property. They are not the usual nodes, but instead each of them is of the type AtomicReference which facilitates the atomic updating and operations on the nodes. The data field holds the

```
public class ProjectThreadDemo1 extends ThreadParameters
{
  public ProjectThreadDemo1(){}
  public ProjectThreadDemo1(int i,String str, String name)
    {
        threadPriority=i;
        threadState=str;
        threadName=name;
    }

  public void run()
    {
        System.out.println("#####THE EXECUTION OF"+" "+threadName+ " "+" IS COMPLETED#####\n");
    }
}
```

required data. In this implementation, a generic template has been declared for the

Code Snippet 3: The Node Structure

data item field. But the actual information that is being stored belongs to the thread instances and their parameters. The next pointer holds a pointer to the next node in the queue structure. The value of this field is null if it is the last node in the queue. Since the pointer always points to another AtomicReference node, the type of the next field is another AtomicReference node. A constructor for the node structure is also declared that assigns values to the two fields for each (priority and state) of the nodes. This entire structure of the node is declared as private and static, so that the structure remains unmodified throughout the implementation process. The corresponding lines of code are illustrated in code snippet 3.

This declaration of a non-blocking queue node can be used for further creation of similar nodes with different values for the respective fields. However to begin with, every non-blocking queue has a dummy node as the first node or at the beginning of the list. The values of the data item field and the next pointer field for this dummy node are both initialized to null and remain the same irrespective of any number of insertions and deletions that are performed on the queue. Therefore this dummy node is declared as a final node and remains constant throughout.

Here is also defining two other important pointers, namely the head pointer and the tail pointer. As always, the head pointer points to the first node in the list and the tail pointer points to the last node. But in the initial case, when there are no nodes in the list, the head and tail pointers both point to the dummy node. The concerned snapshot is given in code snippet 4.

```
private final Node<E> dummy = new Node<E>(null, null);//DUMMY NODE
private AtomicReference<Node<E>> head = new AtomicReference<Node<E>>(dummy);//HEAD POINTER
private AtomicReference<Node<E>> tail= new AtomicReference<Node<E>>(dummy);//TAIL POINTER
```

Code Snippet 4: Declaration of the dummy node and the head and tail pointers

Now that the basic structure needed for the implementation has been defined, further operations like insertion and deletion can be performed on the queue data structure. The following sections deal with the operations- insertion, deletion, sorting and display to be performed on the Non-Blocking Queue data structure.

3.6.4 Insertion into a Non-Blocking Queue

Once the data structure for the non-blocking queue is defined, the prominent operation that is to be performed is insertion. This operation inserts thread instances that are created by different thread classes into the non-blocking queue. The insert method takes a generic item but here in particular, it takes a thread object as a parameter. Then, a new node is created and the data field for that node is assigned with the parametric value and the next field is initialized to null.

Subsequently the tail item is fetched and in case the node under consideration is not the tail node then we fetch the node immediately following the current tail node. The latter situation arises when multiple threads try to access the queue almost simultaneously. On the other hand if the tail pointer is pointing to the actual tail node at the moment then the updating has already occurred. In such a case, we perform compare and swap atomic operation and update the next pointer of the current tail to point to the newly inserted node. Also the tail pointer itself is updated and the new node becomes the new tail node. If there are nodes that still follow the current tail then we update the tail node first and then continue with the new-node insertion operation. A unique action that occurs here in the use of atomic hardware primitives is that the threads share their tasks and help the previous thread complete its job before it could actually proceed with its own.

```
Node<E> currentTail = tail.get();
Node<E> rmngNode = currentTail.next.get();
   if (rmngNode == null)//THERE ARE NO NODES FOLLOWING THE CURRENT TAIL
   {
        if (currentTail.next.compareAndSet(null, newNode))//UPDATING THE NEXT POINTER
        //COMPARE "CURRENTTAIL.NEXT" WITH "NULL" AND IF THEY ARE EQUAL SET "NEWNODE" AS "CURRENTTAIL.NEXT"
        {
          tail.compareAndSet(currentTail, newNode) ;//UPDATING THE TAIL
          //COMPARE "TAIL" WITH "CURRENTTAIL" AND IF THEY ARE EQUAL SET "NEWNODE" AS TAIL

             return true;
        }
   }
   else //IF THERE ARE NODES FOLLOWING THE CURRENT TAIL
   {
        tail.compareAndSet(currentTail, rmngNode);
        return false;
   }
```

Code Snippet 5: Insertion in a Non-Blocking Queue

3.6.5 Sorting in a Non-Blocking Queue

The insertion operation in a non-blocking queue will put all the threads together, thus forming a pool of threads, but there is no proper order that is followed in the allocation of threads in the queue. There has to be some sort of allocation mechanism or scheduling mechanism to make the threads run. We cannot allow the threads to run randomly; there may be many real-time threads whose immediate completion may be of great interest. So we first have to sort the threads according to some property, which will then make the whole idea of executing a thread meaningful. So for that reason, here we follow a priority based scheduling mechanism and sort the threads based on their thread priority. The highest priority thread is given the first slot in the queue and the least priority thread is given the last slot. Thus the threads are ordered or sorted in decreasing order of their priorities. A common selection sort procedure is followed to sort the threads in the non-blocking queue. Every iteration modifies the thread order and subsequent orderings will finally lead to a priority based sorted order of threads in the non-blocking queue. Here, we first compare the priority of the thread at the first node with that of the thread at the second node, and if the priority of the second is greater than the first then a swap occurs and the process starts again. This procedure continues until the first node holds the thread with the highest priority. Then the same procedure is followed for the second, third and so on until the last node, when the whole thread pool is finally sorted.

3.6.6 Deletion from a Non-Blocking Queue:

Once the thread is allocated to the TPE or it is successfully migrated this entry has to be removed from the non-blocking queue. The deletion operation and updating of pointer is given in code snippet 6.

```
System.out.println("\n\n*******PARAMETERS OF THREAD BEING DELETED******");
System.out.println("THREAD NAME:"+t.threadName);
System.out.println("THREAD STATE:"+t.threadState);
System.out.println("THREAD PRIORITY:"+t.threadPriority);
if(nhNode.next==temp.next)
  System.out.println("THREAD ALREADY DELETED AND POINTER ALREADY MODIFIED\n");
else
{
  nhNode.next=temp.next;
  System.out.println("THREAD DELETED, POINTER UPDATED");
}
```

```
System.out.println("****DELETED THREAD BEING SENT FOR EXECUTION****\n");
System.out.println("****BEFORE BEING SENT FOR EXECUTION CHECKING FOR STATE UPDATION****\n");
System.out.println("********ENTERING STATERANDOMIZE()********\n");
stateRandomize(t);
System.out.println("*********EXITING STATERANDOMIZE()********\n");
if(t.threadState=="Ready")
    {
      BinaryMinHeap bmh= new BinaryMinHeap(50);
      int y=bmh.identifyProcessor(bmh);
      System.out.println("****THREAD BEING ALLOCATED ON THE PROCESSOR WITH CURRENT LOAD " + y +" ****\n");
      t.start();
    }
else
    {
     insert((E)t);
     System.out.println("******INSERTING AGAIN DUE TO SUDDEN STATE UPDATION******\n");
     display();
    }
```

Code snippet 6: thread deletion and updating of pointers

3.6.7 Binary Min-Heap Implementation

The tree structure implemented as a Binary Min-Heap is actually intended to represent hierarchical structure of processors and their loads. Here, nodes represent processor loads and insertion and deletion of nodes is strictly according to the Heap mechanism, and ultimately the root node is the node that is returned since it represents the processor that has the least load.

```
heapSize++;
data[heapSize - 1] = value;
siftUp(heapSize - 1);
```

Code Snippet 7: Heap- Insertion (check numbers)

In insertion, we simply increment the heap size and insert the new processor load at the end following a complete binary tree criterion and then we move the element upwards in order to place it at the right position by successive shifts.

```
int child=0;
data[child]=data[heapSize-1];
heapSize--;
int childs;
for(child=0;child<=heapSize;child++)
   {
     for(childs=child+1;((data[child]>data[childs]) && childs<=heapSize-1);childs++)
      {
        int tmp=data[child];
        data[child]=data[childs];
        data[childs]=tmp;
      }
   }
```

Code Snippet 8: Heap- Deletion

In deletion, we always delete the root node and then immediately shift the last node element to the root position. We then perform successive comparisons and shifts until the whole structure satisfies the heap property.

```
if (nodeIndex != 0)
{
      parentIndex = getParentIndex(nodeIndex);
      if (data[parentIndex] > data[nodeIndex])
      {
            tmp = data[parentIndex];
            data[parentIndex] = data[nodeIndex];
            data[nodeIndex] = tmp;
            siftUp(parentIndex);
      }
}
```

Code Snippet 9: Heap Property Tracker

The shift up routine is written to check whether the tree satisfies the heap property i.e. the left children of a node should be less than the value at the parent node and the right children should have a value more than the parent node. This is called the heap property.

```
public int identifyProcessor(BinaryMinHeap bmh)
{
  bmh.insert(2);
  bmh.insert(7);
  bmh.insert(5);
  bmh.insert(2);
  bmh.insert(1);
  bmh.insert(8);
  bmh.insert(3);
  Random r= new Random();
     int j=r.nextInt(6);
  if(j==2 || j==4)
       bmh.insert(4);
  else if(j==1 || j==5)
       bmh.delete();
  int x=bmh.getMinimum();
  return(x);
}
```

Code Snippet 10: The identifyProcessor Function

The method identify Processor is the most important function. It returns the processor with the least load for the threads to be executed on that processor. It inserts some load values into the tree and then returns the value at root node i.e. the processor with the least load value. In order for this function to portray some sort of dynamism we perform some insertions and deletions randomly. But ultimately it is the node that represents the processor with least load on which the thread gets executed. This is illustrated in code snippet 10.

3.6.8 Execution of Threads

The final step in the implementation process is the execution of a thread. If the thread is in ready state even after deletion and randomization, then there is no reason why the thread should not be executed. Thus the ultimate intention is to execute or run the thread on a particular processor. For this purpose, we firstly identify a processor with least load on which the thread can be executed. The identifyProcessor method of the BinaryMinHeap class mentioned in the previous section performs this required functionality. After the processor is identified, the thread is ready to run on that processor. In order to run a thread the start method of the Thread Class has to be called which in turn invokes the run method of the corresponding thread class represented by the invoking thread object. The following code illustrates this functionality.

```
BinaryMinHeap bmh= new BinaryMinHeap(50);
int y=bmh.identifyProcessor(bmh);
System.out.println("****THREAD BEING ALLOCATED ON THE PROCESSOR WITH CURRENT LOAD " + y +" ****\n");
t.start();
```

Code Snippet 11: Thread Execution

When the start method is called by a thread object, the run method pertaining to the invoking thread object is invoked in its respective class and the user written code in the run method of that thread class gets executed.

```
public void run()
  {
      System.out.println("#####THE EXECUTION OF"+" "+threadName+ " "+" IS COMPLETED#####\n");
  }
```

Code Snippet 12: The run() method in a thread class

Thus a simple message indicating that the thread has been executed is displayed to the user in this case. Thus the successful execution of a thread is accomplished and the intended functionality portrayed.

3.7 Thread Migration Implementation with HTPE based RTS

The proposed HTPE will be integral part of the RTS. On each node, a binary min heap tree is constructed as part of system initialization of RTS and updated. The RTS on the current system(node) updates number of processes running periodically (a configurable parameter for RTS). The related information of the present system is obtained from the operating system's *Proc data structure*. The current number of processes is sent to other systems using MPI interface and this information the current node will be updated on all other systems of HDSM. This information is stored in the form of min-heap tree structure. The min-heap tree is a hierarchical tree representing the hierarchy of the systems configured for HDSM. Similarly the current load of other systems is received by the RTS's min-heap module and updates the current min-heap tree of the RTS. Any system being configured or new node becoming operational is also communicated across HDSM nodes. This is done to cater to node failure and recovery cases for while DSM is running.

The information is updated in the Min-heap structure of current system periodically by using RTS system's call. The RTS in turn gets the information from the underlying Operating systems Services layer. The RTS after obtaining this information will pass on to the other systems using other RTS services, which may lead to call to communication libray module to send this to the Min-heap sub module of the RTS present on other system.

The thread migration occurs when the current system's TPE cannot accommodate additional threads. As part of this event processing the min-heap tree is traversed to find out the nearest system with minimum load. Once the system is identified the RTS will signal the Thread Migration module on the current system to invoke the necessary procedure for thread migration with target systems's name as one of input parameters. The related threads along with the datum are passed on to the migration layer. The HDSM's migration layer will invoke necessary steps for proper thread migration incorporating the HDSM's consistency mechanisms.

Once threads are migrated, an acknowledgement of the migration is sent by the recipient system to the present system's migration layer and in turn to the RTS. Once acknowledgement is received, the threads that are migrated are removed from the non-blocking queue and the HTPE will continue next operation for further thread placement. This completes the thread migration mechanism of the HDSM.

3.8 Testing of HTPE With Standard Benchmark Programs.

The proposed HTPE was tested with few standard multi-threaded applications like Matrix multiplication and N-queens and Satisfiability. T Test has been performed on gain factor with null hypothesis as mean gain factor as 20, with alternative gain factor >20 for all data set for HTPE and TPE for above applications. A performance improvement asymptotic gain of 20% is observed.

3.8.1 Testing of MATMUL with HTPE:

We developed an application, MATMUL which is a n × n square matrix multiplication application. This is developed as two independent applications. **First** as TPE application and **Second** integrating this with the HTPE as a callable method. The obtained results are tabulated for comparison.

#CPUs	Initial Corepool Size	Matrix Dimension	Execution time using HTPE(msec)	Execution time using TPE(msec)	Improvement % of performance w.r.t. TPE
2	2	16	**3**	**2**	-50
2	2	32	**5**	11	54.5455
2	2	64	**1**	5	80
2	2	128	**9**	17	47.0588

2	2	256	**33**	46	28.2609
2	2	512	**306**	390	21.5384
2	2	1024	**3155**	4025	21.6149
4	4	16	**2**	**1**	**-100**
4	4	32	**4**	8	50
4	4	64	**1**	3	66.6667
4	4	128	**9**	16	43.75
4	4	256	**32**	50	36
4	4	512	**303**	389	22.1079
4	4	1024	**3189**	4003	20.3347
6	6	16	**3**	**2**	**-50**
6	6	32	**3**	9	66.6667
6	6	64	**2**	6	66.6667
6	6	128	**5**	19	73.6842
6	6	256	**34**	49	30.6122
6	6	512	**315**	390	19.2307
6	6	1024	**3251**	4038	19.4898

Table 3.1. Matrix Multiplication with varying matrix size and varying CPUs.

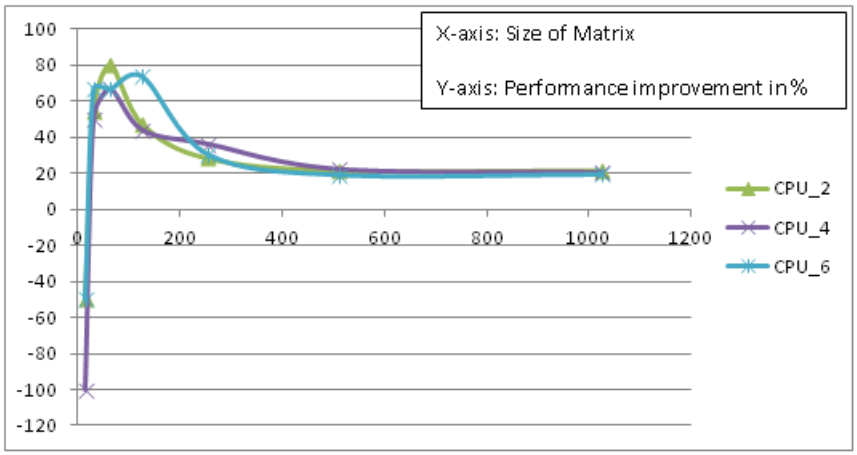


Figure 3.2 Matrix Multiplication Performance

3.8.2 Testing n-queens with HTPE

The Second Standard bench mark which was integrated and tested was n-queens problem. The nqueens problem is a java program which will place given number of queens on the chessboard without conflict using branch and bound technique. This was also tested as a single threaded program and was integrated with HTPE and the obtained results are tabulated for further analysis and comparison. The results are tabulated in Table 3.2

#CPUs	Initial Corepoolsize	N-Queens	Execution time using HTPE(secs)	Execution time using TPE(secs)	Improvement % of performance
2	2	9	**0.024**	0.022	-9.09091
2	2	10	**0.026**	0.052	50
2	2	11	**0.052**	0.106	50.9434
2	2	12	**0.203**	0.299	32.107
2	2	13	**1.073**	1.342	20.0447
2	2	14	**5.986**	7.303	18.0337
2	2	15	**37.332**	47.486	21.3831
2	2	16	**280.393**	345.369	18.8135
2	2	17	**2085.745**	2702.898	22.833
4	4	9	**0.021**	0.043	51.1628
4	4	10	**0.023**	0.067	65.6716
4	4	11	**0.096**	0.135	28.8889
4	4	12	**0.279**	0.438	36.3014
4	4	13	**1.05**	1.609	34.7421
4	4	14	**4.015**	7.146	43.8147
4	4	15	**22.032**	42.196	47.7865
4	4	16	**148.711**	293.293	49.2961
4	4	17	**1018.742**	2038.762	50.0313
6	6	9	**0.044**	0.053	16.9811
6	6	10	**0.082**	0.09	8.88889
6	6	11	**0.168**	0.194	13.4021
6	6	12	**0.425**	0.483	12.0083
6	6	13	**1.148**	1.99	42.3116
6	6	14	**4.737**	10.347	54.2186
6	6	15	**24.797**	62.086	60.0602
6	6	16	**157.418**	420.496	62.5637
6	6	17	**1140.81**	2915.52	60.8711

Table 3.2 Nqueens Problem with varying CPUS and varying corePoolSize.

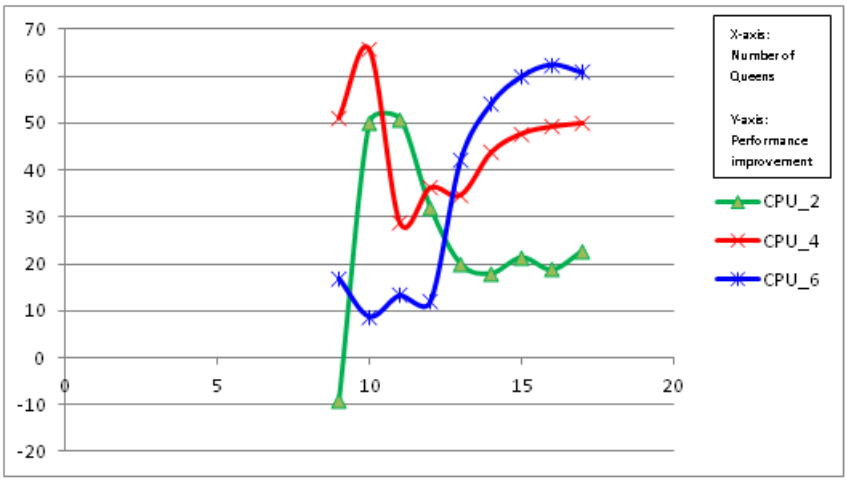


Figure 3.3 N-queens Performance

3.8.3 Testing HTPE with Boolean Satisfiability:

The Boolean Satisfiability problem checks for 30/28/26 equations to be compared for the Satisfiability condition. The Execution times for Boolean Satisfiability problem are given Table 3.3

#CPUs	Initial Corepool Size	Number of variables	Execution time using HTPE(secs)	Execution time using TPE(secs)	Improvement % of performance w.r.t. TPE
2	2	30	**161.2457**	**308.2223**	47.68526
2	2	28	**38.1463**	73.4667	48.07675
2	2	26	**8.5912**	17.6487	51.32106
4	4	30	**77.6554**	145.345	46.57175
4	4	28	**18.49013**	34.8576	46.95524
4	4	26	**4.1366**	8.06491	48.7086
6	6	30	**54.4096**	101.2311	46.25209
6	6	28	**12.8355**	**23.8858**	46.26305
6	6	26	**2.8902**	5.4902	47.35711

Table 3.3 Boolean Satisfiability Problem with varying CPUs and varying number of variables in SAT equation

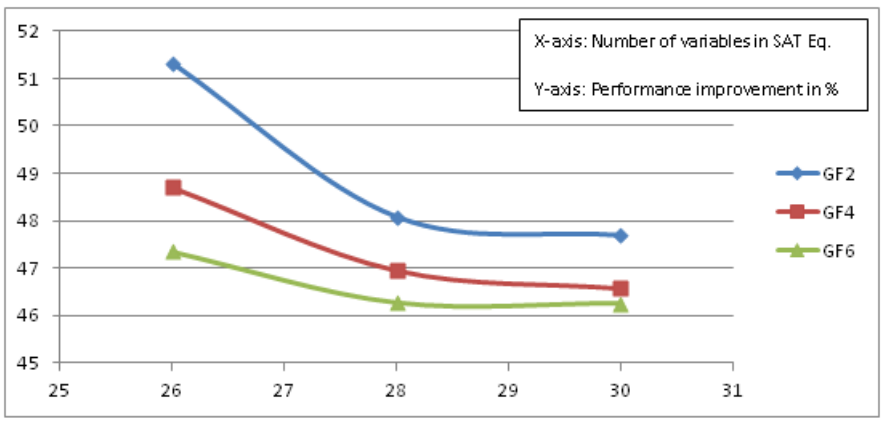


Figure 3.4 Satisfiability Performance for 2, 4 and 6 CPUs.

It is observed that the Gain factor for the Satisfiability the miniminm gain is 46%. The maximum gain observed is 51.321 for 6 CPUS. This shows that the HTPE performed better than TPE for all the cases of Satisfiability problem. We can also conclude that the HTPE performance is better with minimum of 46%.

3.9 Conclusions

In this chapter we have proposed a Hierarchical Thread Pool Executer (HTPE) that works on each node of the DSM. Whenever a new task arrives and if the worker thread of the system it is initiated is free, a thread is assigned to it. In case no worker thread is available, instead of storing them into a blocking queue, we store them into non-blocking queue. The tasks are sorted on their priority of execution in non-blocking queue. It checks for availability of worker threads into other systems HTPE daemon, this is accomplished by maintaining a common min-heap structure that stores the availability of worker threads. Thread migration takes place to a node with minimum workload. The algorithm is tested with variety of mixed programs and the result shows clear performance improvement compared to standard TPE.

Chapter 4: Dynamic Pre-fetching Strategies

4.1 Introduction

This chapter introduces pre-fetching in general and proposes n-gram based pre-fetching techniques in particular. It also highlights the applicability of pre-fetching in the domain of computer science applications. The need for enhancing the performance of HDSM has been dealt with in section 4.3. It is also shown that how stride history (sequence of page fault) can play an important role in designing the pre-fetching strategies.

Since an n-gram can be used to obtain the frequency of a particular pattern of page faults, this motivated us to propose a novel n-gram model based pre-fetching strategy for HDSM which is explained in section 4.6. For the completeness purpose the overview of n-gram models and their applications in Computer Science domain are presented in section 4.6. The performance evaluation of proposed n-gram pre-fetching model has been consolidated in 4.8.

The following section provides an overview of pre-fetching.

4.2 Pre-fetching:

Most (but not all) pre-fetching schemes are designed to fetch data from main memory into the processor cache in units of cache blocks. However, those multiple word cache blocks themselves are a form of data pre-fetching. By grouping consecutive memory words into single units, caches exploit the principle of spatial locality to implicitly prefetch data that is likely to be referenced in the near future.

A major overhead of software DSM/HDSM is the long remote access latency when the accessed page is not in the cache. One method for tolerating the remote access latency is to prefetch the pages before they are accessed. The advantages of prefetch include the reduction of page fault handling time and the reduction of the number of

messages. All applications are benefited from the pre-fetching in overall running time, and may achieve considerable performance improvement.

The run time system is the final sub layer of DSM, where in the user's application program is executed. The aim of any user written application (including DSM) is to provide a mechanism for faster and efficient execution while utilizing the available resources in an effective or optimal way. Any methods employed/implemented are meant to achieve this goal.

We know that pre-fetching is the process of making the required data/program code available in cache memory in advance to avoid memory stalls. For example, the Manta RTS [92] finally is able to execute the object's method of the user's application. If we are able pass this user's object reference to the RTS and if RTS is able to place this thread on requisite node (where the node's number of processes/threads running is minimum), then improvement on part of scheduling of the processes leads to performance improvement.

However, alternatively as a design choice, we can use dynamic model based pre-fetching as well, which leads to effective utilization of CPU time by the processor. We may recall that the memory references for both data and the program are taken from cache rather than fetching from the main memory. This is true because, as the cache memory access is roughly more than 10 fold than its main memory access time (which is exactly done by prefetch filling cache with required datum).

It is well known that once memory stall occurs, it takes several milliseconds for the operating system to re-schedule the process/thread. We presume that the effect of memory stall avoidance using n-gram model based dynamic prefetcher will achieve this, though there is associated overhead of storing stride numbers and searching for its presence of stride in memory. DSM selectively employs the model based pre-fetching while it has been incorporated in the implementation of RTS.

The detailed mechanism of various pre-fetching Techniques and their relative merits are covered in detail in section 4.2.1 through 4.2.3 of this chapter.

4.2.1 Hardware Pre-fetching:

Hardware pre-fetching is a prediction process: given some current prediction state, the prefetcher guesses what a future memory reference may be and requests that location from the memory subsystem. A major advantage of hardware techniques is that they need no support from the programmer or compiler. However, hardware pre-fetching wherever proposed requires additional hardware to be provided (mostly in the form of content addressable memory) for storing the frequently referred data in the processor in the form of tables. Hence it finally becomes processor dependent.

A hardware stride pre-fetching was proposed which makes use of hardware based stride prediction table (SPT) John W. C. Fu Janak H. Pate1 and Bob L. Janssens [87] and another hardware pre-fetching based on Reference Prediction Table (RPT) was also proposed by Fredrik Dahlgren and Per Stenstrom[88].

4.2.2 Software Pre-fetching:

Most contemporary microprocessors support some form of fetch instruction which can be used to implement pre-fetching. The implementation of a fetch can be as simple as a load into a processor register that has been hardwired to zero. Slightly more sophisticated implementations provide hints to the memory system as to how the prefetched block will be used. Such information may be useful in multiprocessors where data can be prefetched in different sharing states.

Although particular implementations will vary, all fetch instructions share some common characteristics. Fetches are non-blocking memory operations and therefore require a lockup-free cache that allows prefetches to bypass other outstanding memory operations in the cache. Prefetches are typically implemented in such a way that fetch instructions do not cause exceptions. Exceptions are suppressed for prefetches to insure that they remain an optional optimization feature that does not affect program correctness or initiate large and potentially unnecessary overhead, such as page faults or other memory exceptions.

The hardware required to implement software pre-fetching is modest compared to other pre-fetching strategies. Most of the complexity of this approach lies in the

judicious placement of fetch instructions within the target application. The task of choosing a place in the program to place a fetch instruction relative to the matching load or store instruction is known as *prefetch scheduling.*

In practice, it is not possible to precisely predict when to schedule a prefetch so that data arrives in the cache at the moment it will be requested by the processor. The execution time between the prefetch and the matching memory reference may vary, as will memory latencies. These uncertainties are not predictable at compile time and therefore require careful consideration when scheduling prefetch instructions in a program.

Fetch instructions may be added by the programmer or by the compiler during an optimization passes. Unlike many optimizations which occur too frequently in a program or are too tedious to implement by hand, prefetch scheduling can often be done effectively by the programmer. Studies have indicated that adding just a few prefetch directives to a program can substantially improve performance. However, if programming effort is to be kept at a minimum, or if the program contains many pre-fetching opportunities, compiler support may be required.

Software-controlled data pre-fetching is a promising technique for improving the performance of the memory subsystem to match today's high-performance processors. The software pre-fetching is designed and implemented using the processor's prefetch instructions. As part of modern processor's Architecture variant of Prefetch instructions are provided as part of instruction repertoire for the processor.

The prefetch instruction will be generated by the Compiler or user program or application layer of the software such as DSM. RTS of DSM will be able to provide an interface for the pre-fetching to aid the faster and efficient execution of the application.

Software based DSMs using Pre-fetching was carried out for home based JIAJIA based on history pre-fetching and aggregate pre-fetching was implemented by Haiming Liu and Weiwu Hu [84]. For TREADMARKS SDSM an adaptive++ pre-fetching was implemented by Ricardo Bianchini, Raquel Pinto, and Claudio L.

Amorim [85],which is based on past history of memory access faults. A review of few research papers on software based pre-fetching are already presented in Chapter 2.

Alternatively it could also be model based pre-fetching, which is dealt in subsequent section 4.2.3 of this chapter.

4.2.3 Model based Pre-fetching:

Several approaches ranging from dynamic hardware to static software mechanisms have been studied in the past. A stand alone dynamic software data pre-fetching solution Entirely SOftware and DYnamic data Pre-fetcher (ESODYP) was proposed by Beyler et.al. [86]. ESODYP is based on a memory strides Markov model. It runs in two main phases: a short training phase where a graph coding sequences of occurring memory strides is constructed and an optimizing phase where predicted addresses are pre-fetched. Information in the graph is updated continuously by monitoring the strides accessed.

We have taken ESODYP as a reference model for both implementation and comparison because the ESODYP employs dynamic pre-fetching. ESODYP is Markov chain based model keeps track of the addresses that are pre-fetched. Model need to place these addresses referenced in a memory resident data structure. Since placing every memory address reference in memory will lead to large memory and searching for the referenced address present in the list consumes more time, *the model uses the stride of addresses.*

Many earlier reported works have shown that data pre-fetching can be an efficient answer to the well-known memory stalls. If one can reduce these stalls, it leads to performance improvement in terms of overall execution time for a given application.

Pre-fetching is not only restricted to fetching data from main memory into a processor cache. It is a generally applicable technique used for moving memory objects up in the memory hierarchy before they are actually needed by the processor. Pre-fetching mechanisms for instructions and file systems are commonly used to prevent processor stalls.

The detailed explanation and importance of pre-fetching is dealt in various sections of the present chapter.

4.3 Necessity of Pre-fetching in DSM:

The pre-fetching in DSM is desired because the time taken to handle a page fault is several times more than that of the access time of main memory or cache memory. If the required datum is available in cache then the time saved in terms of execution of the instruction is several folds the time required for executing instruction. We also need to ensure that any application package (including DSM) has to execute the user application in minimum time (faster execution**)**.

The primary benefit of pre-fetching in software DSM systems comes from its function of latency hiding. That is, by pre-fetching, we can overlap some of the communication with computation. As a result, the page fault handling time is reduced. Though the number of messages can be reduced a lot by merging several prefetch requests into one message, it seems this is not the primary contributor to the speed-up achieved by pre-fetching. This attributes to the performance improvements, mainly due to the aggregation of smaller messages into larger ones.

4.4 Motivation for n-gram based model

The n-gram model was popularly used by the researcher in the field of Information retrieval and speech recognition areas for the last 15 years. This has propelled us to explore the possibility of contributions related to information retrieval using n-gram model.

After detailed analysis with regard to usage of the ESODYP for the DSM, and to overcome the deficiencies/limitations, we propose to use n-gram model based pre-fetching for DSM. This gives better performance with reference to prediction of the referred addresses in the DSM.

Use of n-gram model for the dynamic pre-fetching gave us scope to evaluate our method vis-à-vis the existing ESODYP model. In other words we have taken ESODYP model as the reference model for both implementation and integration for

these applications and we have shown that the proposed n-gram model has better performance when integrated with the any of the existing DSM/HDSM for majority of the application types.

4.5 n-gram model

A Language model can be formulated as a probability distribution P(W) over word strings W that attempts to reflect how frequently a string W occurs as a sentence.

P(W) can be decomposed as $P(W) = P(w_1 w_2, \dots, w_n)$

$$P(W) = P(w_1)P(w_2|w_1)P(w_3|w_1 w_2) \dots P(w_n|w_1 w_2, \dots, w_{n-1}) \quad (1)$$

$$= \prod_{i=1}^{n} P\,(w_i|w_1 w_2, \dots, w_{i-1})$$

Where P(w_i|w_1,w_2,…,w_{i-1}) is the probability that w_i will follow given that the word sequence w_1,w_2,…,w_{i-1} was presented previously. In equation 1, the choice of w_i thus depends on the entire past history of input. For a vocabulary of size *v* there are v^{i-1} different histories and so, to specify P(w_i|w_1,w_2,…,w_{i-1}) completely, v^i values have to be estimated. In reality, the probabilities P(w_i|w_1,w_2,…,w_{i-1}) are impossible to estimate for moderate values of i, since most histories Where w_i,w_2,…,w_{i-1} are unique or occurred only few times.

A practical solution to the above problem is to assume that P(w_i|w_1,w_2,…,w_{i-1}) only depends on some equivalence classes. The equivalent class can be simply based on the several previous words w_{i-N+1},w_{i-N+2},…,w_{i-1}. This leads to an n-gram language model. If the word depends on the previous three words, we have tri-gram language model. Similarly we can have unigram: P(w_i) or bigram:P(w_i|w_{i-1}) language models.

The trigram is particularly powerful as most words have a strong dependence on two previous words and it can be estimated reasonably well with an attainable corpus.

In bigram model, we make the approximation that probability of a word depends only on identity of the immediately preceding word.

For a trigram model, the probability of a word depends on two preceding words. The trigram can be estimated by observing the frequencies or counts of word pair C(w_{i-2},w_{i-1}) and triplet C(w_{i-2},w_{i-1},w_i) as follows:

$$P(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2},w_{i-1},w_i)}{C(w_{i-2},w_{i-1})} \quad (2)$$

The estimation of equation 2 is based on maximum likelihood principle, because this assignment of probabilities yields trigram model that assists the highest probability to the training data of all possible trigram models.

We sometimes refer to value of *n* of an n-gram model as its order. This terminology comes from area of Markov models, of which n-gram models are an instance. In particular, an n-gram model can be interpreted as a Markov model of order n-1.

Predicting a word from previous *n* words in a sample of text, using n-gram based model on classes of words was done by Peter E.Brown, Peter V.deSouza *et.al* [62]. The vocabulary V words into C classes using a mathematical function which maps w_i with a class C_i. These classes can be merged by means of an algorithm to reduce the number of classes.

4.6 n-gram model based pre-fetching:

Conventionally, n-gram model is used for speech recognition and sentence or phrase or word searching/occurrence in languages. It is used extensively in natural language processing and speech recognition. In this thesis we used n-gram model for stride pre-fetching. This is compared with the most recent ESODYP for performance evaluation.

4.6.1 Language Models

We propose to introduce the n-gram model with reference to language and speech recognition applications for ease of understanding the concepts.

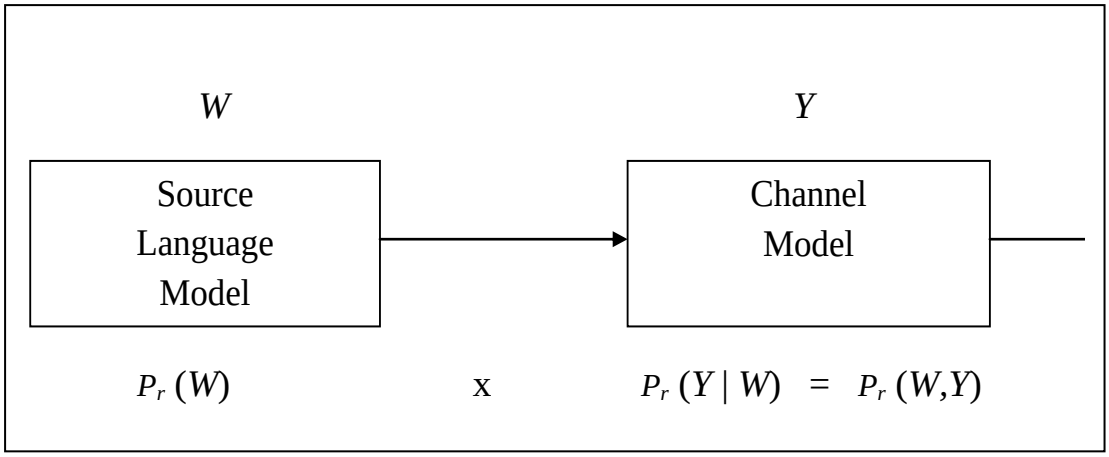


Figure 4.1 Source-channel setup

Figure 4.1 shows a model that has long been used in automatic speech recognition (Bahl, Jelinek, and Mercer 1983) and has been proposed for machine translation [62] and for automatic spelling correction (Mays, Demerau, and Mercer 1990). In automatic speech recognition, y is an acoustic signal; in machine translation, y is a sequence of words in another language; and in spelling correction, y is a sequence of characters produced by a possibly imperfect typist. In all three applications, given a signal y, we seek to determine the string of English words, w, which gave rise to it. In general, many different word strings can give rise to the same signal and so we cannot hope to recover w successfully in all cases. We can, however, minimize our probability of error by choosing as our estimate of w that string ŵ for which the a posteriori probability of ŵ given y is greatest. For a fixed choice of y, this probability is proportional to the joint probability of ŵ and y which, as shown in Figure 4.1, is the product of two terms: the a priori probability of ŵ and the probability that y will appear at the output of the channel when ŵ is placed at the input. The a priori probability of ŵ, $P_r(\hat{w})$, is the probability that the string ŵ will arise in English. We do not attempt a formal definition of English or of the concept of arising in English. Rather, we blithely assume that the production of English text can be characterized by a set of conditional probabilities, $P_r(w_k \mid w_1^{k-1})$, in terms of which the probability of a string of words, w_1^k, can be expressed as a product:

In an n-gram language model, we treat two histories as equivalent if they end in the same *n - 1* words, i.e., we assume that for $k \geq n$, $P_r(w_k|w_1^{k-1}) = P_r(w_k|w_{k-n+1}^{k-1})$Where P_r represents probability of word w′s occurrence. For a

vocabulary of size V, a 1-gram model has $(V-1)$ independent parameters, one for each word minus one for the constraint that all of the probabilities add up to 1. A 2-gram model has $V(V-1)$ independent parameters of the form $P_r\ (w_2|\ w_1)$ and $(V-1)$ of the form P_r(w) for a total of V^2-1 independent parameters. In general, an n-gram model has V^n-1 independent parameters: $V^{n-1}-1$ (V - 1) of the form P_r $(w_1\ |w_1^{n-1})$, which we call the order-n parameters, plus the V^{n-1} parameters of an (n - 1) gram model.

We estimate the parameters of an n-gram model by examining a sample of text t_1^T which we call the training text, in a process called training. If C(w) is the number of times that the string w occurs in the string t_1^T, then for a 1-gram language model the maximum likelihood estimate for the parameter P_r (w) is C(w)/T.

To estimate the parameters of an n-gram model, we estimate the parameters of the (n - 1)-gram model that it contains and then choose the order-n parameters so as to maximize $P_r(t_n^T|t_1^{n-1})$. Thus, the order-n parameters are

$$P_r(w_n|w_1^{n-1}) = \frac{C(w_1^{n-1}w_n)}{\sum_w C(w_1^{n-1}\mathrm{w})} \qquad (3)$$

We call this method of parameter estimation sequential maximum likelihood estimation. We can think of the order-n parameters of an n-gram model as constituting the transition matrix of a Markov model, the states of which are sequences of n - 1 words.

Thus, the probability of a transition between the state $w_1, w_2, \ldots w_{n-1}$ and the state $w_2, w_3, \ldots w_n$ is $P_r(w_n \mid w_1 w_2, \ldots w_{n-1})$. The steady-state distribution for this transition matrix assigns a probability to each (n-1)-gram, which we denote $S(w_1^{n-1})$. We say that an *n-gram* language model is *consistent* if, for each string w_1^{n-1}, the probability that the model assigns to w_1^{n-1} is $S(w_1^{n-1})$. Sequential maximum likelihood estimation does not, in general, lead to a consistent model, although for large values of T, the model will be very nearly consistent.

As n increases, the accuracy of an *n-gram* model increase, but the reliability of our parameter estimates (sequential maximum likelihood) decreases as they must be drawn from a limited training text. Jelinek and Mercer (1980) describe a technique

called *interpolated estimation* that combines the estimates of several language models so as to use the estimates of the more accurate models where they are reliable and, where they are unreliable, to fall back on the more reliable estimates of less accurate models. If $P_r^{(j)}(w_i \mid w_1^{i-1})$ is the conditional probability as determined by the *j*th language model, then the interpolated estimate, $\hat{P}_r(w_i \mid w_1^{i-1})$, is given by

$$P_r(w_i|w_1^{i-1}) = \sum_j \lambda_i w_1^{i-1} P_r^{(j)}(w_i|w_1^{i-1}) \qquad (4)$$

Given values for $P_r^{(j)}$, the $\lambda_j(w_1^{i-1})$ are chosen, with the help of the EM algorithm, so as to maximize the probability of some additional sample of text called the *held-out* data(Baum 1972, Dempster, Larid and Rubin 1977, Jelink and Mercer 1980). When we use interpolated estimation to combine the estimates from 1-,2-, and 3-gram models, we choose the λs to depend on the history, w_1^{i-1}, only through the count of the 2-gram, $w_{i-2}w_{i-1}$. We expect that where the count of the 2-gram is high, the 3-gram estimates will be reliable, and, where the count is low, the estimates will be unreliable.

4.6.2 n-gram Model Application to Stride Pre-fetching:

In modern architectures, perfecting is common technique that help the application program to bring a set of addressed into main memory prior to execution. These set of address are commonly known as stride. This process helps in saving the executing time of the running program by avoiding page faults. In computational paradigm stride plays an important role. If the stride is not in the cache memory, then the page fault will occur. On page faults the current state of the process is saved in Process Control Block (PCB), and the process status changes to hold. Consequently the system reads the required block from the disk drive that adds some delay of the order of milliseconds. Sometimes this leads to context switching to another process also, and the current process resumes the executing according to the OS scheduling policies.

In DSM the single address space is realized by combining the main memory of all participating stand alone systems. When an application program executes on this system, the required memory block may be local to the processor or remote. In both cases the block of the memory should be available in the main memory of the

processor. In case it is remote the block is obtained by the run time system and given to the processor. This is achieved by sending messages to the remote system and the data (perhaps stride) is returned in the reply.

If one is able to predict the sequence of memory accesses, there is a possibility of bringing the required strides into the memory before it has been referred by the processor. This is not straight forward and involves remembering the earlier access patterns. It is possible to achieve this to some extend by using Markov chain based model and was demonstrated in ESODYP.

One of the most recent ESODYP model (Section 4.2.3) suffers with many drawbacks, some of them are: It required a training phase (construction), which required initial time to build trees. Searching for the presence of strides takes considerable time

1. It requires more memory for storing the data structures used in the implementation.
2. Frequent flushing

All these problems can be addressed efficiently if we use the n-gram model (section 4.5).

Before going into technical details, let us consider the following illustrations which give the stride history of a process.

We have considered the random stride sequence of **1, 2, 16, 2, 32, 2, 16, 2, 32, 4, 9, 25, 36, 9, 4, 4** as a reference stride sequence for building n-gram model. A typical matrix representation of 1-gram model and 2-gram model for the above sequence is given in fig 4.2, 4.3and 4.4.

In the given $n \times n$ matrix above referenced stride sequence 1, 2, 16, 2, 32, 2, 16, the item (2, 16) = 2 indicate that the sequence 2 followed by 16 occurs twice (count = 2) in the above given sequence. Similarly item (32, 2) = 1 represents 32, 2 sequence occurs once in the above stride sequence (fig 4.2 one gram).

1-gram	1	2	4	9	16	25	32	36
1	0	1	0	0	0	0	0	0
2	0	0	0	0	**2**	0	2	0
4	0	0	1	1	0	0	0	0
9	0	0	1	0	0	1	0	0
16	0	2	0	0	0	0	0	0
25	0	0	0	0	0	0	0	1
32	0	1	1	0	0	0	0	0
36	0	0	0	1	0	0	0	0

Figure 4.2 1-gram representation for the above stride sequence

4,	1	2	4	9	16	24	32	36
4,1	0	0	0	0	0	0	0	0
4,2	0	0	0	0	0	0	0	0
4,4	0	0	0	0	0	0	0	0
4,9	0	0	0	0	0	1	0	0
4,16	0	0	0	0	0	0	0	0
4,25	0	0	0	0	0	0	0	0
4,32	0	0	0	0	0	0	0	0
4,36	0	0	0	0	0	0	0	0

9,	1	2	4	9	16	25	32	36
9,1	0	0	0	0	0	0	0	0
9,2	0	0	0	0	0	0	0	0
9,4	0	0	1	0	0	0	0	0
9,9	0	0	0	0	0	0	0	0
9,16	0	0	0	0	0	0	0	0
9,25	0	0	0	0	0	0	0	1
9,32	0	0	0	0	0	0	0	0
9,36	0	0	0	0	0	0	0	0

16,	1	2	4	9	16	25	32	36
16,1	0	0	0	0	0	0	0	0
16,2	0	0	0	0	0	0	2	0
16,4	0	0	0	0	0	0	0	0
16,9	0	0	0	0	0	0	0	0
16,16	0	0	0	0	0	0	0	0
16,25	0	0	0	0	0	0	0	0
16,32	0	0	0	0	0	0	0	0
16,36	0	0	0	0	0	0	0	0

25,	1	2	4	9	16	25	32	36
25,1	0	0	0	0	0	0	0	0
25,2	0	0	0	0	0	0	0	0
25,4	0	0	0	0	0	0	0	0
25,9	0	0	0	0	0	0	0	0
25,16	0	0	0	0	0	0	0	0
25,25	0	0	0	0	0	0	0	0
25,32	0	0	0	0	0	0	0	0
25,36	0	0	0	1	0	0	0	0

32,1	0	0	0	0	0	0	0	0
32,2	0	0	0	0	1	0	0	0
32,4	0	0	0	1	0	0	0	0
32,9	0	0	0	0	0	0	0	0
32,16	0	0	0	0	0	0	0	0
32,25	0	0	0	0	0	0	0	0
32,32	0	0	0	0	0	0	0	0
32,36	0	0	0	0	0	0	0	0
36,	1	2	4	9	16	25	32	36
36,1	0	0	0	0	0	0	0	0
36,2	0	0	0	0	0	0	0	0
36,4	0	0	0	0	0	0	0	0
36,9	0	0	1	0	0	0	0	0
36,16	0	0	0	0	0	0	0	0
36,25	0	0	0	0	0	0	0	0
36,32	0	0	0	0	0	0	0	0
36,36	0	0	0	0	0	0	0	0

Figure 4.3 2-gram representation for the above stride sequence

Similarly for 2-gram the node (36, 9, 4) =1 indicate that node sequence of 36 followed by 9 as 2-gram followed by 4 is occurring twice in the given sequence.

Similar tables can be generated and stored for observing for 3-gram, 4-gram… n-gram. From the above tables we can observe that the tables to be represented in the memory require multi dimensional array representation and thus occupy large memory. Especially when the application program requires large memory or frequent data references then stride sequence will be long and the table sizes for given n-grams increases exponentially.

4.6.3 Computation and Importance of Conditional Probability Transition Matrix (CPTM)

For a given gram (1-gram…n-gram), the number of stride references are stored in a linked list.

The frequency of occurrences of strides is represented in the form of a $n \times n$ matrix, where n represents stride number, indicating the sequence in which the strides are referred in the application. We use this matrix to compute Conditional Probability Transition Matrix (CPTM) using following procedure:

These lists are traversed for a given gram which is represented as a CPTM of size $n \times n$, where rows and columns represent the stride numbers and references to strides as a count. The element at (1,1) in 1-gram matrix represents stride-1 followed by stride-1 as a sequence, and how many times this stride sequence is referred by the application program.

We compute the sum of references of the strides for that row of strides and the conditional probability is defined as individual reference divided by the sum of the references for that row. We compute conditional probabilities of the different strides and sequences (based on gram number). As the stride is being accessed by the model, the address is mapped to stride number and the presence of the stride number in the list is checked by using a suitable hashing technique. If the stride is present, then its corresponding count is incremented by 1.

We also estimate conditional probabilities starting from 1-gram continuing to n-grams by computing conditional probabilities and then summing these conditional probabilities to check for termination. The termination condition for *n* will be the sum of conditional probabilities for which *n* will be unity. We also can arrive at the optimal gram for the given series of strides, which can be a configurable parameter.

Given stride list: 1 2 16 2 32 2 16 2 32
4 9 25 36 9 4 4

preparing 1 gram list(1)
1gram Probability Transition matrix.
Stride list: 1 2 c=1 p=1.000000
Stride list: 2 16 c=2 p=0.500000
Stride list: 2 32 c=2 p=0.500000
Stride list: 4 4 c=1 p=0.500000
Stride list: 4 9 c=1 p=0.500000
Stride list: 9 4 c=1 p=0.500000
Stride list: 9 25 c=1 p=0.500000
Stride list: 16 2 c=2 p=1.000000
Stride list: 25 36 c=1 p=1.000000
Stride list: 32 2 c=1 p=0.500000
Stride list: 32 4 c=1 p=0.500000
Stride list: 36 9 c=1 p=1.000000

preparing 2 gram list(1 2)
2gram Probability Transition matrix

Stride list: 1 2 16 c=1 p=1.000000
Stride list: 2 16 2 c=2 p=1.000000
Stride list: 2 32 2 c=1 p=0.500000
Stride list: 2 32 4 c=1 p=0.500000
Stride list: 4 9 25 c=1 p=1.000000
Stride list: 9 4 4 c=1 p=1.000000
Stride list: 9 25 36 c=1 p=1.000000
Stride list: 16 2 32 c=2 p=1.000000
Stride list: 25 36 9 c=1 p=1.000000
Stride list: 32 2 16 c=1 p=1.000000
Stride list: 32 4 9 c=1 p=1.000000
Stride list: 36 9 4 c=1 p=1.000000

preparing 3 gram list(1 2 16)
3gram Probability Transition matrix

Stride list: 1 2 16 2 c=1 p=1.000000
Stride list: 2 16 2 32 c=2 p=1.000000
Stride list: 2 32 2 16 c=1 p=1.000000
Stride list: 2 32 4 9 c=1 p=1.000000

Stride list: 4 9 25 36   c=1   p=1.000000
Stride list: 9 25 36 9   c=1   p=1.000000
Stride list: 16 2 32 2   c=1   p=0.500000
Stride list: 16 2 32 4   c=1   p=0.500000
Stride list: 25 36 9 4   c=1   p=1.000000
Stride list: 32 2 16 2   c=1   p=1.000000
Stride list: 32 4 9 25   c=1   p=1.000000
Stride list: 36 9 4 4   c=1   p=1.000000

preparing 4 gram list(1 2 16 2)
4gram Probability Transition matrix

Stride list: 1 2 16 2 32   c=1 p=1.000000
Stride list: 2 16 2 32 2   c=1 p=0.500000
Stride list: 2 16 2 32 4   c=1 p=0.500000
Stride list: 2 32 2 16 2   c=1 p=1.000000
Stride list: 2 32 4 9 25   c=1 p=1.000000
Stride list: 4 9 25 36 9   c=1 p=1.000000
Stride list: 9 25 36 9 4   c=1 p=1.000000
Stride list: 16 2 32 2 16   c=1 p=1.000000
Stride list: 16 2 32 4 9   c=1 p=1.000000
Stride list: 25 36 9 4 4   c=1 p=1.000000
Stride list: 32 2 16 2 32   c=1 p=1.000000
Stride list: 32 4 9 25 36   c=1 p=1.000000

preparing 5 gram list(1 2 16 2 32)
5gram Probability Transition matrix

Stride list: 1 2 16 2 32 2   c=1   p=1.000000
Stride list: 2 16 2 32 2 16   c=1   p=1.000000
Stride list: 2 16 2 32 4 9   c=1   p=1.000000
Stride list: 2 32 2 16 2 32   c=1   p=1.000000
Stride list: 2 32 4 9 25 36   c=1   p=1.000000
Stride list: 4 9 25 36 9 4   c=1   p=1.000000
Stride list: 9 25 36 9 4 4   c=1   p=1.000000
Stride list: 16 2 32 2 16 2   c=1   p=1.000000
Stride list: 16 2 32 4 9 25   c=1   p=1.000000
Stride list: 32 2 16 2 32 4   c=1   p=1.000000
Stride list: 32 4 9 25 36 9   c=1   p=1.000000

preparing 6 gram list(1 2 16 2 32 2)
6gram Probability Transition matrix

Stride list: 1 2 16 2 32 2 16   c=1   p=1.000000
Stride list: 2 16 2 32 2 16 2   c=1   p=1.000000
Stride list: 2 16 2 32 4 9 25   c=1   p=1.000000
Stride list: 2 32 2 16 2 32 4   c=1   p=1.000000
Stride list: 2 32 4 9 25 36 9   c=1   p=1.000000
Stride list: 4 9 25 36 9 4 4   c=1   p=1.000000

Stride list: 16 2 32 2 16 2 32 c=1 p=1.000000
Stride list: 16 2 32 4 9 25 36 c=1 p=1.000000
Stride list: 32 2 16 2 32 4 9 c=1 p=1.000000
Stride list: 32 4 9 25 36 9 4 c=1 p=1.000000

preparing 7 gram list(1 2 16 2 32 2 16)
7gram Probability Transition matrix

Stride list: 1 2 16 2 32 2 16 2 c=1 p=1.000000
Stride list: 2 16 2 32 2 16 2 32 c=1 p=1.000000
Stride list: 2 16 2 32 4 9 25 36 c=1 p=1.000000
Stride list: 2 32 2 16 2 32 4 9 c=1 p=1.000000
Stride list: 2 32 4 9 25 36 9 4 c=1 p=1.000000
Stride list: 16 2 32 2 16 2 32 4 c=1 p=1.000000
Stride list: 16 2 32 4 9 25 36 9 c=1 p=1.000000
Stride list: 32 2 16 2 32 4 9 25 c=1 p=1.000000
Stride list: 32 4 9 25 36 9 4 4 c=1 p=1.000000

preparing 8 gram list(1 2 16 2 32 2 16 2)
8gram Probability Transition matrix

Stride list: 1 2 16 2 32 2 16 2 32 c=1 p=1.000000
Stride list: 2 16 2 32 2 16 2 32 4 c=1 p=1.000000
Stride list: 2 16 2 32 4 9 25 36 9 c=1 p=1.000000
Stride list: 2 32 2 16 2 32 4 9 25 c=1 p=1.000000
Stride list: 2 32 4 9 25 36 9 4 4 c=1 p=1.000000
Stride list: 16 2 32 2 16 2 32 4 9 c=1 p=1.000000
Stride list: 16 2 32 4 9 25 36 9 4 c=1 p=1.000000
Stride list: 32 2 16 2 32 4 9 25 36 c=1 p=1.000000

preparing 9 gram list(1 2 16 2 32 2 16 2 32)
9gram Probability Transition matrix

Stride list: 1 2 16 2 32 2 16 2 32 4 c=1 p=1.000000
Stride list: 2 16 2 32 2 16 2 32 4 9 c=1 p=1.000000
Stride list: 2 16 2 32 4 9 25 36 9 4 c=1 p=1.000000
Stride list: 2 32 2 16 2 32 4 9 25 36 c=1 p=1.000000
Stride list: 16 2 32 2 16 2 32 4 9 25 c=1 p=1.000000
Stride list: 16 2 32 4 9 25 36 9 4 4 c=1 p=1.000000
Stride list: 32 2 16 2 32 4 9 25 36 9 c=1 p=1.000000

preparing 10 gram list(1 2 16 2 32 2 16 2 32 4)
10gram Probability Transition matrix

Stride list: 1 2 16 2 32 2 16 2 32 4 9 c=1 p=1.000000
Stride list: 2 16 2 32 2 16 2 32 4 9 25 c=1 p=1.000000

```
Stride list:  2 16  2 32  4  9 25 36  9  4  4   c=1          p=1.000000
Stride list:  2 32  2 16  2 32  4  9 25 36  9   c=1          p=1.000000
Stride list: 16  2 32  2 16  2 32  4  9 25 36   c=1          p=1.000000
Stride list: 32  2 16  2 32  4  9 25 36  9  4   c=1          p=1.000000
```

Figure 4.4 Stride lists along with conditional probabilities for grams 1 to 10.

The different fields in the figure 4.6 are stride numbers being referred by the model: c represents the count, and p represents the conditional probability for the stride under consideration. The sequence and the linked lists are created as per the gram level. The lists are organized in gram level and within gram level the node level hierarchy. Hence forth, searching the node availability is both automatic and complete for the nth level under consideration.

The sequence that is mentioned in the gram list

Stride list: 1 2 16 c=1 p=1.000000 is the valid gram considered by the data set under consideration for the given gram (ex: for a two gram the first entry is 1 2 followed by new stride 16 followed by count(c =1) followed by conditional probability 1.00 which is a real number.

The model after computing the conditional probability will be able to find out best probable stride for a given sequence. The computed stride can be pre-fetched in advance and the list is updated once the stride is pre-fetched. If the stride is not in the list then it is a clear case of a miss. Now this new stride is pre-fetched and if the stride is not present in the lists then the corresponding list is updated to indicate that the new stride is in the memory.

The model gets updated incrementally whenever a new stride arrives. We can also make the model configurable for maximum number of misses; once this pre-defined limit is reached, the model may be reset, so that the prediction can start afresh and the memory data structures can be released and training can start afresh (flushing).

Our method is efficient in the way as and when the new stride address is computed the whole process of computing and updating lists and computing probabilities are done in one single step in an effective manner. The maximum number of strides to be stored in the memory depends on dynamic memory allocated for storing the list, which again, will be a configurable parameter for a given application.

4.7 Dynamic Pre-fetch Algorithm

Algorithm: create i^{th} gram node

Input: stride number;
Output: updated stride list;
Pre Condition: partition the given stride list into sub-lists of size i-1 each;

```
{
   newNode = allocate memory enough for Node object;
  if ( newNode  is not NULL )
     {
      newNode ->strideList = allocate memory enough for i-1 strides ;
      copy(newNode->strideList, kth sublist ) ;
      newNode-> count = 0 ;
      newNode->conditionalProbability = 0.0;
      newNode->link = NULL ;
      newNode->status = UNFREEZED;
     }
   return newNode ;
}
```

Repeat

```
i=0;
{
pNew = Create ith gram node  with i+1 list size per each node;
Add pNew node to the list;
//compare the node with each element in the list
//scan the linked list for the ith gram and add the node at the end;
for ( each node temp  in the list )
{
    If (pNew->list = temp-> list )
       Update pNew->count
 }
/*Compute the conditional probability */
for ( each node temp  in the list )
{
   totalstridesequences = totalstridesequences + temp->count;
}
for (each node temp  in the list)
{
  temp->conditionalprobability = temp->count / totalstridesequenses;
 }
Increment i; // next gram
}
```

Until conditional probability of each gram is 1;
End;

The data structure used to represent the node (each stride is called as a node) in the form of a linked list is given below.

```
Struct  NodeType
```

```
{
    Float  prob;  //conditional probability
    int strideNo;    //stride number
    int count;                //How many times this stride is referred
    struct NodeType *nextNode;  //next node in the list struct       NodeType *listHead;  //head
          node
    struct NodeType *listTail;  //tail node
};
```

This data structure is used by the model to represent and update as and when the new memory reference is made by the application program. In HDSM, it is the RTS module used on behalf of the application program. For data access the HDSM (RTS) typically treats the object as one stride of size 256 bytes maximum, and it is assumed that the stride is prefetched as one entity (follows object level granularity).

The algorithm is designed in such a way that it works for any number of grams. It also has the flexibility of learning while being created. Since this is linked list based, the lists are updated/created as and when new stride arrives. If the stride which is referenced already exists then its count field is incremented. If it is a new node, it is placed in the appropriate gram's entry in the related sequence.

The n-gram model may take more time in case the lists grow and grams are also more. It is so designed that we can fine tune the model with the parameters for flushing. We will be able to arrive at a suitable mechanism for both searching the number of grams (as tunable parameters). We can also have the number of grams itself as a configurable parameter. Similarly, another configurable parameter is number of miss-predictions that is if the number of misses in the model are more than a predefined parameter, we can also flush the model and the performance improvement can once again be improved dynamically.

4.8 Result and Discussion

4.8.1 Testing with applications:

We have conducted experiments with the standard computational examples for comparison of the performance of the models under consideration. We have integrated popular ESODYP and proposed n-gram models with the standard

applications and the results obtained are presented in this section. We have studied both the models and detailed performance analysis was carried with these applications.

We have individually modified source code for these applications. We have incorporated necessary function calls for these applications appropriately to perform dynamic pre-fetching, create and maintain necessary memory resident data structures. These applications are generated as separate applications and were run independently. The results are obtained on the same data sets and environment to carry out the performance analysis.

1. **Transport problem:** The program computes the efficient and economical mechanism of moving goods from sources to destinations. The transport problem we considered has three sources and four destinations. This problem takes input cost matrix from each source to possible destinations. It then computes the overall cost for transportation. This involves several computations.

   The application program passes control to the model as a function call with the address as input; computation of stride number and maintenance of stride data structures and prediction is done by model. We integrated this application with the ESODYP model for monitoring the execution time and we have recorded the execution time with the ESODYP model.

   We have integrated the same transport application with the n-gram model and recorded the execution time. We have instrumented the necessary changes to pass control to the n-gram model and checked the execution time for this as well. We chose Linux as Operating System and included the system calls to compute the execution time of Transport application (with both models used for stride prediction). The execution time for the Transport problem with ESODYP model was 122 µseconds (micro-seconds). Similarly the execution time for the Transport Problem with n-gram model was 85 µseconds. **Thus over all speedup of about 1.453 is observed for this problem.**

2. **N-Queens Problem:** The N-Queen's problem is to find the number of ways in which the queens can be placed on a chess board. This also is a computational

intensive application with large memory access. We have implemented backtracking paradigm based code for both ESODYP and n-gram models and as a single application, the program is executed and the execution time is recorded. We have considered a maximum of 16 queens to be placed on a chess-board. We have incremented the number of queens starting from 1 to 16 and recorded the execution time with both ESODYP and n-gram models. The results of the study are given below for comparison. It is observed that the n-gram model is faster (for 3 queens) ESODYP takes about 10 µseconds as compared to 1 µseconds for n-gram, which is speed up 10.0.

No.of Queens	**n-gram (time in µsecs.)**	**ESODYP (time in µsecs.)**	**Improvement % of performance w.r.t. ESODYP**
2	1	143	99.300
3	1	10	90.00
4	1	9	88.888
5	1	8	87.500
6	1	11	90.909
7	5	18	72.222
8	16	42	61.904
9	59	140	57.857
10	235	545	56.880
11	2516	2555	0.0152
12	8993	10768	16.484
13	26554	27519	0.0351
14	145144	148999	0.0258
15	898657	918992	0.00212

Table 4.1 N-Queens problem performance with ESODYP and n-gram

It can be observed that the execution time for n-gram is approximately same as with the ESODYP because as the linked list size increases and grams increase, the formation time for the linked list and subsequent search time increases for this application (the same can be observed with any Depth First Search based problem). Moreover being the DFS algorithm (implemented using stack), every time the new page is brought in the main memory, the probability transition matrix is calculated and flushing takes place.

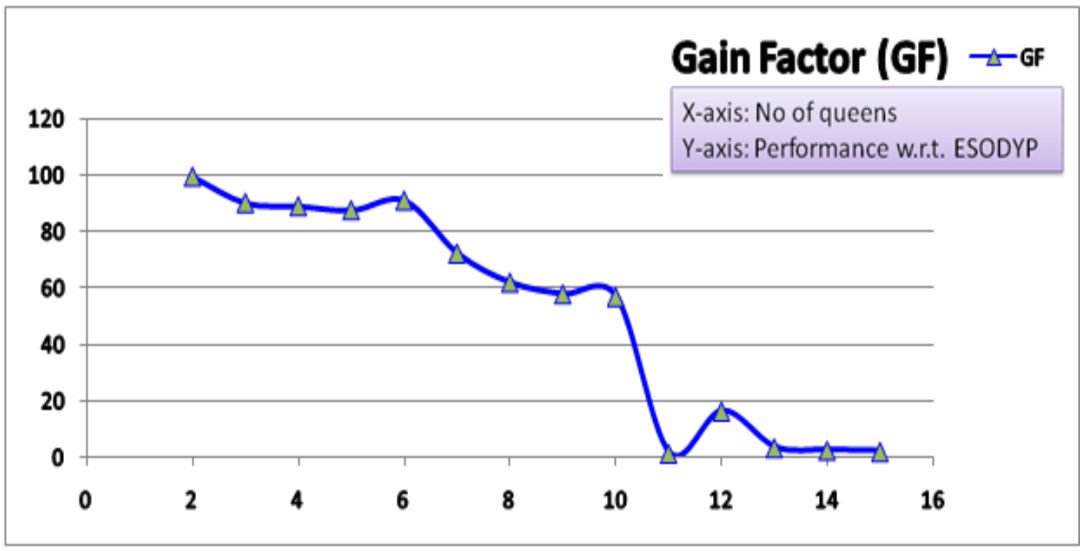


Figure 4.5 N-queens Gain Factor w.r.t. ESODYP.

3. **Satisfiability Problem:** Let $x_1, x_2,...,x_n$ denotes boolean variables. Let $\bar{x}_i$ denote the negation of x_i. A literal is either a variable or its negation. A formula in propositional calculus is an expression that can be constructed using literals and ***and*** or ***or***. The satisfiability problem is to determine if a formula is true for some assignment of truth values to the variables. The problem checks for 30 equations to be compared for the satisfiability condition. The Execution times for satisfiability problem are:
    - with ESODYP model 4:45.89 minutes (4 minutes 45.89 seconds) and
    - with n-gram model was 2:21.23 minutes. **Which shows a speed up is 2.02 times**.

4. **Devil:** This is a memory intensive program. This program creates two integer arrays of fixed size. Using a random function it generates three integer numbers within this range. Using these three numbers as indexes, it accesses the two integer array elements and sums up the two integer array elements and stores the sum in the third element of second array. This process is repeated for number of times of the size of integer array. This application process is repeated argument number of times by the main program of the Devil.

   Hence this can be truly called as a memory intensive program for testing performance of both the models. Sample test cases for both n-gram and ESODYP models (results of test cases) are enclosed below for reference.

No.of Iterations	**n-gram (time in µsecs.)**	**ESODYP (time in µsecs.)**
5000	0.19199	47.09705
50,000	1.86372	486.71996
500,000	23.737451	4988.62859

Table 4.2: Execution time Summary Table for Application DEVIL both ESODYP and n-gram

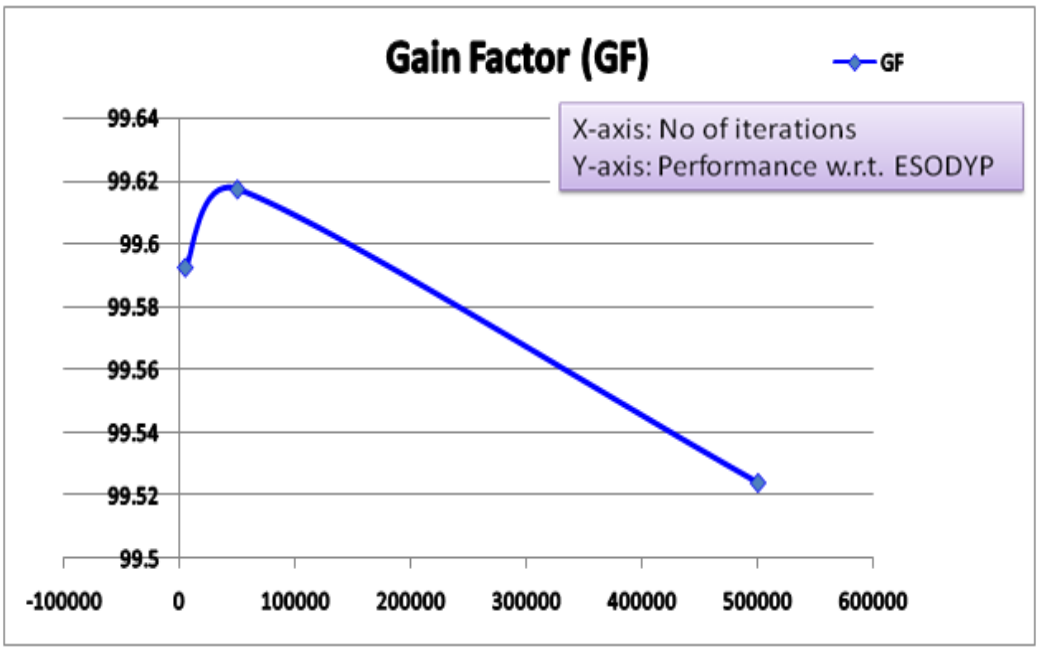


Figure 4.6 Performance Gain factor for Devil.

5. **Turtle:** Turtle is a CPU intensive program. This is an integer summation program written in C. The summation is done in function which is called for million times (10^7) and this is done using both the models for performance monitoring. The above function is called from the main program, the number of times given as a command line argument. For example,/Turtle 400 means for loop addition of integer is performed $400X10^7$ times and every time the models are called from these functions. Hence the performance time is recorded.

The test cases for both the models are included for reference. The time for executing Turtle with 1000 millions values, using n-gram model was 9.611885 seconds and with ESODYP was 78.634086 Seconds, which has a speed up of 8.180922 Times.

Conclusions

We can infer from the above referenced applications that the n-gram model performs much better when the memory references are more and significantly better than compared to the ESODYP for applications involving dynamic pre-fetching except for the application involve traversal of a graph, where the gain is less. The n-gram model gives better performance for applications that are memory intensive and/or

computational bound. We have integrated the applications with the model where memory stride references are required. This model can be integrated with Distributed Shared Memory implementation where the program and data require frequent pre-fetching for performance improvement. Since ESOYP model was already integrated into JACKAL, an object based DSM, on similar line; the proposed n-gram model can be integrated with the JACKAL SDSM. Operating systems process scheduling and any computational intensive and/or memory intensive applications are the best choices.

Chapter 5: Parallelization of n-gram Model

5.1 Introduction

This section we propose a novel parallel n-gram model, which has better time complexity compared to its sequential counterpart. We explain the mechanism of parallel algorithm along with complexity factor. The model's performance issues are also presented in this chapter.

The sequential dynamic pre-fetcher presented in Chapter 5 works on each node of the DSM. Since nodes of DSM can be muticore, it is appropriate to propose a parallel method that can take advantage of this kind of architecture.

Looking into sequential counterpart it can be observed that the process also involves inherent parallelism for computing the stride sequences. Also it was felt that the process of learning phase of the model where in model remembers the strides being referred can be optimized. Since we are working on a DSM system and many cores may be available on each node, the immediate extension of sequential pre-fetching to parallel pre-fetching can be realized. Since the parallelism is available at finer level, the most appropriate theoretical model for parallel computation is PRAM model, which is discussed in section 6.2.

While realizing n-gram for sequential pre-fetching, we used link list based structure for storing the stride lists related to the pre-fetching; here we store them in an array.

This lead to optimizing various options for speed up of accessing structures of the n - gram resulting better performance. It is assumed that proposed parallel n-gram will subsequently be integrated in to SDSM.

5.2 PRAM model

Many theoretical models of parallel computations appeared in the literature, a very basic and important model is Parallel Random Access Machine (PRAM) model. PRAM model is a straightforward and natural generalization of RAM. It is an idealized model of a shared memory SIMD machine.

Its main features are:

1. **Unbounded** collection of numbered RAM processors $P_0, P_1, P_2,...$
2. **Unbounded** collection of **shared** memory cells $M[0], M[1], M[2],....$
3. Each P_i has its own (unbounded) **local memory (registers)** and knows its index i.
4. Each processor can access **any** shared memory cell (unless there is an access conflict, see further) in **unit time**.
5. Input of a PRAM algorithm consists of n items stored in (usually the first) n shared memory cells.
6. Output of a PRAM algorithm consists of n' items stored in n' shared memory cells.
7. PRAM instructions execute in **3-phase cycles**.
    1. **Read** (if any) from a shared memory cell.
    2. **Local computation** (if any).
    3. **Write** (if any) to a shared memory cell.
8. Processors execute these 3-phase PRAM instructions **synchronously**.
9. **Special assumptions** have to be made about R-R and W-W shared memory **access conflicts**.
10. The only way processors can exchange data is by writing into and reading from memory cells.
11. P_0 has a special **activation register** specifying the maximum index of an active processor. Initially, only P_0 is active, it computes the number of required active processors and loads this register, and then the other corresponding processors start executing their programs.
12. Computation proceeds until P_0 halts, at which time all other active processors are halted.
13. **Parallel time complexity** = the time elapsed for P_0's computation.
14. Space complexity = the number of shared memory cells accessed.

PRAM is an attractive and important model [123] for designers of parallel algorithms, since

1. It is **natural**: the number of operations executed per one cycle on *p* processors is at most *p*.
2. It is **strong**: any processor can read or write **any** shared memory cell in unit time.
3. It is **simple**: it abstracts from any communication or synchronization overhead, which makes the complexity and correctness analysis of PRAM algorithms easier. Therefore,
4. It can be used as a **benchmark**: If a problem has no feasible/efficient solution on PRAM, it has no feasible/efficient solution on any parallel machine.
5. It is **useful**: it is an idealization of existing (and nowadays more and more abundant) shared memory parallel machines.

The PRAM corresponds intuitively to the programmer's view of parallel computer; it ignores lower level architectural constraints, and details, such as memory access and overhead, synchronization overhead, interconnection network throughput, connectivity, speed limits and link bandwidth, etc.

In PRAM model, although active, processors execute identical instructions; every processor has unique index and work on different data set. Hence such models can also be called as Single Instruction Multiple Data Stream models (SIMD). In one step each processor can either read from or write on into one memory location.

5.3 Fine grain parallelism in n-gram model

The n-gram sequential version prepares 1-gram list, 2-gram lists…n-gram lists sequentially. In the proposed parallel n-gram model, we make use of fine grain parallelism. It is important to observe that the forming of the 1-gram list need to be done sequentially and the other gram lists (2-gram…n-gram) are formed in parallel.

5.4 Parallel n-gram Algorithm

Algorithm Parallel Pre-Fetching()

Node gram[MAXGRAM][201]// max number of strides upper limit assumed as 200//

Input: array of stride numbers, maxgram;

Output: 2-D Array of gram (With stride sequence), Count //number of times stride sequence occurred;//

Begin

Step1: Create 1-gram using Sequential Pre-fetching algorithm //using sequential algorithm in Chapter 4//

Step2: Repeat

Step: 2.1: For j=2 to Maxgram do

Begin

Temp=gram(j,0)

Step2.1.1: **For all** i= 0 to length (gram(j-1))**in parallel do**

Begin

Step 2.1.1.1 Search in the gram(j-1)

If Index [Temp] matches element x in the gram(j-1) then CreateNode P by concatenating gram(j-1) with x;

Step 2.1.1.2 Insert a node P into gram(j-1);

end

Step 2.1.2 **For all** Pk (k=0 to n) **in parallel do**

ParallelComputeProbability(nodelist,k)

Until (conditional Probability==1)

end

End

Algorithm ParallelComputeProbability (nodelist, k)

Input: array of nodes //nodelist//

Output: nodelist with different groups having same stridelist sequence

Begin

Step1 For all element i in the list do

Step 1.1 temp.stridelist=list(i).stridelist

j=0

Step 1.2 Repeat

grouplist(j)=list(i)

increment i,j;

Until temp.stridelist<> list(i).stridelist

Step1.3 Create thread(k) and assign grouplist(k) to it.

Step1.4 Compute sum in its group

Step1.5 Compute Conditional probability for each grouplist

End

5.4.1 Parallel algorithm Complexity

We have estimated the parallel time complexity of the parallelization algorithm of n-gram model based pre-fetching and is given below.

While parallelizing the sequential pre-fetching algorithm, we assume the existence of unbounded parallel processors, each of type RAM.

- Sequential pre-fetching algorithm works for finding the conditional probability matrix. If the size of the list is **m** (the number of pages/strides brought in to main memory), and the number of grams is **n**, then time taken by the first iteration of the computation takes

$$\frac{((m-1)(m-2))}{2} + q$$

Where q is the number of steps required for grouping strides starting with same stride sequence.

- The overall complexity of the process, if it obtains the conditional probability matrix value as 1 can be given as

$$\sum_{i=1}^{n} \frac{((m - i)(m - (i - 1))}{2} + q$$

- While parallelizing the sequential pre-fetching algorithm, we assume the existence of unbounded parallel processors, each of type RAM.
- Step 2.1.1 of Parallel n-gram algorithm can be done in $O(\log_2 m)$ time as parallel summation operation require minimum $O(\log_2 m)$ time.
- Step 2.1.2 calls ParallelComputeProbability algorithm. With unbounded parallel processors step 1.3 of ParallelComputeProbability takes O(m-i) time.
- Overall parallel time complexity (using PRAM) of parallel algorithm is

$$\sum_{i=1}^{n} ((m - i) + \log_2 m)$$

using m processors.

- Generally it has been observed that the value of n does not go beyond 7 for all practical purposes.

5.5 Illustration Creation of grams

For ease of explanation the following stride sequence is considered.

1, 2, 16, 2, 32, 2, 16, 2, 16, 2, 16, 2, 32, 4, 9, 25, 36, 36 the corresponding index of the stride stored in array is given below (table 6.1).

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
strideno	1	2	16	2	32	2	16	2	16	2	16	2	32	4	9	25	36	36

Table 5.1: Index of the stride stored in array

While the stride numbers are repeated in the sequence but the index for a particular stride in sequence is unique. Referring to stride 2 which is repeated number of times the indexes for this stride are 1, 3, 5, 7, 9, and 11.

It is assumed that if strides are stored in integer array say stridelist[17] the index 0,1,2 represent the array index of stride 1,2,16 of the first three stride numbers respectively. The index number of the stride 2 when occurred first time is 1 and second time is 3 and third time 5 and so on. Hence index number of a stride sequence is unique.

While forming the lists, the index of stride is compared with already stored stride list to identify whether the sequence been repeated; this gives the stride sequence count.

5.5.1 1-gram sequence generation

This step is performed sequentially (step1 of algorithm). 1-gram list of stride sequence along with the indexes in **sorted order** is given below. That is stride sequence 1 followed by stride 2 (1, 2) along with corresponding indexes 0 and 1. An entry is made into the 1-gram sequence with indexes shown below. The stride sequence table is scanned for any presence of 1, 2 stride sequence. Since this is unique only one entry is made.

Index	Stride no
0,1	1,2

Table 5.2: One Entry stride

The next given sequence in the table is 2, 16 stride sequence. Its occurrence in the order of sorted index is given below, which will be entered in to the 1-gramlist as given table 6.3.

Index	Stride sequence
1,2	2,16
5,6	2,16
7,8	2,16
9,10	2,16

Table 5.3: Index entries for 2,16 stride sequence

This indicates that the stride sequence 2 followed by 16 has repeated 4 times with the corresponding indexes for this sequence are (1, 2), (5, 6), (7, 8), (9, 10). The above table is in sorted order on index and stride sequence as well. Now we can identify stride sequence 2, 16 uniquely by having the corresponding index pairs associated with stride sequence 2, 16.

Similarly for other stride sequences of 1-gram are given below.

Index	Stride sequence
3,4	2,32
11,12	2,32

Table 5.4: Index entries for 2, 32 stride sequence

Index	Stride Sequence
13,14	4, 9
14,15	9, 25

Table 5.5: Index entries for 4, 9 & 9, 25 stride sequences

While forming the 1-gram stride sequence we sequentially form the stride sequences in *sorted order* of their indexes. 1-gram stride sequence is formed sequentially in sorted order.

Index	Stride Sequence
2,3	16,2
6,7	16,2
8,9	16,2
10,11	16,2

Table 5.6: Sorted Stride Sequences

Index	Stride Sequence
15,16	25,36
4,5	32,2
12,13	32,4
16,17	36,36

Table 5.7: Sorted Stride Sequences

If we glance at the all the entries in 1-gram table, we find the sequences are sorted on stride numbers and also on their indexes as well.

1-gram Probability Transition matrix

Stride list: 1 2 c=1 p=1.000000
Stride list: 2 16 c=4 p=0.666667
Stride list: 2 32 c=2 p=0.333333
Stride list: 4 9 c=1 p=1.000000
Stride list: 9 25 c=1 p=1.000000
Stride list: 16 2 c=4 p=1.000000
Stride list: 25 36 c=1 p=1.000000
Stride list: 32 2 c=1 p=0.500000
Stride list: 32 4 c=1 p=0.500000
Stride list: 36 36 c=1 p=1.000000

5.5.2 2-gram to n-gram Sequence generation:

Inherent parallelism involved in 2-gram, 3-gram… n-gram sequences allow us to perform internal calculations in parallel from the 1-gram sequence tables, although they are obtained in sequence.

Parallel processes are initiated for each unique stride sequence, which means that the stride sequence 1, 2 will be one sequence and stride sequence 2, 16 is another sequence and 2, 32 is yet another sequence, etc.. Each of these stride sequences present in 1-gram will lead to invoking of parallel process which will read the 1-gram sorted sequence (PRAM model with CREW for parallel programming) along with the index numbers and the process of checking and forming a two gram sequence is carried out in the manner given below (step 2.1.1 of parallel n-gram algorithm).

Since we are constructing a 2-gram the stride numbers in sequence will start with 2 strides followed by 3rd stride number occurrences. The 2-gram stride sequence generation is a parallel process. The first occurred stride sequence (1, 2) is taken and its associated process identification (ID) 1 is assigned. Second stride which is different from first stride sequence (2, 16) generation is also invoked in parallel and is assigned process ID 2 and so on. It means that all these processes run in parallel and access the 1-gram stride sequence using PRAM model. Each of these processes will check for occurrences of particular the stride sequence and if found they are added to corresponding 2-gram stride sequences.

The representation of 2-gram is given below. For example parallel process 1 (process associated with stride sequence 1, 2 with their indexes as (0,1). We first check index of second stride number in the sequence 1, 2 which is stride 2 and take its index (1). Referring table 6.3 (sequential sorted 1-gram stride list), now check for other sequence in 1-gramlist , with index for stride number 2 compare this index with another sequence in one gram with same sequence of 2 (index being 1).

The first stride sequence 2, 16 and its first stride number(2)'s index 1, hence it matches with the current sequence (1,2,) (2,16) hence the stride sequence to be formed will be 1,2 followed by 2,16 hence we represent it as 1,2,16 this becomes the first element in the 2-gram list.

With this stride sequence 1, 2, 16 node is inserted into the 2-gram list (step 2.1.1.2). Now form the list and also record the corresponding indexes of all these three strides. The entry formed is given below.

Indexes	Stride sequence
0,1,2	1,2,16

Table 5.8: entry formed by process id 1

The process1 will continue to scan through the 1-gram list for other matching sequences starting with stride sequence 1,2 and if some more entries are found then these nodes are also inserted into the 2-gram stride sequence corresponding 1, 2 followed by index matching third stride number (step 2.1.1.2).

Process 1 will be associated with the one stride sequence formation. The entire 1-gram stride sequence is checked by the process1 which ultimately able to form all the stride sequences starting with 1, 2 stride sequence(for illustrated stride sequence only one entry).

Similarly the stride sequence starting with stride sequence 2, 16 with process number 2 will be constructed in parallel with process with id 2. It will search for matching sequences starting with 2,16 followed by third stride number and if there are valid entries the nodes will be inserted with 2,16 followed by the matching third stride number using index matching as explained above (parallel step 2.1.1.1 and 2.1.1.2). The next entry in table 6.6 with 16, 2 stride sequence is with index of 16 is 6 which is greater than previous entry for stride 16 being 2, hence to be considered as a separate entry.

The next entry in the 1-gram model for 2,16 is with the index for 16 being 6 and it matches with 16,2 second entry in table 6.6 hence a stride sequence of 2,16,2 is formed with indexes as 5,6,7 and is given below. With similar analogy the other entries of 2,16,2 are given table 6.9.

Index	Stride Sequence
1,2,3	2,16,2
5,6,7	2,16,2
7,8,9	2,16,2
9,10,11	2,16,2

Table 5.9: (Entries are formed by process id 2)

In effect the parallel process with id 2 will make the above 4 entries in the 2-gram stridelist (parallel step 2.1.1.2). Now check whether any more stride sequences with 2,16 with same index of second stride number 2 with index1 are matching. It can be seen from table 6.3,though there are other sequences 2,16 (three stride sequences present), all the indexes for 2,16 stride sequence is more than the current stride number 2's index(1) ,hence these should not be considered for possible 2-gram sequence.

Likewise the strides with other 2-gram sequences will work parallel and make entries into 2-gram list in parallel with process Id numbers 3 to n-1 (Parallel step2.1.1).

The count for stride sequence 2,16,2 is 4 and stored along with the stride sequence table for 2,16,2. With similar analogy, the other combinations of other possible stride sequences which will execute step 2.1.1.2 in parallel which form part of 2-gram stride lists given below for completeness.

Index	Stride sequence
3,4,5	2,32,2
11,12,13	2,32,4
13,14,15	4,9,25
14,15,16	9,25,36

Table 5.10: (Entries are formed by process id 2)

Index	Stride sequence
6,7,8	16,2,16
8,9,10	16,2,16
2,3,4	16,2,32
10,11,12	16,2,32

Table 5.11: (Entries are formed by process id 2)

Index	Stride sequence
15,16,17	25,36,36
4,5,6	32,2,16
12,13,14	32,4,9

Table 5.12: (Entries are formed by process id 2)

5.5.3 Count and Conditional Probability Computation:

After the 2-gram stride sequence generation is completed by the parallel processes (step 2.1.1), one more parallel activity being performed is computation of count and conditional probability. The parallel step 2.1.2 of parallel n-gram, we invoke parallel process for each stride sequence, that sequentially processes the stride sequences and if the stride sequence is repeated (same stride numbers with different indexes) then the count for the stride sequence is incremented(step 1.1&1.2 of ParallelComputeProbability).

For all processors in parallel we perform the count and conditional probability computation for that stride sequence (step 1.3.1 through 1.3.3) being achieved by associating a thread for each of the stride sequence under consideration. The conditional probability for the stride sequence is computed as sum divided by count for the given stride sequence. This is repeated for each gram's stride sequence with the termination condition for the gram's conditional probability of unity for all the stride sequences of that particular gram. This corresponds to step 2.2 of parallel n-gram algorithm.

The Taking an example of stride sequence of 2,32 (table 5.10) occurrence the 2,32 followed by 2 is once hence count is 1 and 2,32 followed by 4 also count is one, where as sum for 2,32 (2-gram) sequence is two. The conditional probability is computed as count divided by sum. Henceforth the conditional probability of sequences 2,32,2 and 2,32,4 will be 0.5 and 0.5 respectively. The detailed table representing count and conditional probability for the 2-gram stride sequence repeated here for better understanding.

preparing 2-gram list
2-gram Probability Transition matrix

Stride list: 1 2 16	c=1	p=1.000000
Stride list: 2 16 2	c=4	p=1.000000
Stride list: 2 32 2	c=1	p=0.500000
Stride list: 2 32 4	c=1	p=0.500000
Stride list: 4 9 25	c=1	p=1.000000
Stride list: 9 25 36	c=1	p=1.000000
Stride list: 16 2 16	c=2	p=0.500000
Stride list: 16 2 32	c=2	p=0.500000
Stride list: 25 36 36	c=1	p=1.000000
Stride list: 32 2 16	c=1	p=1.000000
Stride list: 32 4 9	c=1	p=1.000000

Table 5.13: 2-gram stridelist with count and conditional probability

The notable feature of this mechanism is that a thread can be assigned for forming the stride sequences in parallel.

5.5.4 Generation of n-gram sequence

The conditional probability of the entire sequences of n-1-gram is checked for unity. If this condition is satisfied this acts as termination condition for the generation of n-gram sequences. If conditional probability of any of the sequences is less than unity then the process next gram generation will be repeated.

Making use the n-1-gram sequences and using the above procedure used for 2-gramsequence generation, all the other gram's sequences are generated. A typical table involving gram number, conditional probability, count is given in table below for completeness of explanation.

preparing 3-gram list.

3-gram Probability Transition matrix

Stride list: 1 2 16 2	c=1	p=1.000000
Stride list: 2 16 2 16	c=2	p=0.500000
Stride list: 2 16 2 32	c=2	p=0.500000
Stride list: 2 32 2 16	c=1	p=1.000000
Stride list: 2 32 4 9	c=1	p=1.000000
Stride list: 4 9 25 36	c=1	p=1.000000

Stride list: 9 25 36 36 c=1 p=1.000000
Stride list: 16 2 16 2 c=2 p=1.000000
Stride list: 16 2 32 2 c=1 p=0.500000
Stride list: 16 2 32 4 c=1 p=0.500000
Stride list: 32 2 16 2 c=1 p=1.000000
Stride list: 32 4 9 25 c=1 p=1.000000

preparing 4-gram list
4-gram Probability Transition matrix

Stride list: 1 2 16 2 32 c=1 p=1.000000
Stride list: 2 16 2 16 2 c=2 p=1.000000
Stride list: 2 16 2 32 2 c=1 p=0.500000
Stride list: 2 16 2 32 4 c=1 p=0.500000
Stride list: 2 32 2 16 2 c=1 p=1.000000
Stride list: 2 32 4 9 25 c=1 p=1.000000
Stride list: 4 9 25 36 36 c=1 p=1.000000
Stride list: 16 2 16 2 16 c=1 p=0.500000
Stride list: 16 2 16 2 32 c=1 p=0.500000
Stride list: 16 2 32 2 16 c=1 p=1.000000
Stride list: 16 2 32 4 9 c=1 p=1.000000
Stride list: 32 2 16 2 16 c=1 p=1.000000
Stride list: 32 4 9 25 36 c=1 p=1.000000

preparing 5-gram list
5-gram Probability Transition matrix

Stride list: 1 2 16 2 32 2 c=1 p=1.000000
Stride list: 2 16 2 16 2 16 c=1 p=0.500000
Stride list: 2 16 2 16 2 32 c=1 p=0.500000
Stride list: 2 16 2 32 2 16 c=1 p=1.000000
Stride list: 2 16 2 32 4 9 c=1 p=1.000000
Stride list: 2 32 2 16 2 16 c=1 p=1.000000
Stride list: 2 32 4 9 25 36 c=1 p=1.000000
Stride list: 16 2 16 2 16 2 c=1 p=1.000000
Stride list: 16 2 16 2 32 4 c=1 p=1.000000
Stride list: 16 2 32 2 16 2 c=1 p=1.000000
Stride list: 16 2 32 4 9 25 c=1 p=1.000000
Stride list: 32 2 16 2 16 2 c=1 p=1.000000
Stride list: 32 4 9 25 36 36 c=1 p=1.000000

preparing 6-gram list
6-gram Probability Transition matrix

Stride list: 1 2 16 2 32 2 16 c=1 p=1.000000
Stride list: 2 16 2 16 2 16 2 c=1 p=1.000000
Stride list: 2 16 2 16 2 32 4 c=1 p=1.000000
Stride list: 2 16 2 32 2 16 2 c=1 p=1.000000
Stride list: 2 16 2 32 4 9 25 c=1 p=1.000000
Stride list: 2 32 2 16 2 16 2 c=1 p=1.000000
Stride list: 2 32 4 9 25 36 36 c=1 p=1.000000
Stride list: 16 2 16 2 16 2 32 c=1 p=1.000000
Stride list: 16 2 16 2 32 4 9 c=1 p=1.000000
Stride list: 16 2 32 2 16 2 16 c=1 p=1.000000
Stride list: 16 2 32 4 9 25 36 c=1 p=1.000000
Stride list: 32 2 16 2 16 2 16 c=1 p=1.000000

Table 5.14: stride list for 3-gram to 6-gram along with count and conditional probability

5.6 Integration of n-gram model in SDSM and Conclusion

The n-gram model based dynamic pre-fetching is carried out as and when a new address is chosen for pre-fetching. The model computes the stride number by subtracting the base address and this stride number is checked in the n-gram sequence list along with other existing strides. This process involves updating the stride sequences for various grams.

The model utilizes the array element's addresses for comparison to check the stride sequence presence in the gram lists. If the nodes are multi-core systems, we can perform formation of 2-gram to n-gram lists in parallel, there by exploiting inherent hardware resources otherwise being idle.

Once the lists are formed another parallel operation carried is computation of conditional probabilities step 2.1.2 of parallel algorithm. Once the conditional probabilities are computed in parallel we can check for termination of n-gram where in the all the conditional probabilities of the n^{th} gram being unity. This implies that prediction is deterministic.

If the number of grams is small as a configurable parameter one can choose to come out with most probable stride sequence and the most probable stride can be pre-fetched leading overall performance improvement.

The n-gram parallel pre-fetching is called selectively by the runtime system of the DSM for exploiting the combination of pre-fetching and communication latency overlapping. Hence forth the pre-fetching in general and parallel pre-fetching in particular will give better performance once these are integrated into the RTS of the SDSM.

The work is in progress to integrate parallel n-gram into existing DSM's and not covered in this thesis.

Chapter 6. Heterogeneous DSM Architectural Framework

6.1 Introduction

Design and implementation of Heterogeneous DSM is one of the popular distributed computing areas of research amongst researchers. The earlier research work carried was based on the realization of Heterogeneity and implementation for a particular case under consideration. For example if the two systems hardware is different, proper environment to be provided such that DSM software runs on both hardware platforms. Since it is implementation dependent, no two HDSM implementations were alike. In this chapter we are proposing a theoretical framework for the HDSM and present the component diagram. We also proposed new Hierarchical Thread Pool Executor and Pre-fetching strategies (Sequential and Parallel) as important components (Explained in the earlier chapters) from our proposed framework. The remaining components can be chosen as pluggable modules from other works, which are presented in this chapter.

6.2 Need for Unified HDSM framework

In a HDSM, typically the hardware platform on one node may likely to be different from other node. The operating system on these two nodes can also be different. Unless the implementation mechanism is same (ex: PVM based) it is impossible to integrate these two HDSM nodes. When an application program written in java is submitted to HDSM, the user need not be aware of the underlying software layers. HDSM can be configured with different nodes with different operating systems and its related layers. These DSM nodes need to interact to run user's application as part of realization of application run time environment. When the application program is executing, if the underlying application layer generates certain errors, it may result in either user application termination and/or will not be able to understand (cryptic) messages that are generated by incompatible HDSM nodes. Hence, it will also be difficult for the application programmer to either understand or analyze the related issues, or at least to identify the problem. This encouraged us to propose an integrated

framework which will enable user to select compatible pluggable components. Hence, we proposed an integrated framework for HDSM.

6.3 Components of Framework

The proposed integrated framework is broadly divided in to five layers. These are Application Layer, Object Management Layer, Migration and Consistency Layer, Multi-core Operations Layer and finally Operating System/Hardware Layer. The details of proposed framework are given in the figure 6.1.

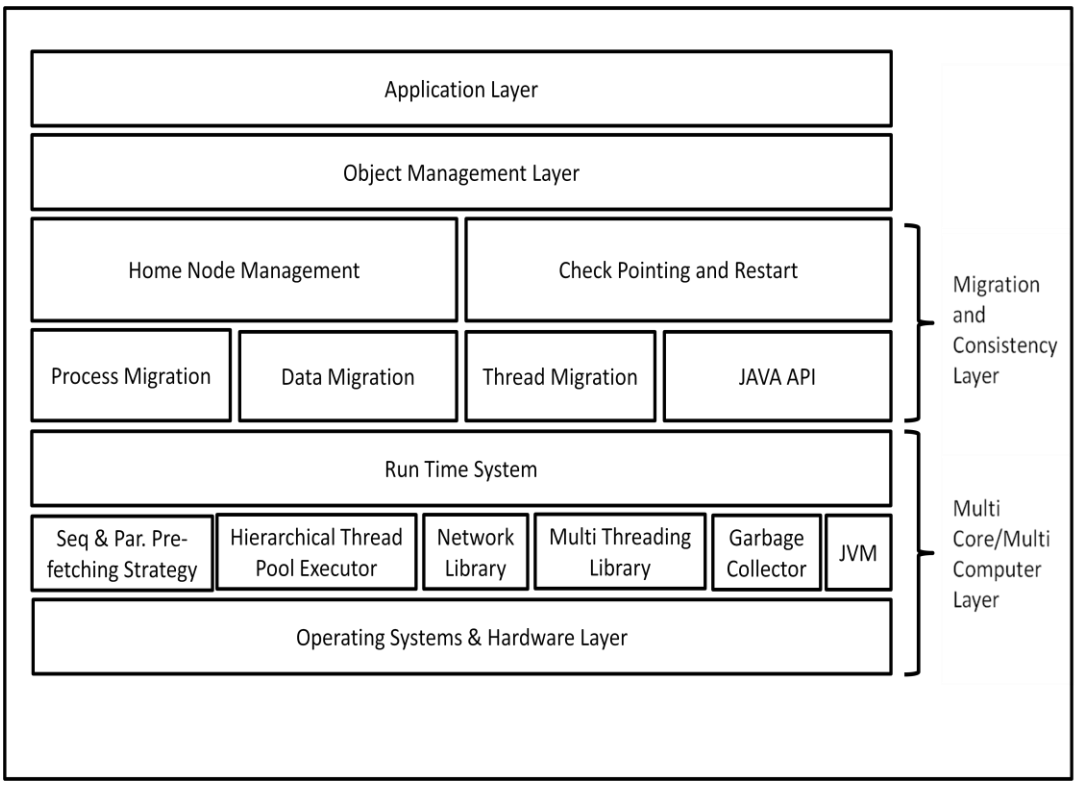


Figure 6.1 Proposed Heterogeneous Distributed Shared Memories Architectural Framework

The detailed explanation of each of these proposed layers in a Top-Down approach is given in the next section.

6.4 Application Layer

This is the layer with which user application interacts with the HDSM system. This layer deals with the necessary interface for user to submit the application. In this layer the user is provided with the necessary user interface and the required procedures to be followed as instructions for usage.

The user is supposed to know which HDSM flavor will be apt for the application under consideration and chooses the necessary user interface. The selection is based on what language is used for application development, the operating systems present on HDSM nodes, the type of application (thread based/process based) and communication layer that is preferred. Subsequently depending on the application the user can select the components required and once the user considers his/her choices then the necessary layers will be configured and job is ready to be deployed.

The different procedures and necessary instructions for the user to choose are provided as options. The related application interfaces, runtime systems along with available operating systems are configured. Prominently the available hardware architectures that form the necessary nodes for the HDSM for application execution are the final choices to be selected.

The Application layer is also responsible for the error log and system log that will record all the major events and statistics with reference to user application performance. This will become integral part of Application layer. The performance monitor applications and the errors and possible remedial measures to be followed for error recovery & analysis and limitations of the application will be provided to the user as part of application layer. This interacts with the immediate layer Object management layer.

6.5 Object Management Layer

This layer deals with the object management (creating an object, grouping and managing), which partitions the shared memory according to logical data structures. Object-based DSMs are most often implemented at the language (JAVA)/compiler

(used for inserting annotations for access checks) or library layers (user libraries or runtime system assisted).

The object based HDSMs can be of either C++ based or JAVA based. The runtime system and the HDSM nodes support either C++ or JAVA. The interactive user submission process of Application layer will provide the necessary environment. However, object-based DSMs do offer some unique and important features. An implementation at higher layers makes object based DSMs very flexible and system independent. The important DSM design parameters in object-based DSM are the block size, access method, communication mechanism, memory consistency model, etc... The DSMs can be directly implemented on top of existing generic communication subsystems, such as TCP/IP, PVM.

This layer employs techniques involved in automatic data partitioning & distribution, parallelism, extraction, and communication optimization. For this layer the necessary, implementations supporting primitives are provided. These are used in programming for distributing data, utilizing efficient memory consistency model, and performance tuning.

If the DSM is implemented using language JAVA, the DSM itself can be called as object based. The granularity of the DSM will be object level rather than page level. The consistency models can still be any of the sequential or multiple writer or Lazy release consistency depending, what memory model is chosen [18].

The other issue in implementing object-based DSMs is choosing a suitable communication subsystem. An ideal communication subsystem must be general enough and have well-defined communication interface, in which system details are encapsulated and are dealt in subsequent layers. This layer usually interacts with the Home node management layer/ Migration layer (as part Consistency maintenance) to identify the location of the object and updating the instances of the object.

6.6 Migration and Consistency Layer

This layer comprises of two sub layers namely migration and Consistency layers. The Migration layer consists of three type's viz. Process Migration, Data Migration and

Thread Migration. The other layer consists of Home Node Management and Check Point & Restart. The details of these are presented in subsequent sections.

6.6.1 Home Node Management

In all the nodes of the HDSM, copy of HDSM software along with the application software is pre-loaded and operational. Whenever a data item (object or variable) in application software changes, it may be necessary to update all the DSM nodes for subsequent updates and necessary recovery in case of failure. This is done by memory consistency sub-layer of the migration layer. Certain DSMS optimize this updating mechanism by updating in single copy of the DSM and to distinguish it from other copies this particular node is called as Home node. Examples of DSMs for home node based are JACKAL [18], SCASH [2].

JACKAL [18] is home node based SDSM which employs lazy release consistency model and it use dynamic home node mechanism. It is object based compiler directed SDSM. The home node is identified and then depending on how much data need to be updated on a node, the home node is migrated as and when required. The disadvantage of home node is cost of migration in particular dynamic home node migration and also the cost of maintaining the home node and mechanism of updating.

The Home node management HDSM layer is selected based on user's choice as a pluggable module, as part of user submission process in Application layer. This option is chosen by the user when the application's execution time and consistency are critical and the HDSM nodes are home node based.

6.6.2 Checkpointing and Restart

"Checkpointing" is the process of saving the state of a running application to "stable storage". At some later time, this saved state can be used to "restart" the application at the same point in the computation when the checkpoint was taken. This is one of the desirable features from few applications which are written keeping in view the application can be checkpointed.

Many operating systems provide a facility for the user process to be restarted from identifiable location (point) or a label from where the process can be started in case of a failure. These points are called as checkpoints**.** A checkpoint library is provided that must be linked with application code. Part of this library is signal handler that can create a checkpoint file of the process so that it can be restarted on a machine of compatible architecture and operating system.

When the user application in HDSM chooses Checkpointing and restart as one of the options, this pluggable module is selected. This is option is selected by the user in following scenarios. Firstly if application takes several minutes for execution, secondly application requires fault tolerance (due to likely failure of the subsystems) and finally the operating system on the HDSM nodes provide Checkpointing as in built mechanism for user process.

If the HDSM [50] node is based on PVM (Processor Virtual Machine) then checkpointing and restart procedure is a built in feature of the operating system. While implementing the DSM layer (consistency layer) the global or local check pointing mechanism and recovery process are provided as part of redundancy and failure recovery. If the HDSM supports (in turn operating system supports checkpointing) one can choose the different chekpointing techniques applicable and suitable pluggable modules can be enabled to become integral part of HDSM under consideration.

There are several different checkpointing implementations exists few are User Level Transparent Checkpointing, System Level or Kernel Level CheckPointing(SLC), Application-Level Checkpointing (ALC) [46], each technique has its own advantages and disadvantages. Usually only one of these techniques is supported by the HDSM node. Hence when user selects this option it is expected that user is aware of the compatibility issues and its related problems. Hence we presume that the user is aware of it and selects the checkpointing and restart option accordingly.

6.6.3 Process Migration

Process Migration is ability to move a currently executing process between different processors which are connected only by a network (that is, not using locally shared

memory). The originating machine's operating system collects the entire state of the process so that the destination machine may continue its execution on the other machine. In HDSM environment the hardware architectures of the machines usually differs, means the instruction set & the data representation in main memory is likely to be different. This also means that data conversion routines (provided as dynamic libraries) are mandatory and are provided as part of user selection process.

If the user application is process oriented then user can choose this pluggable module, which will lead to proper HDSM modules which are based on process migration framework. This module is complex and implementation specific [41] because of realization of process migration involves several issues are to be solved as part of process migration.

6.6.4 Data Migration

Whenever a runtime error occurs on the present processor (hardware error) or if the load on the current processor exceeds a threshold, DSM employs two techniques process migration and data migration. In HDSMs, the data migration may be a better option because of hardware challenges posed by the architectural difference on the receiver processor.

The data migration is also a novel concept and is more economical compared to process migration, as the same process (same process name) in other system can be used to perform action that supposed to be performed by the migrated process in case of process migration. The concept of data migration is that, if the particular code is available in the destination node then the data for that can be sent using message passing techniques for multicomputer and the results can be sent back. Some of the DSMs employ both data migration and process migration techniques. *If the DSM nodes are homogeneous then the data migration may be preferred choice.*

One important advantage of data migration is that it can improve the locality of accesses: after the first access by a thread to some data, successive accesses will be local. In addition, cache-coherent shared memory increases concurrency for read-shared data (shared data that is rarely written) by replicating it in different caches that can be accessed concurrently.

Data migration can be useful mechanism for parallel model of distributed computing wherein the process is duplicated on different machines running in parallel. MPI communication involves mainly data migration model of implementation. In general data migration can be applied to situations where global functions can be called with different data sets in object migration and also in cases of computation migration.

The only disadvantage of data migration is the cost of data to be transmitted to the remote machine. If the data to be transferred are within some allowable limit then one can dynamically choose to opt for data migration instead of process migration. If the implementation language is JAVA then the data migration can become part of object migration which involves both migrations. Data migration usually involves data conversion in heterogeneous systems as the architectures of both source and destination are different. The data conversion is explained in the next section.

6.6.4.1 Data Conversion

Data items may be represented differently on various types of hosts due to differences in the machine architectures, the programming languages for the applications, and their compilers. For data types as simple as integers, the order of the bytes can be different. For floating point numbers, the lengths of the mantissa and exponent fields, as well as their positions can differ. For higher level structured data types (e.g., records, arrays), the alignment and order of the components in the data structure can differ between hosts.

Sharing data among heterogeneous hosts means that the physical representation of the data will have to be converted when the data is transferred between hosts of different types. In the most general case, data conversion will not only incur run-time overhead, but also may be impossible due to nonequivalent data content (e.g., lost bits of precision in floating point numbers, and mismatch in their ranges of representation, i.e. differences in processor specific internal representation of floating point number). This may represent a potential limitation to HDSM for some systems and applications. To cater to this the HDSM layer should be equipped for invoking the conversion routines necessary to handle such intricacies. These are provided by the dynamic libraries used for data conversion.

As the architectures of different nodes differ, the data stored on one machine may be reported as Big Endian and other may be Little Endian which requires to be handled. Hence the compiler related issues the alignment of the word/byte/quad word and the data storage wrappers need to be developed in case of HDSM data conversion [43].

Yet another mechanism for data conversion is to convert the data into machine independent format and send the data to the other HDSM node and reconvert data to the machine dependent format (i.e. XDR format [118]).

Some aspects of data migration is taken care by the TCP/IP, socket communication protocols, which are of conversion of the data to the destination machine format which is partially solved by the communication layer of proposed HDSM.

6.6.5 Thread Migration

Thread migration can be used to improve load balance as well as locality. Migrating an entire thread can be expensive, since there may be a large amount of state to move. It also may be unnecessary, since we often need to migrate only a small amount of state in a thread to improve locality substantially. In addition, migrating every thread that accesses a datum to the destination's processor could put too much load on that processor.

This becomes an automatic pluggable module once HDSMs nodes are object oriented and the user application is written in JAVA. If object migration is required, probable destination for migration of the object is identified, utilizing the bottom layer of process migration or thread/object migration depending on the implementation mechanism of the HDSM under consideration.

Few of HDSMs are based on JAVA. Since Java by default supports threads, the threads can be used to perform requisite functions to be carried by the HDSM on behalf of user applications. JVM supports thread to be placed on different processors (HDSMs based on distributed JVMs, Multijav [52]), thread migration is a preferred way of the HDSMs based on Java.

Any DSM, which is based on Thread based implementation involve either/or object migration and thread migration techniques. The only problem with thread migration is

the overhead involved with the JVM, its references, and object instantiation and method invocations.

The proposed HDSM framework envisages the thread placement on different cores for uniform load distribution. Also the proposed Hierarchical Thread Pool Executor (chapter 3) is also based on thread instantiation on cores with minimum load.

6.7 Multi-Core Operations Layer

This layer has two sub layers Runtime system Layer and Interface Layer for the Multi-core. The later has following components:

Network Library, Thread Pool Executor, Pre-fetching Strategies, Garbage Collector and multi-Threaded library are modules that will be operational as per the requirement of a user application. These interfaces will be then become integral part of the HDSM on the node.

6.7.1 Run Time System (RTS)

The runtime system of typical HDSM resides on top of the low level modules like Networking module, Thread Pool Executor, Pre-fetching Strategies, Garbage Collector and multi-Threaded library. In addition to the creation, caching and transfer of objects between different nodes in the cluster using the DSM Protocol, the runtime system must also perform Garbage Collection.

The task of the runtime system is to give information needed for the creation of a global address space to the migration layer. The RTS provides global address space for the DSM. The global address to individual address translation identification of the addresses for the required objects/methods will be carried by the RTS. It provides implementation dependent Interfaces, so that the user process is executed at machine level. In other words, it enables the intermediate code to be finally transformed to machine instructions. RTS supports mechanisms like objects being sent from one system to other, necessary support for the process migration at higher layer for the user program ultimately to execute on different machines or systems.

The RTS would be different for different Heterogeneous systems (different plug-ins will be selected as part of DSM initialization). The implementation of heterogeneity will differ based on whether it is JAVA based/C++ based (machine dependent) and will provide the necessary runtime libraries required for C++.

RTS also should be able to cater to coarse grain or fine grain granularity for DSM implementation, namely the memory consistency models across the systems under operation. The other issues are how to update the other system's memory structures with the current state of the processes in the system under consideration.

RTS also provides the necessary support to handle the inter system communications (Network Library) by either using the standard UDP or TCP/IP protocol stack or other communication mechanisms (sockets), or the cluster software. The primary connectivity in any DSM is that the systems are networked and the RTS should interact with base communication available as a communication network.

The RTS should have mechanism to save and restore the state of the system in case of failures. The failure could be hardware failure of a subsystem, communication failure or process errors. The RTS data structures and the mechanism of recovery should be automatic and without any user intervention and should not affect the running of application software. The mechanism for failure detection and choosing the nearest system that should take care and the data integrity will be some of the critical issues need to be addressed as part of design and implementation of RTS.

Different services required for the RTS can be broadly divided into modules which are called from the RTS. These are Network Library, Multi-threaded library, Garbage Collector, Hierarchical Thread Pool Executor and Dynamic pre-fetching strategies. Each of these is covered in the sections 6.6.2 through 6.6.6.

These Services usually require the Operating System Services which is detailed as a separate layer subsequently.

6.7.2 Network Library

On top of the operating system the DSM's networking library is placed. The network library provides a stream–oriented interface for runtime system. This provides user a

higher level library calls than individual communication mechanisms used by the library modules. Standard modules such as MPI [107], TCP/IP and UDP and support for both upcall mechanism and Interrupt driven mechanism of receiving message are hidden from the higher RTS layer. The various other MPI interfaces that are used are MPI-2 an upgraded version of MPI, MPI-Java MPI interface with callable interfaces, MPICH used for the Grid architecture based DSM, JavaMPI with process migration (follows client server model).

6.7.3 Multi-Threaded Libraries

The Multi-Threaded library module provides a uniform environment for the thread to be executed. It provides a higher level interactive mechanism for the RTS to interact with thread based methods/functions. The different components of multi-threaded library are like prthreads, psthreads and task based threaded library calls (OpenMP [28],TBB [122]). Depending on the calling mechanism and the higher level (RTS) modules which interact with the multi-threaded library, the various compatible functions are provided, which will be called by the different modules of RTS.

6.7.4 Garbage Collector (GC)

In Java, garbage collection (GC) is a part of the language specification and all JAVA based HDSM thus has built in support for it. The popular version of garbage collector used is a blocking, parallel version of mark-and-sweep. DSM has to use a distributed garbage collection. The problem with distributed garbage collection is that a machine may be running several threads which may all be updating objects, adding and removing references. The Garbage Collector algorithms and implementation has to be uniform on all nodes of HDSMs under consideration.

When a GC is scheduled to run it will stop all threads on all machines. All incoming messages are queued and once all the machines stopped by means of a barrier, every node in parallel starts marking its root-set. The root-set contains global pointer variable's local thread stacks. If a pointer 'p' points to an address owned by *CPU proc* and its page is mapped and the region is valid then send the list of pointers to the designated cpu. Now unmarked objects can be deleted.

A GC phase is only triggered if a node's local memory is full (*malloc* fails for the local heap). When this occurs, execution of all threads on all machines is stopped. After all machines have acknowledged that they have indeed stopped (a barrier is reached), all machines simultaneously start the garbage collection process. We speed up the garbage collection process by checking the cache of globally cached memory so we can reduce the size of the deferred list.

6.7.5 Hierarchical Thread Pool Executor (HTPE)

In several DSMs, threads are integral part of DSM implementation (ex: object based DSM) and managing their life cycle is an important issue. The straight forward approach for building a thread based application is to create a new thread each time a request arrives and service the request in the new thread.

A big drawback is that the overhead of thread creation for each request is very memory consuming; a application that creates a new thread for each request spends more time and consumes more system resources creating and destroying threads, than it does processing actual user requests.

JAVA 1.5 onwards facilitates Concurrent Utilities (java.concurrent.Utility Package) the concurrent utilities provide facilities such as atomic variables, hash functions and more importantly *Thread Pool Executors*.

The basic structural elements that make up a Thread Pool are a collection of tasks or runnables that line up in a queue, and a collection of live threads that constantly verify the collection of runnables, execute the run method of each task. After this is finished, instead of finishing the life cycle, the thread returns to the pool, ready to pick up another task. Lightweight executable frameworks can be constructed in many ways, but all stem from the basic idea of using one thread to execute many unrelated tasks (here, in succession). These threads are known as worker threads, background threads, and as thread pools when more than one thread is used.

Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each

Thread Pool Executor also maintains some basic statistics, such as the number of completed tasks.

As part of the integrated framework for the HDSM we also propose a **Hierarchical Thread Pool executor (Chapter 3)** which is an integral part of DSM architecture, provides a Hierarchical thread pool for the HDSM system.

The run time system is also fine tuned to identify threads being created by the Thread Pool executor and will eventually make use of the global data structures for its final placement of the thread on particular core in a multi-core environment of Heterogeneous distributed Shared Memory.

6.7.6 Dynamic pre-fetching strategies

In DSMs the objects/data that are frequently referred are obtained as and when referred by the application program. These objects/data need to be accessed, updated and referenced datum to cater to the consistency of the data. It may lead to locating the data on the correct node and transfer this data to required module under consideration. This involves data to be brought from main memory of a particular HDSM node. If the data is not in main memory, it may lead to memory stall which may take several milliseconds. *If we are able to predict the most probable data item being referred next and keep the data in cache memory, we will be able to achieve considerable speed up in overall execution of the related modules used for accessing the required data pertaining to application module being executed by the DSM (dynamic pre-fetching).*

As part of the integrated framework for the HDSM, we also propose a novel **n-gram based Dynamic Pre-fetching strategies (chapter 4)** which is an am integral part of DSM architecture, provides an object pre-fetching for the HDSM system. This proposed prefetcher will be integral part of RTS on each HDSM node.

6.8 Operating System/Hardware Layer:

This layer provides the services offered by the operating system which are required for the RTS to be functional. Examples for system services in Linux are

fork(),Open(), read etc. Additional systems services include are free memory available, number of processes running on this system and memory allocation and reallocation etc. Functions required for converting data from one hardware system to other hardware system (Endian conversions) are carried by dynamic runtime libraries.

The proposed Heterogeneous DSM should be able to provide support to Multi-core architecture too. This aspect is the recent development, as each node of DSM comprises a multi-core processor. If we assume that the operating system is Linux, the Operating system's *Proc* data structure would be accessed by the RTS component.

The other multi-core related features include number of cores for the present processor (dual core quad core etc), number of CPUs per core, number of CPUs per core active and the present load on the system. These parameters are stored as data structures available for the run-time system. These structures are stored as min-max heaps and are accessed by run-time system for the DSM. Some of the information is obtained from the operating system and these data structures will be used by RTS in turn facilitate the migration layer of the HDSM.

6.9 Conclusion:

In this chapter we have proposed a unified integrated framework for Heterogeneous DSM. We have proposed component diagram for it and also shown how the open source plug-ins can be used. The proposed integrated framework is broadly divided in to five layers. These are Application Layer, Object Management Layer, Migration and Consistency Layer, Multi-core Operations Layer and finally Operating System/Hardware Layer. We have also proposed HTPE, Dynamic pre-fetching strategies and its parallelization as important components. These components can be integrated even to an existing DSM's.

Chapter 7. Conclusion and Future Work

7.1 Conclusion

In this thesis we have proposed a ***Hierarchical Thread Pool Executer (HTPE)*** that works on each node of the DSM. Whenever a new task arrives and if the worker thread of the system(TPE) it is initiated is free, a thread is assigned to it. In case no worker thread is available, instead of storing them into a blocking queue, we store them into non-blocking queue. The tasks are sorted on their priority of execution in non-blocking queue. It checks for availability of worker threads into other systems HTPE daemon, this is accomplished by maintaining a common min-heap structure that stores the availability of worker threads. Thread migration takes place to a node with minimum workload. The algorithm is tested with variety of mixed programs and the result shows clear performance improvement compared to standard TPE.

Pre-fetching is the process of bringing the object/page from the memory into the cache before it is accessed. This is an important activity especially when we are working on DSM. One of the popular dynamic pre-fetcher in the literature is ESODYP. The drawback with ESODY is that the construction, prediction phases are costly in terms of time. We can infer from the proposed n-gram model based pre-fetcher that it performs much better when the memory references are more and significantly better than compared to the ESODYP for applications involving dynamic pre-fetching. The n-gram model gives many fold better performance for applications that are memory intensive and/or computational bound, and better for all the varieties of problems. We have integrated the applications with the model where memory stride references are required. This model can be integrated with Distributed Shared Memory implementation where the program and data require frequent pre-fetching for performance improvement.

The proposed dynamic pre-fetcher works on each node of the DSM. In case the nodes are multicores then we can extend the proposed system to make use of this

architecture. So we have proposed a PRAM based parallel algorithm and reported its parallel complexity.

At the end we proposed a unified integrated framework for Heterogeneous DSM. We have proposed component diagram for it and also shown how the open source plug-ins can be used. The proposed integrated framework is broadly divided in to five layers. These are Application Layer, Object Management Layer, Migration and Consistency Layer, Multi-core Operations Layer and finally Operating System/Hardware Layer. We have also proposed HTPE, Dynamic pre-fetching strategies and its parallelization as important components. These components can be integrated even to an existing DSM's.

7.2 Future Work

The proposed methods in the thesis can be integrated with the existing DSM's to get the performance improvement.

Since ESOYP model was already integrated into JACKAL, an object based DSM, on similar line; the proposed n-gram model can be integrated with the JACKAL SDSM. We are in the process of testing the parallel algorithm's performance on several data sets with various applications, and the initial results are very encouraging but it is beyond the scope of the thesis.

The results obtained (dynamic pre-fetching) can be extended to the other domains of the computer science, i.e. in Operating System.

Bibliography

1. Weiwu Hu, Weisong Shi, Zhimin Tang, JIAJIA: A software DSM system based on a new cache coherence protocol. *High-Performance Computing and Networking, LNCS*, Volume: 1593, pp: 461-472, 1999.
2. Hiroshi Harada, Yutaka Ishikawa, Atsushi Hori, Hiroshi Tezuka, Shinji Sumimoto, toshiyuki Takahashi, Dynamic Home Node Reallocation on Software Distributed Shared Memory. *Fourth International conference on High performance computing in Asia Pacific Region 2000*, Volume: 1, pp: 158-161 May 2000.
3. Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, Mitsuhisa Sato, PM: An operating system coordinated high performance communication library. *High-Performance Computing and Networking, LNCS*, Volume: 1225, pp: 708-717, 1997.
4. Daniel J. S., Kourosh Gharachorloo, Design and performance of the Shasta distributed shared memory protocol. *ICS '97 Proceedings of the 11th international conference on Supercomputing*, ACM New York, NY, USA, pp: 245-252, 1997.
5. Daniel J. S., Kourosh Gharachorloo, Chandramohan A. Thekkath, Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. *ASPLOS VII Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ACM New York, NY, USA, pp: 174-185, 1996.
6. Daniel Lenoski, Kourosh Gharachorloo, James Laudon J., Anoop Gupta, Hennessy J., Mark Horowitz, Monica Lam, Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. *Compcon Spring '90. Intellect*, pp: 62-67, 1990.
7. Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, Willy Zwaenepoel, TreadMarks: shared memory computing on networks of workstations. *Journal Computer*, Volume: 29, Issue: 2, pp: 18-28, February 1996.
8. Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, Willy Zwaenepoel, Compiler and software distributed shared memory support for irregular applications. *PPOPP '97 Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM New York, NY, USA, Volume: 32, Issue: 7, pp: 48-56, July 1997.
9. Bennett J. K., J. B. Carter J. B., W. Zwaenepoel, Munin: distributed shared memory based on type-specific memory coherence. *PPOPP '90 Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, ACM New York, NY, USA, Volume: 25, Issue: 3, pp: 168-176, March 1990.
10. Kai Li, IVY: A Shared Virtual Memory System for Parallel Computing. *ICPP (2)*, pp: 94-101, 1988.
11. Ng M. C., Wong W. F., ORION: an dynamic home-based software distributed shared memory system. *Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, pp: 187, 2000.
12. Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, David A. Wood, Fine-grain access control for distributed

shared memory. *ASPLOS VI Proceedings of the sixth international conference on rchitectural support for programming languages and operating systems*, Volume: 28, Issue: 5, pp: 297-306 December, 1994.

13. Falsafi, Babak; Lebeck, Alvin R.; Reinhardt, Steven K.; Schoinas, Ioannis; Hill, Mark D.; Larus, James R.; Rogers, Anne; Wood, David A., Application-specific protocols for user-level shared memory. *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pp: 380-389, November 14-18, 1994.
14. Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, David A. Wood, Sirocco: cost-effective fine-grain distributed shared memory. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, pp: 40-49, October 1998.
15. Leonnidas kontothanassis,Galm Hunt,Robert Stets,Mivhal cierniak,Srinivasan Parthasarathy,Wagner Meira Jr.,Sandhya D,Michael Scott: VM-based shared memory on low-latency, remote-memory-access networks. *24th annual international symposium on Computer architecture (ISCA '97)*, Volume: 25, Issue: 2, pp: 157–169, May 1997.
16. Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis! Srinivasan Parthasarathy, and Michael Scott, Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-WriteNetwork. *Sixteenth ACM symposium on Operating systems principles*, Volume: 31, Issue: 5, pp: 170–183, December 1997.
17. Muralidharan Rangarajan , Liviu Iftode, Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance (2000). *In Proceedings of the 3rd Extreme Linux Workshop*, pp: 341-352, 2000.
18. R.veldema,R.A.F.Bhoedjang,H.E.Bal Jackal, A Compiler Based Implementation of Java for Clusters Of Workstations (2001). *In proceedings of Principles & practice of parallel programming PPOP*, 2001.
19. K.L.Jhonson,M.F.Kaashoek,D.A,Wallah, CRL: high-performance all-software distributed shared memory. *Fifteenth ACM symposium on Operating systems principles, POSP '95*, Volume: 29, Issue: 5, pp: 213-226, December 1995.
20. John B. Carter, Dilip Khandekar, Linus Kamb Distributed Shared Memory: Where We Are and Where We Should Be Headed. *Fifth Worskshop on Hot Topics in Operating Systems*, pp: 119-122, May 1995.
21. Todd Mowry , Charles Q. C. Chan , Adley K. W. Lo, Comparative evaluation of latency tolerance techniques for software distributed shared memory. *Fourth International Symposium on High-Performance Computer Architecture*, pp: 300-311, February 1998.
22. R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, C.J.H. Jacobs, H. E. Bal, Source Level Global Optimizations for Fine-Grain Distributed Shared Memory Systems. *ACM SIGPLAN symposium on Principles and practices of parallel programming ,PoPP '01*, Volume: 36, Issue: 7, pp: 83-92, 2001.
23. Anant Agarwal, David A. Kranz, Venkat Natarajan, Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, Volume: 6 , Issue: 9, pp: 943-962, September 1995.

24. Anant Agarwal, David Chaiken, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Gino Maa, Dan Nussbaum, Mike Parkin, Donald Yeung, The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor *In Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, 1991.
25. Rohit Chandra, Ding-Kai Chen, Robert Cox, Dror E. Maydan, Nenad Nedeljkovic, Jennifer M. Anderson, Data distribution support on distributed shared memory multiprocessors. *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, Volume: 32, Issue: 5, pp: 334-345, May 1997.
26. A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Zwaenepoel, Software versus hardware shared-memory implementation: a case study. *ISCA '94 Proceedings of the 21st annual international symposium on Computer architecture*, Volume: 22, Issue: 2, 106-117, April 1994.
27. John K. Bennett, John B. Carter, Willy Zwaenepoel, Adaptive software cache management for distributed shared memory architectures. *ISCA '90 Proceedings of the 17th annual international symposium on Computer Architecture*, Volume: 18, Issue: 3a, pp: 125-134, June 1990.
28. Ayon Basumallik, Seung-Jai Min, Rudolf Eigenmann, Towards OpenMP Execution on Software Distributed Shared Memory Systems. *ISHPC '02 Proceedings of the 4th International Symposium on High Performance Computing*, pp: 457-468, 2002.
29. Robert Bisiani, Mosur Ravishankar, PLUS: A distributed shared-memory system. *17th Annual International Symposium on Computer Architecture*, pp: 115-124, May 1990.
30. Chi-Keung Luk, Todd C. Mowry, Compiler-based prefetching for recursive data structures. *ASPLOS VII Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, Volume: 30, Issue: 5, pp: 222 - 233, December 1996.
31. Matteo Monchiero, Gianluca Palermo, Cristina Silvano, Oreste Villa, Exploration of distributed shared memory architectures for NoC-based multiprocessors. *Journal of Systems Architecture: the EUROMICRO Journal*, Volume: 53, Issue: 10, pp: 719-732, October 2007.
32. Sotiris Ioannidis, Sandhya Dwarkadas, Compiler and Run-Time Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems. *Languages, Compilers, and Run-Time Systems for Scalable Computers*, Volume: 1511, pp 107-122, 1998.
33. Henri E. Bal, Orca: a language for distributed programming. *ACM SIGPLAN*, Volume: 25, Issue: 5, pp: 17-24, May 1990.
34. Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, Wen-mei W. Hwu, An asymmetric distributed shared memory model for heterogeneous parallel systems. *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pp: 347-358, March 2010.
35. Bal, H.E. Kaashoek, M.F. Tanenbaum, A.S Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, Volume: 18, Issue: 3, pp: 190–205, March 1992.

36. Raoul Bhoedjang, Tim Ruhl, Rutger Hofman, Koen Langendoen, Henri Bal, Frans Kaashoek, Panda: a portable platform to support parallel programming languages. *Sedms'93 USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems,* Volume: 4, pp: 11-11, 1993.
37. Matchy J.M. Ma, Cho-Li Wang, Francis C.M. Lau, JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing*, Volume: 60, Issue: 10, pp: 1194–1222, October 2000.
38. Mark Nuttall, Brief survey of systems providing process or object migration facilities. *ACM SIGOPS Operating Systems,* Volume: 28 Issue: 4, pp: 64-80, October 1994.
39. Geoffroy Vallée, Christine Morin, Renaud Lottiaux, Jean-Yves Berthou, Ivan Dutka Malen, Process Migration Based on Gobelins Distributed Shared Memory. *CCGRID '02 Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp: 325, 2002.
40. Balkrishna Ramkumar, Volker Strumpen, Portable Checkpointing for Heterogeneous Archtitecturesm. *FTCS '97 Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97),* pp: 58, 1997.
41. Dejan S. Miloji, Fred Douglis, Yves Paindaveine, Richard Wheeler, Songnian Zhou, Eleanor Rieffel, Wolfgang Polak, Process migration. *ACM Computing Surveys*, Volume: 32, Issue: 3, pp: 241—299, September 2000.
42. Vadim Iosevich, Assaf Schuster, Comparison of sequential consistency with home-based lazy release consistency for software distributed shared memory, *18th annual international conference on Supercomputing,ICS'04*, pp: 306-315, 2004.
43. Songnian Zhou, Michael Stumm, Tim McInerney, Extending Distributed Shared Memory to Heterogeneous Environments, *10th International Conference onDistributed Computing Systems*, pp: 30-37, 1990 .
44. Michael J. Quinn, Parallel Computing: Theory and Practice. *McGraw-Hill Series in Computer Science*, September 1993.
45. Kuo-Feng Ssu, Fuchs, W.K., PREACHES-portable recovery and checkpointing in heterogeneous systems. *Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, pp: 38-47, 1998.
46. Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, Martin Schulz, Application-level checkpointing for shared memory programs. *Proceedings of the 11th international conference on Architectural*, pp: 235-247, 2004.
47. Jack J. Dongarra, G. A. Geist, Robert Manchek, V. S. Sunderam, Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, pp: 166-175, 1993.
48. Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada, Yutaka Ishikawa, PM2: High Performance Communication Middleware for Heterogeneous Network Environments. *International conference on Super Computing 00*, pp: 52-53, 2000.
49. Ming-chit Tam, Jonathan M. Smith, David J. Farber, A taxonomy-based comparison of several distributed shared memory systems. *ACM Operating Systems Review*, Volume: 24, pp: 40-67, 1990.

50. Wen-Yew Liang, Chun-Ta King ; Feipei Lai, Adsmith: an efficient object-based distributed shared memory system on PVM. *Second International Symposium on Parallel Architectures, Algorithms, and Networks*, pp: 173-179, 1996.
51. Ce-Kuen Shieh, An-Chow Lai, Jyh-Chang Ueng, Tyng-Yue Liang, Tzu-Chiang Chang, Su-Cheong Mac, Cohesion: an efficient distributed shared memory system supporting multiple memory consistency models. *PAS '95 Proceedings of the First Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, pp: 146-152, 1995.
52. X. Chen , V. H. Allan, MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines. *In Proceedings of the Conference on Parallel and Distributed Processing Techniques and Applications*, pp: 91-98, 1998.
53. R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. Run-time optimizations for a Java DSM implementation. *Concurrency and Computation: Practice and Experience*, Volume: 15, Issue: 3-5, pp: 299–316, 2003.
54. Weijian Fang; Cho-Li Wang; Wenzhang Zhu; Lau, F. C M, A novel adaptive home migration protocol in home-based DSM. *IEEE International Conference on Cluster Computing 2004* , pp: 215-224, September 2004.
55. Michael Factor, Assaf Schuster, Konstantin Shagin, JavaSplit: A Runtime for Execution of Monolithic Java Programs on Heterogeneous Collections of Commodity Workstations. *IEEE International Conference on Cluster*, pp: 110-117, 2003.
56. Wenzhang Zhu, Cho-Li Wang, Lau, F.C.M., JESSICA2 : a distributed Java Virtual Machine with transparent thread migration support. *IEEE International Conference on Cluster Computing*, pp: 381-388, 2002.
57. Andrew Erlichson, Neal Nuckolls, Greg Chesson, John Hennessy, SoftFLASH: analyzing the performance of clustered distributed virtual shared memory. *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, Volume: 30, Issue: 5, pp: 210-220, December 1996.
58. Byung-Hyun Yu, Zhiyi Huang, Stephen Cranefield, Martin Purvis, Homeless and home-based Lazy Release Consistency protocols on Distributed Shared Memory. *Proceedings of the 27th Australasian conference on Computer science*, Volume : 26, pp: 117-123, 2004.
59. Fredrik Dahlgren, Per Stenström, Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, Volume: 7, Issue: 4, pp: 385-398 April 1996.
60. MinSeong Kim, Andy Wellings, Using the executor framework to implement asynchronous event handling in the RTSJ. *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pp: 16-25, 2010.
61. Hong Zhu, Zhaolin Yin, Ying Ding, Java Annotated Concurrency Based on the Concurrent Package. *Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'06)*, pp: 38-43, December 2006.

62. Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, Jenifer C. Lai, Class-based n-gram models of natural language. *Journal of Computational Linguistics*, Volume: 18, Issue: 4, pp: 467-479, December 1992.
63. Dou Shen, Jian-Tao Sun, Qiang Yang, Zheng Chen, Text classification improved through multigram models. *Proceedings of the 15th ACM international conference on Information and knowledge management*, pp: 672–681, 2006.
64. Kuansan Wang, Christopher Thrasher, Evelyne Viegas, Xiaolong Li, Bo-june (Paul) Hsu, An overview of Microsoft web N-gram corpus and applications. *HLT-DEMO '10 Proceedings of the NAACL HLT 2010 Demonstration Session*, pp: 45-48, 2010.
65. Adam Pauls, Dan Klein, Faster and Smaller N-Gram Language Models. *HLT '11 Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Volume: 1, pp: 258-267, 2011.
66. Taro Watanabe, Hajime Tsukada, Hideki Isozaki, A succinct N-gram language model. *ACLShort '09 Proceedings of the ACL-IJCNLP 2009 Conference*, pp: 341-344, 2009.
67. Hatice U Osmanbeyoglu and Madhavi K Ganapathiraju, N-gram analysis of 970 microbial organisms reveals presence of biological language models. *BMC Bioinformatics*, Volume: 12, 2011.
68. Steffen Bickel, Peter Haider, Tobias Scheffer, Predicting Sentences using N-Gram Language Models. *HLT '05 Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pp: 193-200, 2005.
69. Fredrik Dahlgren, Michel Dubois, Per Stenström, Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, Volume: 6, Issue: 7, pp: 733-746, July 1995.
70. Daniel F. Zucker, Michael J. Flynn, Ruby B. Lee, A comparison of hardware prefetching techniques for multimedia benchmarks. *International Conference on Multimedia Computing And Systems*, pp: 236-244, 1995.
71. Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, Charles C. Weems, Guided region prefetching: a cooperative hardware/software approach. *ISCA '03 Proceedings of the 30th annual international symposium on Computer architecture*, pp: 388-398, 2003.
72. Jean-Loup Baer, Tien-Fu Chen, Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, Volume: 44, Issue: 5, pp: 609-623, May 1995.
73. David Callahan, Ken Kennedy, Allan Porterfield, Software prefetching. *ASPLOS IV Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pp: 40-52, 1991.
74. T.-F. Chen, J.-L. Baer, A performance study of software and hardware data prefetching schemes. *ISCA '94 Proceedings of the 21st annual international symposium on Computer architecture*, pp: 223-232, 1994.
75. I Ganusov, M Burtscher, Future execution: a hardware prefetching technique for chip multiprocessors. *14th International Conference on*

Parallel Architectures and Compilation Techniques PACT 2005, pp: 350-360, September 2005.

76. Todd C. Mowry, Monica S. Lam, Anoop Gupta, Design and evaluation of a compiler algorithm for prefetching. *ASPLOS V Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pp: 62-73, 1992.
77. Doug Joseph, Dirk Grunwald, Prefetching using Markov predictors. *ISCA '97 Proceedings of the 24th annual international symposium on Computer architecture*, pp: 252-263, 1997.
78. Skeppstedt, J., Dubois, Michel, Hybrid compiler/hardware prefetching for multiprocessors using low-overhead cache miss traps. *International Conference on Parallel Processing*, pp: 298-305, August 1997.
79. José González, Antonio González, Speculative execution via address prediction and data prefetching, *ICS '97 Proceedings of the 11th international conference on Supercomputing*, pp: 196-203, 1997.
80. John Tse, Alan Jay Smith, CPU Cache Prefetching: Timing Evaluation of Hardware Implementations. *IEEE Transactions on Computers*, Volume: 47, Issue: 5, May 1998.
81. A-R Adl-Tabatabai, R L Hudson, S Subramoney, Prefetch injection based on hardware monitoring and object metadata. *Conference on Programming Language Design and Implementation*, pp: 267-276, 2004.
82. Srinath, S., Mutlu, O., Hyesoon Kim, Patt, Y.N., Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. *IEEE 13th International Symposium on High Performance Computer Architecture*, pp: 63-74, February 2007.
83. Klaiber, A.C., Levy, H.M., An architecture for software-controlled data prefetching. *The 18th Annual International Symposium on Computer Architecture*, pp: 43-53, 1991.
84. Haiming Liu; Weiwu Hu, A comparison of two strategies of dynamic data prefetching in software DSM. *Proceedings 15th International Symposium Parallel and Distributed Processing*, pp: 6, April 2001.
85. Ricardo Bianchini, Rape1 Pinto, Claudio L. Amorim, Data Prefetching for Software DSMs. *In Proceedings of the 1998 International Conference on Supercomputing*, pp: 385-392, 1998.
86. Jean Christophe Beyler, ESODYP: An Entirely Software and Dynamic Data Prefetcher based on a Markov Model. *In Proc. 12th Workshop on Compilers for Parallel Computers*, 2006.
87. Fu, John W. C., Patel, J.H., Janssens, B.L., Stride Directed Prefetching In Scalar Processors. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992. MICRO 25, pp: 102-110, December 1992.
88. Dahlgren, F., Stenstrom, P., Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *First IEEE Symposium on High-Performance Computer Architecture*, pp: 68-77, 1995.
89. Mark Moir, Practical implementations of non-blocking synchronization primitives. *PODC '97 Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pp: 219-228, 1997.

90. Maurice Herlihy, Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, Volume: 13, Issue: 1, pp: 124-149, January 1991.
91. Alexey Gotsman, Byron Cook, Matthew Parkinson, Viktor Vafeiadis, Proving that non-blocking algorithms don't block. *POPL '09 Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp: 16-28, 2009.
92. Jason Maassen, Thilo Kielmann, Henri E. Bal, Efficient replicated method invocation in Java. *JAVA '00 Proceedings of the ACM 2000 conference on Java Grande*, pp: 88-96, 2000.
93. Kuansan Wang, Xiaolong Li, Efficacy of a constantly adaptive language modeling technique for web-scale applications. *IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP*, pp: 4733-4736, April 2009.
94. Samuel Kariin, Chris Burge, Dinucleotide relative abundance extremes: a genomic signature. *Trends in Genetics*, Volume: 11, Issue: 7, pp: 283-290, July 1995.
95. Allan Campbell, Jan Mrázek, Samuel Karlin, Genome signature comparisons among prokaryote, plasmid, and mitochondrial DNA. *Proc Natl Acad Sci U S A*, Volume: 96, Issue: 16, pp: 9184–9189, 1999.
96. Wooyoung Kim, Voss, M., Multicore Desktop Programming with Intel Threading Building Blocks. *Software, IEEE*, Volume: 28, Issue: 1, pp: 23-31, 2011.
97. Sato, H., Nanri, T., Shimasaki, M., Design and implementation of PVM-based portable distributed shared memory system on the workstation cluster environment. *International Conference on Parallel and Distributed Systems*, pp: 578-583, December 1997.
98. Kivanc Dincer, jmpi and a Performance Instrumentation Analysis and Visualization Tool for jmpi, *In First UK Workshop on Java for High Performance Network Computing, EUROPAR '98*, pp: 1998.
99. Adam Ferrari Ferrari, JPVM: Network Parallel Computing in Java. *In ACM 1998 Workshop on Java for High-Performance Network Computing*, 1997.
100. Protic, J., Tomasevic, M., Milutinovic, V., Distributed shared memory: concepts and systems. *Parallel & Distributed Technology: Systems & Applications, IEEE*, Volume: 4, Issue: 2, pp: 63-71, Summer 1996.
101. Jul, E., Steensgaard, B., Implementation of distributed objects in Emerald. *International Workshop on Object Orientation in Operating Systems*, pp: 130-132, Oct 1991.
102. Michael Klemm, Jean Christophe Beyler, Ronny T. Lampert, Esodyp+: Prefetching in the Jackal Software DSM. *Euro-Par 2007 Parallel Processing*, Volume: 4641, pp: 563-573, 2007.
103. Pradeep Kumar Nalla, Rajeev Wankar, Arun Agarwal, Design of Concurrent Utilities in Jackal: A Software DSM Implementation. *Distributed Computing and Networking*, Volume: 4904, pp: 176-181, 2008.
104. Java Specification Request 133, http:/today.java.net/pub/n/ JSR133PR
105. Robert D. Blumofe, Charles E. Leiserson, Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, Volume: 46, Issue: 5, pp: 720-748, September 1999.

106. Angkul Kongmunvattana, Santipong Tanchatchawal, Nian-Feng Tzeng, Coherence-Based Coordinated Checkpointing for Software Distributed SharedMemory Systems. *ICDCS '00 Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pp: 556, 2000.
107. Diana Hecht, Constantine Katsinis, Fault-tolerant Distributed-Shared-Memory on a Broadcast-based Interconnection Network. *Parallel and Distributed Processing*, Volume: 1800, pp: 1286-1290, 2000.
108. Ian Foster, Nicholas T. Karonis, A grid-enabled MPI: message passing in heterogeneous distributed computing systems. *Supercomputing '98 Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp: 1-11, 1998.
109. Hai Jiang; Chaudhary, V., MigThread: thread migration in DSM systems. *International Conference on Parallel Processing Workshops*, pp:581-588, 2002.
110. Weisong Shi, Heterogeneous Distributed Shared Memory on Wide Area Network. *IEEE TCCA Newsletter*, pp: 71–80, 2001.
111. Nicholas Carriero, David Gelernter, Linda in context. *Communications of the ACM*, Volume: 32, Issue: 4, pp: 444-458, April 1989.
112. Delp, G.S., Farber, D.J., Minnich, R.G., Smith, J.M., Tam, M.C., Memory as a network abstraction, *Network, IEEE*, Volume: 5, Issue: 4, pp: 34-41, July 1991.
113. James, D.V., The Scalable Coherent Interface: scaling to high-performance systems. *Digest of Papers Compcon Spring '94*, pp.64-71, February March 1994.
114. Erik Hagersten, Anders Landin, Seif Haridi, DDM: A Cache-Only Memory Architecture. *Journal of Computer*, Volume: 25, Issue: 9, pp: 44-54, 1992.
115. Stephen Lucci, lzidor Gertne, Ani1 Gupta, Uday Hegde, Reflective-Memory Multiprocessor. *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pp: 85-94, 1995
116. Maples, Creve, Wittie, L., MERLIN. A superglue for multicomputer systems. *Compcon Spring '90, Intellectual Leverage. Digest of Papers, Thirty-Fifth IEEE Computer Society International Conference*, pp: 73-81, 1990.
117. Iftode, L., Singh, J. P., Li, K., Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Theory of Computing Systems*, Volume: 31, Issue: 4, pp: 451-473, August 1998.
118. Sun Microsystems, Inc., External Data Representation Standard: Protocol Specification. *RFC1014 by the ARPA Network Information Center*.
119. Bill Nitzberg, Virginia Lo, Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Conferences on Computer*, pp: 52-60, 1991.
120. John Paul Walters, Hai Jiang, Vipin Chaudhary, An Adaptive Heterogeneous Software DSM. *International conference on parallel Processing Workshops (ICPPW'06)*, 2006.
121. Kang-Lyul Lee, Pham, H.N., Hee-seong Kim, Hee Yong Youn, Ohyoung Song, A Novel Predictive and Self-Adaptive Dynamic Thread Pool Management. *011 IEEE 9th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp: 93-98, May 2011.

122. Wooyoung Kim and Michael Voss, Multi-core Desktop Programming with Intel TBB, IEEE *Software* ,pp:23-31,January/February 2011.
123. Michael Quinn, Parallel Computing: Theory and practice, McGraw Hill Publications, 1994.

Design of n-Gram Based Dynamic Pre-fetching for DSM

Sitaramaiah Ramisetti[1], Rajeev Wankar[2], and C.R. Rao[2]

[1] CVR College of Engineering Hyderabad India
sitara_r@cvr.ac.in
[2] Department of Computer and Information Sciences University of Hyderabad, Hyderabad, India
rajeev.wankar@gmail.com, crrcs@uohyd.ernet.in

Abstract. Many earlier reported works have shown that data pre-fetching can be an efficient answer to the well-known memory stalls. If one can reduce these stalls, it leads to performance improvement in terms of overall execution time for a given application. In this paper we propose a new n-gram model for prediction, which is based on dynamic pre-fetcher, in which we compute conditional probabilities of the stride sequences of previous *n* steps. Here *n* is an integer which indicates data elements. The strides that are already pre-fetched are preserved so that we can ignore them if the same stride number is referenced by the program due to principle of locality of reference, with the fact that it is available in the memory, hence we need not pre-fetch it. The model also gives the best probable and least probable stride sequences, this information can further be used for dynamic prediction. Experimental results show that the proposed model is far efficient and presents user certain additional input about the behavior of the application. The model flushes once number of miss-predictions exceed pre-determined limit. One can improve the performance of the existing compiler based Software Distributed Shared Memory (SDSM) systems using this model.

Keywords: SDSM, n-gram, pre fetching, Markov Chain.

1 Introduction

Pre-fetching is the process of making the data pre-loaded in to main memory before it is being referenced by the program in execution. For pre-fetching objects/data, we need the prior knowledge of the addresses being referenced. The basic methodologies are hardware pre-fetching and software pre-fetching. For Hardware pre-fetching to take place we require additional hardware at the processor level. Software based pre-fetching is done either by the compiler, wherein the compiler introduces the code that is required for pre-fetching to take place, Jackal DSM [6] is one of the classic examples of DSM that uses it. Another example is loop unrolling where the same addresses are frequently referred hence the compiler will instrument the code required for this; a compiler is able to perform a static analysis of the source code to generate some instruction hints [1]. This form of the optimization is static in nature.

Y. Xiang et al. (Eds.): ICA3PP 2012, Part II, LNCS 7440, pp. 217–224, 2012.

Several approaches ranging from dynamic hardware to static software mechanisms have been studied in the past. A stand alone dynamic software data pre-fetching solution Entirely SOftware and DYnamic data Pre-fetcher (ESODYP) was proposed by Beyler et.al. [2]. ESODYP is based on a memory strides Markov model. It runs in two main phases: a short training phase where a graph coding sequences of occurring memory strides is constructed and an optimizing phase where predicted addresses are pre-fetched. Information in the graph is updated continuously by monitoring accessed strides.

Better runtime optimization (dynamic optimization) is possible only when the application program's behavior is known, or a trace run is performed to identify the addresses that are required for pre-fetching. Hence dynamic analysis and optimizations is an important research area. To obtain dynamic data (address of variables and pointers), we need a model which is able to predict the addresses that are likely to be referenced by the program at run time. This can be achieved by probability estimation of the addresses that are likely to be referenced by the program. We have taken ESODYP as a reference model for both implementation and comparison because the proposed model is based on Markov chain.

Markov chain based model keeps track of the addresses that are pre-fetched. The Markov model stores stride numbers instead of the physical addresses, referencing time will effectively reduced due to the fact that if the referenced stride is in memory we need not pre-fetch it, instead, we can increase reference count of stride (node) to make it more probable for future references. In an operating system analogy the stride typically represents a page of addresses of certain size. Considering memory strides instead of actual addresses can give the predictor a better chance to keep all the information retrievable by the program with the least memory necessary. We propose a new algorithm based on n-gram model for prediction. We also present experimental results for comparison of this model with ESODYP for the set of applications under consideration. In this paper section 2 explains n-Gram model, section 3 presents proposed algorithm, section 4 explains computation of Conditional Probability Transition Matrix (CPTM), section 5 presents results and discussion and finally in section 6 we conclude.

2 n-Gram Model

For any n-gram model the underlying concepts are better explained using words occurrence in a language as an example (usually referred as n-gram language model) [3].

In an n-gram language model, we treat two histories as equivalent if they end in the same n - 1 words, i.e., we assume that for $k \geq n$, $P_r(w_k|w_1^{k-1}) = P_r(w_k|w_{k-n+1}^{k-1})$, Where P_r represents probability of word $w's$ occurrence. For a vocabulary of size$V$, a 1-gram model has $(V-1)$ independent parameters, one for each word minus one for the constraint that all of the probabilities add up to 1. A 2-gram model has $V(V-1)$ independent parameters of the form $P_r(w_2|w_1)$ and $(V-1)$ of the form $P_r(w)$ for a total of V^2-1 independent parameters. In general, an n-gram model has V^n-1 independent parameters: $V^{n-1}-1$ (V - 1) of the form $Pr(w_1|w_1^{n-1})$, which we call the order-n parameters, plus the V^{n-1} parameters of an (n - 1) gram model.

We estimate the parameters of an n-gram model by examining a sample of text t_1^T which we call the training text, in a process called training. If C(w) is the number of times that the string w occurs in the string t_1^T, then for a 1-gram language model the maximum likelihood estimate for the parameter Pr (w) is C(w)/T.

To estimate the parameters of an n-gram model, we estimate the parameters of the (n - 1)-gram model that it contains and then choose the order-n parameters so as to maximize $P_r(t_n^T | t_1^{n-1})$. Thus, the order-n parameters are

$$P_r(\mathrm{w}_n | w_1^{n-1}) = \frac{C(w_1^{n-1} w_n)}{\sum_w C(w_1^{n-1} \mathrm{w})} \quad (1)$$

In the literature n-gram model is typically applied to the language model, we have given the same for readers to understand concept. Our technique makes use of the stride numbers for pre-fetching in place of words used in the language model.

3 Proposed Algorithm

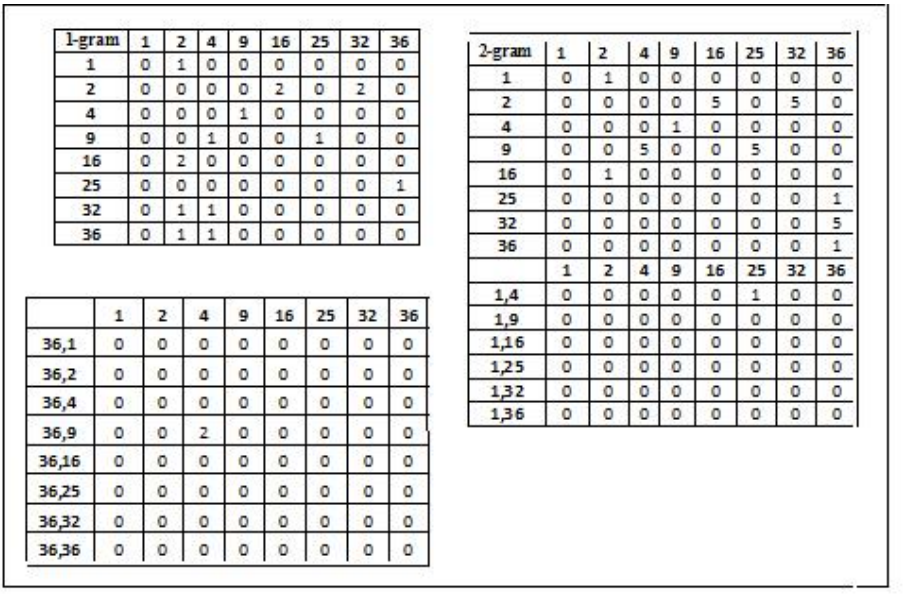
We observe the following problems with the above representation of the given application using the n-gram model.

1. For some strings where occurrences are sparse in nature that leads to (denominator of equation 1 being zero), we should not perform computations for these nodes.
2. The *n* in n-gram model is predefined where as in the present language model, *n* cannot be predefined.

Keeping the above two issues, an algorithm is proposed by considering sparse matrix representation with an incremental n.

Algorithm: Dynamic Pre-fetching
Input: Stride Numbers:
Output: Linked lists of each gram (with stride sequence), count and pointer to next node
Begin
Repeat
{
pNew = Create i^{th} gram node with *i-1* list size per each node;
Add *pNew* node to the list;
If pNew->*list* is already existing in one of the nodes then
        Update *pNew->count*
Compute the conditional probability;
Increment *i*;
}
Until conditional probability of each gram is 1;
end;

The input for nodes is sequence of strides numbers that are referenced by the application which are to be eventually pre-fetched for fulfilling the seamless execution of the applications. We have considered the random stride sequence of 1, 2, 16, 2, 32, 2, 16, 2, 32, 4, 9, 25, 36, 9, 4 as a reference stride sequence for building n-gram model. A typical matrix representation of 1-gram model and 2-gram model for the above sequence is given in fig. 1.



1-gram	1	2	4	9	16	25	32	36
1	0	1	0	0	0	0	0	0
2	0	0	0	0	2	0	2	0
4	0	0	0	1	0	0	0	0
9	0	0	1	0	0	1	0	0
16	0	2	0	0	0	0	0	0
25	0	0	0	0	0	0	0	1
32	0	1	1	0	0	0	0	0
36	0	1	1	0	0	0	0	0

2-gram	1	2	4	9	16	25	32	36
1	0	1	0	0	0	0	0	0
2	0	0	0	0	5	0	5	0
4	0	0	0	1	0	0	0	0
9	0	0	5	0	0	5	0	0
16	0	1	0	0	0	0	0	0
25	0	0	0	0	0	0	0	1
32	0	0	0	0	0	0	0	5
36	0	0	0	0	0	0	0	1
	1	2	4	9	16	25	32	36
1,4	0	0	0	0	0	1	0	0
1,9	0	0	0	0	0	0	0	0
1,16	0	0	0	0	0	0	0	0
1,25	0	0	0	0	0	0	0	0
1,32	0	0	0	0	0	0	0	0
1,36	0	0	0	0	0	0	0	0

	1	2	4	9	16	25	32	36
36,1	0	0	0	0	0	0	0	0
36,2	0	0	0	0	0	0	0	0
36,4	0	0	0	0	0	0	0	0
36,9	0	0	2	0	0	0	0	0
36,16	0	0	0	0	0	0	0	0
36,25	0	0	0	0	0	0	0	0
36,32	0	0	0	0	0	0	0	0
36,36	0	0	0	0	0	0	0	0

Fig. 1. 1-gram and 2-gram representation for the above stride sequence

In the given $n \times n$ matrix above referenced stride sequence 1, 2, 16, 2, 32, 2, 16, the item (1, 2) = 1 indicate that the sequence 1 followed by 2 occurs only once (count = 1) in the above given sequence. Similarly item (2, 16) = 2 represents 2, 16 sequence occurs twice in the above stride sequence. Similarly for 2-gram the node (36, 9, 4) = 2 indicate that node sequence of 36 followed by 9 as 2-gram followed by 4 is occurring twice in the given sequence. An application program (transportation problem) when executed on a system and its memory contains 32 blocks of 512 bytes (stride size 512), the sequence indicated represents memory address reference that program is accessing is *base address* + *strideno* * *stridesize*. First, strides 1 is accessed then stride 2 and then stride 16 so on.

Similar tables can be generated and stored for observing for 3-gram, 4-gram… n-gram. From the above tables we can observe that the tables to be represented in the memory require multi dimensional array representation and thus occupy large memory. Especially when the application program requires large memory or frequent data references then stride sequence will be long and the table sizes for given n-grams increases exponentially.

As part of n-gram implementation, we need to compute conditional probability for each and every entry in the table. This leads to exceptions like divide by zero when denominator for CPTM becomes zero. Hence the node representation which is proposed in our implementation will be ideal choice and will be optimal. Hence the proposed algorithm uses the linked list data structures for implementation for the n-gram.

Observing the sparse nature of the above stride occurrences following node structure is proposed:

Dynamic Stride List	Conditional Probability	PointerToNextNode
A0, A1 ,A2 , ……., An-1 Previous stride List	probability of An being the next stride	link

4 Computation and Importance of Conditional Probability Transition Matrix (CPTM)

For a given gram (1-gram…n-gram) the number of stride references are stored in a linked list. The frequency of occurrences of strides is represented in the form of a $n \times n$ matrix, where n represents stride number, indicating the sequence in which the strides are referred in the application. We use this matrix to compute Conditional Probability Transition Matrix (CPTM) using following procedure:

These lists are traversed for a given gram which is represented as a CPTM of size $n \times n$, where rows and columns represent the stride numbers and references to strides as a count. The element at (1,1) in 1-gram matrix represents stride-1 followed by stride-1 as a sequence, and how many times this stride sequence is referred by the application program.

We compute the sum of references of the strides for that row of strides and the conditional probability is defined as individual reference divided by the sum of the references for that row. We compute conditional probabilities of the different strides and sequences (based on gram number). As the stride is been accessed by the model, the address is mapped to stride number and the presence of the stride number in the list is checked by using a suitable hashing technique. If the stride is present then its corresponding count is incremented. We also estimate conditional probabilities starting from 1-gram continuing to n-grams by computing conditional probabilities and then summing these conditional probabilities to check for termination. The termination condition for n will be the sum of conditional probabilities for that n will be unity. We also can arrive at the optimal gram for the given series of strides, which can be a configurable parameter.

The model after computing the conditional probability will be able to find out best probable stride for a given sequence. The computed stride can be pre-fetched in advance and the list is updated once the stride is pre-fetched. If the stride is not in the list then it is a clear case of miss, now this new stride is pre-fetched. If the stride is present in the lists, then the corresponding list is updated to indicate that the new stride is in the memory.

The model gets updated incrementally when a new stride arrive. We can also make the model configurable for maximum number of misses, once this pre-defined limit reaches the model may be reset, so that the prediction can start a fresh and the memory data structures can be released and training can start a fresh(flushing).

Our method is efficient in the way that whenever the new address is computed the whole process of computing and updating lists and computing probabilities are done in one single and effective manner. The maximum number of strides to be stored in the memory depends on dynamic memory allocated for storing the list, which is again a configurable parameter for a given application.

5 Results and Discussions

We have considered three example programs for our study. We have taken the following examples for comparison of dynamic pre-fetching to evaluate the performance of two models (ESODYP and proposed) in terms of speed up of execution.

1. **Transport problem:** The program computes the efficient and economical mechanism of moving goods from sources to destinations. We integrated this application (the application program passes control to the model as a function call with the address as input, computation of stride number and maintenance of stride data structures and prediction is done by model) with the ESODYP model for monitoring the execution time, recorded the execution time.

   We have inserted the function call in C language [2] to call models (ESODYP and n-gram) to record the stride reference and hence updating of stride reference and probability of reference for prefetching. We have integrated the same transport application with the n-gram model and recorded the execution time. We have instrumented the necessary changes in the application program to pass control to the n-gram model and checked the execution time for this as well. We chose Linux OS, and we included the system calls Gettimeofday(starttime,0x0) to compute the execution time (with both models used for stride prediction).

   We chose 3 sources and 4 destination and cost of transportation and size of the material in tons is given as input. The execution time for the Transport problem with ESODYP is 52 μsecs. (micro-seconds). Similarly the execution time for the Transport Problem with n-gram model is 38 μsecs. Thus over all speedup of about 1.3682 is observed for this problem.

2. **N-Queens Problem:** The N queen's problem is to find how many ways the number of queens can be placed on a chess board. This is also a computational intensive application with large memory access. We have instrumented the necessary code for both ESODYP and n-gram models. We have considered a maximum of 16 queens to be placed on a chess-board. We have incremented the number of queens starting from 1 to 16 and

recorded the execution time with both ESODYP and n-gram models. The results of the study are given in Table 1 for comparison.

It is observed that the n-gram model is faster. For 3 queens ESODYP takes about 8 µsec, as compared to 2 µsec for n-gram, which is 4 times faster.

Table 1. Comparing ESODYP and proposed n-gram model

No.of Queens	ESODYP (time in µsecs.)	n-gram (time in µsecs.)
13	35793	34791
14	201689	197641
15	1237099	1215419

It can be observed that the execution time for n-gram is approximately same as with the ESODYP because as the linked list size increases and grams increase, the formation time for the linked list and subsequent search time increases for this application (the same can be observed with any DFS based application).

3. **Satisfiability Problem:** The satisfiability problem is to determine if a formula in propositional calculus, which is an expression that can be constructed using literals *and* and/or *or*, is true for some assignment of truth values to the variables. The problem checks for 30 equations to be compared for the satisfiability condition.

   The Execution times for satisfiability problem with ESODYP model was 4:45.89 minutes (4 minutes 45.89 seconds) and with n-gram model was 2:21.23 minutes. This shows a speed up of 2.02.

We can infer from the above three reference applications that the n-gram model performs much better when the memory references are more, and significantly better than the ESODYP for applications involving dynamic pre-fetching. The n-gram model gives better performance for applications that are memory intensive and/or computational bound. We have integrated the applications with the model where memory stride references are required. For all the applications which are memory intensive, the performance is much better. This model can be used for applications such as Distributed Shared Memory implementation where the program and data requires frequent pre-fetching for performance improvement.

6 Conclusion

The approach presented in this paper is faster than ESODYP as later uses forest of graphs to search for referenced stride, whereas n-gram model uses optimized linked lists and effective hashing for referencing. We can see from above three reference

applications that the n-gram model performs much better than ESODYP when the application require more memory references and involve dynamic pre-fetching. By extending similar analogy we can conclude that for the DSM, n-gram model will perform better than ESODYP, this is because of the fact that any DSM implementation requires large code and frequent memory references for both data and program code.

References

1. Beyls, K., D'Hollander, E.: Compile-time cache hint generation for epic architectures. In: Proceedings of the 2nd workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techniques (November 2002)
2. Beyler, J.C., Clauss, P.: ESODYP: An entirely software and dynamic data prefetcher based on a Markov model. In: Proceedings of the 12th Workshop on Compilers for Parallel Computers, A Coruna, Spain, pp. 118–132 (January 2006)
3. Brown, P.F., DeSouza, P.V., Mercer, R.L., Della Pietra, V.J., Lai, J.C.: Class-Based n-gram Models of Natural Language. Journal of Computational Linguistic Archive 18(4) (December 1992)
4. Veldema, R., Bhoedjang, R.A.F., Bal, H.E.: JACKAL, A compiler based Implementation of Java for cluster of workstations. In: Proceedings of SIGPLAN's Principles and Practices of Parallel Computing, PPoPP 2001 (2001)
5. Klemm, M., Beyler, J.C., Lampert, R.T., Philippsen, M., Clauss, P.: Esodyp+: Prefetching in the Jackal Software DSM. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 563–573. Springer, Heidelberg (2007)

2011 Second International Conference on Intelligent Systems, Modelling and Simulation

Design of Hierarchical Thread Pool Executor for DSM

Sitharamaiah Ramisetti
Department of Information Technology,
CVR College of Engineering
Hyderabad, India
sitara_r@cvr.ac.in

Rajeev Wankar
Department of Computer and Information Sciences
University of Hyderabad,
Hyderabad, India
rajeev.wankar@gmail.com

***Abstract*—This paper discusses the Design and Implementation of Hierarchical Thread Pool Executor (HTPE) using the non blocking queues. This work enhances the performances of the Distributed Shared Memory (DSM) Programming. Few Java based DSMs make use of the concurrent utilities of Java that support the thread pool executor with blocking queues (i.e. Jackal). The study of the new proposed algorithm is done in a simulated environment which demonstrates that if the non-blocking queues are used, their performance improve significantly. The Hierarchical Thread Pool Executor using the non blocking queue has been simulated on the cluster environment using MinHeap tree data structure.**

Keywords: Thread Pool Executor, DSM, Non-blocking queues, Concurrent Utilities.

I. INTRODUCTION

A normal Thread pooling saves the work of java virtual machine of creating brand new threads for every short-lived task. In addition, it minimizes overhead associated with getting a thread started and cleaning it up after it dies. By creating a pool of threads, a single thread from the pool can be recycled over and over for different tasks. With the thread pooling technique, one can reduce response time because a thread is already constructed & started, and is simply waiting for its next task.

All started threads go into a wait state (which uses very few processor resources) until a task is assigned to it. This fixed size characteristic holds the number of assigned tasks to an upper limit. If all the threads are currently assigned a task, the pool is empty. New service requests can simply be rejected or can be put into a wait state until one of the threads finishes its task and returns itself to the pool.

Java Thread Pool Executor: JAVA 5.0 [4] provides concurrent utilities as a new enhancement that includes thread pool executor which facilitates Executor and Worker pool. The executor provides both mechanism and facility to run a thread. The thread pool is a facility that could be thought as a group of threads in the reserve. All the available threads can be assumed to be placed in the pool and the executor takes these threads for execution when required.

Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. Worker threads exist separately from the Runnable and Callable tasks, and are often used to execute multiple tasks. Use of worker threads minimizes the overhead of thread creation. Thread objects use a significant amount of memory, and in a large-scale application allocating and deallocating many thread objects creates a significant memory management overhead specially in Distributed Shared Memory (DSM) implementations. In one of our earlier works we implemented ReentrantLock, Atomic variable and ThreadPoolExecutor in Jackal [1]. One of the major limitations of using the blocking queue is that the thread in a multithreaded environment is blocked until it is executed causing performance degradation. The proposed design of Hierarchical Thread Pool Executor using the non blocking queues overcomes this problem by sorting the non blocking queue based on the thread priority and ready state; obtain the dynamic load of the system and placing the thread on the minimum load node due to these inherent features it doesn't get blocked and gives better performance. In Section 2 we briefly explain the functionality of the normal Thread Pool Executor. In Section 3 we give brief description of non-blocking queues. Section 4 presents the design of the hierarchical Thread Pool Executor along with the algorithm.

II. THREAD POOL EXECUTOR

ThreadpoolExecutor is a service that executes each submitted task using one of possibly several pooled threads. Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Apart, each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks. One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is terminated (while it is still in use); it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads. An important advantage of the fixed thread pool is that applications using it degrade gracefully.

A simple way to create an executor that uses a fixed thread pool is to invoke the new FixedThreadPool factory method in java.util.concurrent.Executors. This class also provides the following factory methods:

- The newCachedThreadPool method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.

978-0-7695-4336-9/11 $26.00 © 2011 IEEE
DOI 10.1109/ISMS.2011.50

- The newSingleThreadExecutor method creates an executor that executes a single task at a time.
- Several factory methods are ScheduledExecutorService versions of the above executors.

2.1 Core and maximum pool sizes

A normal ThreadPoolExecutor on the other hand will automatically adjust the pool size according to the bounds set by corePoolSize and maximumPoolSize. When a new task is submitted in method execute, and less than corePoolSize threads are running, a new thread is created to handle the request, even if other worker threads are idle. If there are more than corePoolSize but less than maximumPoolSize threads running, a new thread will be created only if the queue is full. By setting corePoolSize and maximumPoolSize the same, one can create a fixed-size thread pool. By setting maximumPoolSize to an essentially unbounded value such as Integer. MAX_VALUE, we allow the pool to accommodate an arbitrary number of concurrent tasks.

2.2 Creating new threads

New threads are created using a ThreadFactory. If not otherwise specified, a defaultThreadFactory() is used, that creates threads to all be in the same ThreadGroup and with the same NORM_PRIORITY priority and non-daemon status. By supplying a different ThreadFactory, we can alter the thread's name, thread group, priority, daemon status, etc. If a ThreadFactory fails to create a thread when asked by returning null from newThread, the executor will continue, but might not be able to execute any tasks.

2.3 Queuing

Any BlockingQueue may be used to transfer and hold submitted tasks. The use of this queue interacts with pool sizing:

- If fewer than corePoolSize threads are running, the Executor always prefers adding a new thread rather than queuing.
- If corePoolSize or more threads are running, the Executor always prefers queuing a request rather than adding a new thread.
- If a request cannot be queued, a new thread is created unless this would exceed maximumPoolSize, in which case, the task will be rejected.

2.4 Direct handoffs, Unbounded queues, Bounded queue

Direct handoffs: A good default choice for a work queue is a SynchronousQueue that hands off tasks to threads without otherwise holding them. Here, an attempt to queue a task will fail if no threads are immediately available to run it, so a new thread will be constructed. This policy avoids lockups when handling sets of requests that might have internal dependencies.

Unbounded queues: Using an unbounded queue (for example a LinkedBlockingQueue without a predefined capacity) will cause new tasks to wait in the queue when all corePoolSize threads are busy. Thus, no more than corePoolSize threads will ever be created.

Bounded queue: A bounded queue (for example, an ArrayBlockingQueue) helps prevent resource exhaustion when used with finite maximumPoolSizes, but can be more difficult to tune and control. Queue sizes and maximum pool sizes may be traded off for each other: Using large queues and small pools minimizes CPU usage, OS resources, and context-switching overhead, but can lead to artificially low throughput.

2.5 A Generic Thread Pool

The class Thread pool is used to pool a set of threads for generic tasks. The worker threads are running inside ThreadPoolWorker objects, when a ThreadPool object is constructed, it constructs as many ThreadPoolWorker objects as are specified. To run a task, ThreadPool is passed a Runnable object through its execute() method. If a ThreadPoolWorker object is available, the execute() method removes it from the pool and hands off the Runnable to it for execution. If the pool is empty, the execute() method blocks until a worker becomes available. When the run() method of the Runnable task passed in returns, the ThreadPoolWorker has completed the task and puts itself back into the pool of available workers. There is no other signal that the task has been completed. If a signal is necessary, it should be coded in the task's run() method just before it returns.

2.6 Benefits of Thread Pooling

Thread pools address two different problems:

- They usually provide improved performance when executing large numbers of asynchronous tasks.
- Due to reduced per-task invocation overhead and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks.

Each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks.

2.7 Considerations of Thread Pooling

ThreadPool serves as the central point of control for managing the worker threads. It holds a list of all the workers created in workerList. The current pool of idle Thread Pool Worker objects is kept in a FIFO queue

III. NON-BLOCKING QUEUES

The traditional approach to multi-threaded programming is to use locks to synchronize access to shared resources. Synchronization primitives such as mutexes, semaphores, and critical sections are all mechanisms by which a programmer can ensure that certain sections of code do not execute concurrently, if doing so would corrupt shared memory structures. If one thread attempts to acquire a lock

that is already held by another thread, the thread will block until the lock is free.

Blocking a thread is undesirable for many reasons. An obvious reason is that while the thread is blocked, it cannot accomplish anything. If the blocked thread is performing a high-priority or real-time task, it is highly undesirable to halt its progress. Other problems are less obvious. Certain interactions between locks can lead to error conditions such as deadlock, livelock, and priority inversion. Using locks also involves a trade-off between coarse-grained locking, which can significantly reduce opportunities for parallelism, and fine-grained locking, which requires more careful design, increases overhead and is more prone to bugs.

Java 5.0 makes it possible for the first time to develop non-blocking algorithms in the Java language and the java.util.concurrent package uses this capability extensively. Non-Blocking algorithms are concurrent algorithms that derive their thread safety not from locks, but from low-level atomic hardware [5] primitives such as compare-and-swap. These algorithms can be extremely difficult to design and implement, but they can offer better throughput and greater resistance to problems such as deadlock and priority inversion.

IV. HIERARCHICAL THREAD POOL EXECUTOR

Hierarchical ThreadpoolExecutor is an extension of ThreadpoolExecutor which schedules different threads on cluster machines (viewed as a DSM) with all basic characteristics of the ThreadpoolExecutor.

The proposed implementation of Hierarchical Thread Pool Executor is definitely a self-contained component. However the motivation is derived from the Java Thread Pool Executor. This design scheme is intended to be a replacement for the usual Thread Pool Executor in a distributed environment with necessary support to execute threads with less effort and proper utilization of resources.

The implementation exploits the underlying system configuration in order to fetch information related to the thread and process requests. This information is utilized by the executor to appropriately prioritize the requests and run them concurrently with due consideration to atomicity and consistency by extensive use of Atomic Variables. The Non-Blocking queues are used to design and implement hierarchical thread pool executor.

4.1 Functional Requirements

Java's implementation of thread pools is based on an Executor and is provided in the form of the ThreadPoolExecutor class introduced in Java 1.5. The java.util.Concurrent API provides support to the implementation of thread pools. Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks. A simple way to create an executor that uses a fixed thread pool is to invoke the newFixedThreadPoolfactory method in java.util.concurrent.Executors.

Non- blocking algorithms guarantee that if there are one or more active processes trying to perform operations on a shared data structure, some operation will complete within a finite number of time steps. On asynchronous (especially multi-programmed) multiprocessor systems, blocking algorithms suffer significant performance degradation when a process is halted or delayed at an inopportune moment. Blocking algorithms allow a slow or delayed process to prevent faster processes from completing operations on the shared data structure indefinitely. Non-blocking algorithms guarantee that if there are one or more active processes trying to perform operations on a shared data structure, some operation will complete within a finite number of time steps.

The concept of a Non-Blocking Queue ensures that the threads competing for a shared resource do not have to go into blocked state or have their execution indefinitely postponed by mutual exclusion. A non-blocking algorithm is wait-free if there is also guaranteed per-thread progress. An algorithm is non-blocking if the suspension of one or more threads will not stop the potential progress of the remaining threads.

4.2 Design Aspects

The hierarchical thread pool executor is designed to actually increase the efficiency when it comes to the execution of threads in a DSM environment. The underlying system configuration is exploited to fetch the information regarding various application threads and allotting them to certain processors with fewer loads for their execution. This allocation is done in a hierarchical fashion by employing a binary tree to fetch the least load processor. The thread then executes on that processor and proceeds to completion.

The various incoming application threads are first collected in a Non- Blocking Queue data structure. These threads are then sorted according to their priority thus adhering to priority scheduling mechanism for threads. Now the non- blocking queue consists of application threads sorted according to priority from head to tail in decreasing order i.e. the highest priority thread is at the head position and the least priority thread at the tail position.

Execution of threads follows insertion, but for which the threads have to be first deleted from the queue to indicate that the thread has done with its execution. So we first check for the state of the thread, in case the thread is in the blocked state, then it is retained in the queue and not deleted. On the other hand, if it is ready to run, then the thread is deleted and sent for execution. By execution we mean invoking the run method of the concerned thread class. In order to include certain amount of dynamism we employ some randomization techniques to change the state of the thread regularly.

The hierarchical thread pool executor is actually intended to be deployed at the application layer on a shared memory environment with a shared address space. But since this is a simulation of the actual implementation, the design simply addresses a single memory space on a single

computer system. However with the use of Jackal [2, 3], a compiler based implementation of DSM, this architecture can be deployed to operate on a cluster of workstations, thus making to compatible to work on a shared address space.

4.3 Simulations

We have simulated the cluster environment using MinHeap tree which stores current system information at each at node of the cluster configuration.

The three step process of the placing the thread can be as follows:

1. Sort the non blocking queue based on the thread priority and ready state
2. Obtain the dynamic load
3. Place the thread on the minimum load node

In the first step a nonblocking queue which contains thread information is sorted on priory field. In the second step the system load is obtained dynamically using Linux system calls, whenever a thread is placed on a node. This information is then sent to the other nodes in the cluster to update their MinHeap tree. The simulation is done by allocating different loads to different nodes. The node with minimum load is obtained by the traversal algorithm on MinHeap of all nodes. In the final step, we place the thread on the system for execution. The simulation program of two n x n matrix multiplication is written in Java with and without HTPE on dual core machine. The results of the simulation are tabulated in Table I.

TABLE I.

Squre Matrix Size	*Time in Milliseconds with Single Thread*	*Time in Milliseconds with Multi Threads*
100	28	36
200	72	82
300	252	211
400	635	479
500	1277	1435
600	2709	1571
700	3794	2334
800	6808	3735
900	8630	5176
1000	13733	8023
1100	19869	11773
1200	22928	14625
1300	31935	20698
1400	46249	28111
1500	61902	35575
1600	66632	48683
1700	87082	57229
1800	108907	74130
1900	129044	88190
2000	175822	118702

The Figure 1 shows that when the value of the value of n in the matrix increases, the performance improves by approximately 67%. Algorithm 1 gives a high level description of the hierarchical thread pool executor.

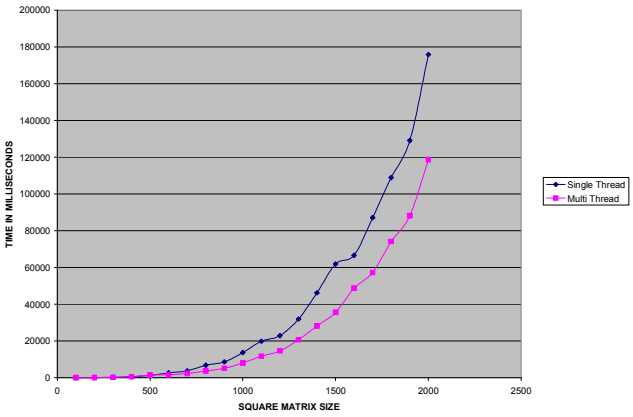


Figure 1. Comparing program execution with and wothout HTPE .

Algorithm 1. Hierarchical thread Pool Executor

```
Initialize
setmax_pool_size=max_threads;
max_number_entries_queue=max_que;
set thread_pool_state=init;

While thread_pool_sate NOT(Shut)
{
Sort(cur_node,que_start);
//Sort the double linked
//Non_blocking queon priority
//and ready_state and return the
//highest priority thread with
//ready_state;
//While updating the Non_bloking que
//use atomic variable (semaphore) to
//sort the queue based on priority;
 If cur_node_state=ready
{

Get_Current_system_load(bin_min_heap,
sys_no,load)
//place the thread on the system
//"sys";
  }
else
  {
   cur_node.state=Idle OR
cur_node.state=blocked
{
//display the thread pool has
//no_work_to_do
//
sleep 10 milli_seconds;
//hope new thread entries with ready
//have arrived in to Non_blocking_que
```

```
return
}
state=shut;
//display "We are exiting "
Exit();
//end of non_blocking queue
//The   following   logic   is   for
//Non_blocking queue;
//Application using HTP has to makes
//use   of   all   the   functions   of
//Non_blocking queue.

{
insert_thread()
      {
node.next_ptr=null   //create   a   new
//node;

set the thread_state=current_thread
//for simulation make thread_state as
//Idle insert the node at the end of
//the chain;
Check for the end of chain= null
then update tail.next to new_node;
and return=true;
If rem_node=node.next_ptr;
{
// now check whether this is pointing
//to null
rem_node.next_ptr=new_node;
return=true;
}

delete_thread()
start  with  head  NBQ  get  the  first
node's data;
for ;
:
If  node_state=blocked
 {
  cur_node=node.next;//skipNode=
temp.next.get();

else
{//node state is ready or idle}
if cur_node.nnext=null
{

return  //  exhausted  no  more  nodes
//whose status is ready

else
{
temp=cur_node.next
cur_node.next=temp.next=cur.node.next
nhNode.next=temp.next;

// pointer is updated hence node is
//deleted

display ready node is deleted;
}
return
//ie   thread   is   invoked   on   the
//designated system
delete_thread()
// call delete function of the non-
//blocking queue
```

V. CONCLUSION

In this work Hierarchical Thread Pool Executor using the non blocking queue has been designed and simulated on the cluster environment using MinHeap tree. One of the major limitations of using the blocking queue is that the thread in a multithreaded environment is blocked until it is executed causing performance degradation. The proposed design of Hierarchical Thread Pool Executor using the non blocking queue overcomes this problem and gives better performance. The work is in progress to implement the idea in one of the popular DSM like Jackal.

REFERENCES

[1] Pradeep Kumar Nalla, Rajeev Wankar, Arun Agarwal, "Design of Concurrent Utilities in Jackal: A Software DSM Implementation", ICDCN-2008, Lecture Notes in Computer Science 4904, Springer Berlin, pp. 176-181. (http://www.springerlink.com/content/t0073n823w011nt7/)

[2] Veldema, Ronald, Hofman, et al.: Source-Level Global Optimizations for Fine- Grain Distributed Shared Memory Systems. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, Snowbird, Utah/USA, pp. 83–92. (June 18-20, 2001)

[3] Veldema, R., Hofman, R.F.H., Bhoedjang, R.A.F., Bal, H.E.: Runtime Optimizations for a Java DSM Implementation. ACM Concurrency: Practice and Experience 15, 299–316 (2003)

[4] Java-1.5 concurrent utilities homepage, http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/packagesummary.html

[5] Atomic library by hp, http://www.hpl.hp.com/research/linux/atomic_ops