VOL. 11, NO. 3, FEBRUARY 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2016 Asian Research Publishing Network (ARPN). All rights reserved.

www.arpnjournals.com

A SCALABLE APPROACH FOR IMPROVING DYNAMIC MULTITHREADED APPLICATIONS ON NUMA BASED ARCHITECTURES

Praveen Kumar Reddy M.[1] and M. Rajasekhara Babu[2]
[1]School of Information Technology and Engineering, VIT University, Vellore, Tamil Nadu, India
[2]School of Computing Science and Engineering, VIT University, Vellore, Tamil Nadu, India
E-Mail: praveenkumarreddy@vit.ac.in

ABSTRACT

Scalability is a key concern for SMP based architecture in the current context. NUMA based architecture design seems to be a promising hope addressing the scalability. At the same time CC-NUMA based design architecture demands a deeper understanding and open vistas for key areas of improvement. With the approach of many core environments where memory is distributed among the different cores, it is challenging to design a thread scheduler along with proper data distribution across different nodes in a productive way. Our proposed research tries to investigate, evolve and analyze one of the key design issues for NUMA machine and proposes an innovative solution to address this key design issue under investigation in the current phase of our work. Our Algorithmic design proposed seems to be outperforming with respect to LibNUMA specifically.

Keywords: NUMA, architecture, SMP, CC-NUMA, UMA, LibNUMA, open MP.

1. INTRODUCTION

With the need for multiprocessing design, shared memory architecture provides processors to share a common memory. Uniform memory access (UMA) is one such architecture where the memory is uniformly accessed by all the processors. If the data size is increased, the processing speed need to be increased where in turn number of processors should be more. Since UMA shares a common bus, the allocation of bandwidth to the processors is a bottleneck which results in a scalability problem.

To overcome this problem, Non-Uniform Memory Access (NUMA) based design has been proposed. In NUMA, the memory is physically distributed among NUMA nodes where a common global address space is maintained [12].These nodes are connected with an interconnect. Memory access is fast and there is no issue of bandwidth. In UMA the latencies are uniform whereas in NUMA it differs with the distance of NUMA nodes. In NUMA each node is associated with a local memory [13]. If the memory accesses happen from this local node, then there is no issue of bandwidth. As there can be an access to global address space at some time, there is a need to access data from the remote nodes. This creates a latency difference between local and remote memory accesses and hence remote access should be minimized. Even though NUMA overcomes the scalability issue, some factors need to be optimized .Such key design factors include Data placement, Processor Affinity, Load Balancing, Cache Coherence, Thread Scheduling [14]. In this paper we plan to discuss different strategies to ensure optimal data placement in NUMA based system using automatic page migration and replication scheme. In NUMA, design for effective data placement is the major concern. Improper data locality leads to increase in remote access. In NUMA a processor can get to its nearby memory speedier than non-neighbourhood memory, that is, memory nearby to another processor or memory shared between processors. For an efficient Data placement proper memory management along with process scheduling has to be done.

Key requirements for design of optimal data placement NUMA engine require:

- Placer data shall have to be available locally to a NUMA node.
- Placer should avoid the overload transmission through interconnect.
- Remote access has to be minimized.
- If a particular node has insufficient memory, memory reclaim mechanism needs to be built in the Data placer.

In case of memory reclaim failure, there should be an alternate mechanism with placer to allocate neighbouring NUMA nodes (preferred nodes) memory.

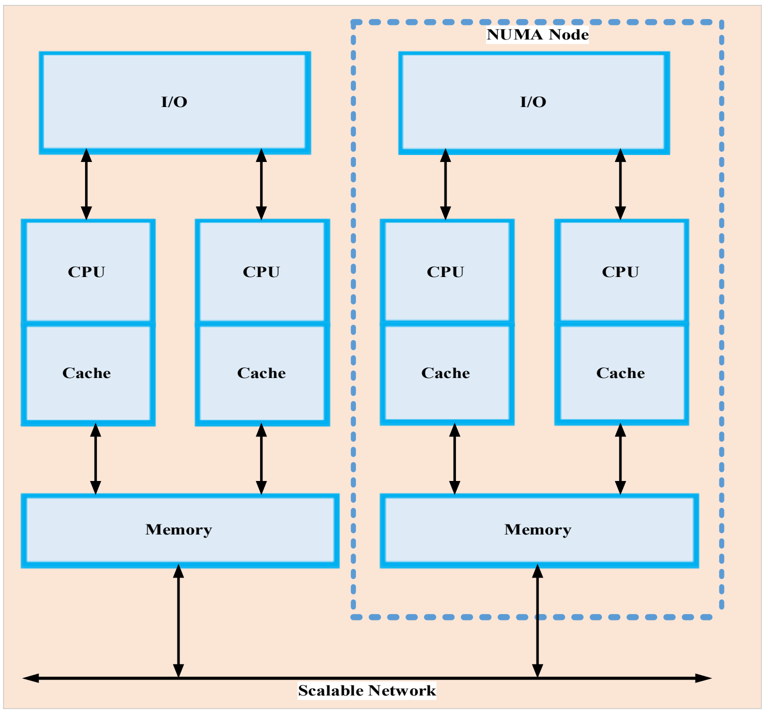

Figure-1. NUMA (Non Uniform Memory Architecture).

Memory affinity is one such data placement policy wherein we can achieve the above mentioned factors. Memory affinity involves data allocation, data migration, data replication which is efficiently being designed/proposed in this paper.

The rest of the paper is planned as follows. In section 2, we provided the various ways of providing memory affinity in NUMA architecture, section 3 gives a detailed explanation of various memory policies followed up with our algorithm in section 4. Section 5 summarizes our design along with discussion and section 6 provides the Conclusion with relevant future work in future.

2. LITERATURE REVIEW

To assure memory affinity in NUMA machines several works have been adopted. These works lead to some major solutions. In this section we briefly discuss various methodologies to assure memory affinity. The cons and pros of each method was identified and discussed.

A. Using LibNUMA

LibNUMA supports page migration, memory policies, and CPU bindings using kernel system calls through. These system calls allow programmer to allocate memory in run time. It majorly includes “numa_migrate_pages”, “numa_set_mpolicy ()”, “numa_move_pages ()”, “mbind ()” and “numa_get_mpolicy ()” [3]. The primary advantage of this kernel system call is that memory distribution can be controlled in a superior manner [4]. Usage of LibNUMA is a complex task since it involves bit masks, pointers, memory pages. Also to use this LibNUMA, one must manually enter the system calls (numactl) so that the application customs to the memory policies, which becomes a complicated work for the programmers. So there must be a mechanism by which the application must automatically adapt to the architecture. One such proposed mechanism is discussed in the later sections.

B. Open MP

Open MP is a language which supports efficient parallelism. The application must be coded using OpenMP primitives in order to get multithreading. To place the data in better way in NUMA machines, certain extensions were added to the OpenMP and it requires an explicit support from compilers. But all compilers don’t give the provision for OpenMP and directives are applied in a static manner [9].In [5] a mechanism to guarantee memory affinity on NUMA machine using OpenMP was presented by the author. The main idea in the paper was to make relation between threads and data .Also the work provides some suggestions of how to extend OpenMP in NUMA machines. Their results proved that OpenMP can perform well on tightly-coupled NUMA machines. Their work was not extended to automatic data placement. Our proposed work tries to provide this mechanism using automatic memory affinity in run time.

In [6] an efficient memory allocation in OpenMP using certain set of OpenMP directives is presented. These directives guide developers to allocate data efficiently on the NUMA architecture. All these directives are limited to the FORTRAN language. Using these directives efficient data distribution and page locality can be achieved [7] [8]. But these directives have to be included for an application in an explicit manner which requires hardware specifications in prior. There must be a better approach independent of prior knowledge on hardware which must provide efficient data placement.

In [16] author presented on how to improve the programming of openMP applications with heavy memory access requirements the main idea in the paper was programming with openMP related with local vs remote accesses. Integrating the openMP and MPI, to have as many threads as cores in the numanode, building an index for MPI and openMP, pays an attention to local accesses. In order to minimize the MPI message rate and minimal sensitivity to the latency, grouping or minimization of MPI processes is required. The runtime setup is crucial for best possible abuse of NUMA with open MP. The numerous cores depend on a multicluster configuration, where every cluster highlights a few basic centres sharing Level1 scratchpad memory. Intercluster correspondence is liable to nonuniform memory access (NUMA) impacts [17]. The programming model comprises of a broadened OpenMP, where extra directives permit to efficiently program the accelerator from a single host program, rather than writing separate host and accelerator programs, and convey the workload among groups in a NUMA-mindful way, accordingly enhancing the performance.

C. Compiler support

Memory affinity in a parallel application can be provided using OpenMP, HPF [10]. However achieving the memory affinity during run time is a complex task. This involves an effective cooperation from compilers and run time systems [11] so that the threads and data migrate dynamically according the behavior of the application. These special abilities have been included in the compiler. Now a day's GCC, Intel C compiler are capable to distribute threads and at the same time achieve memory affinity in NUMA machine. However to accomplish memory policies with OpenMP, the developer should have a prior knowledge of operating system, the topology that machine uses and the application [1, 2]. In [18] author presented compiler-based procedure to progress page placement in NUMA machines, there approach has speed up the software operation by four times without programmer's mediation. There are some limitations in this approach it cannot knob openMP directives, since they are having a specific semantics that was not yet encoded in there system.

3. BASIC WORK

Kernel's Memory affinity policies

Memory affinity is ensuring that processing unit always has their data close to them. So, to achieve utmost performance on NUMA machine the number of remote access during the execution has to minimize. Processor and data need to schedule so as the distance between them are to be closed. To minimize the distance and accordingly increase the performance of NUMA architecture we need some mechanism and tool in order to solve memory allocation, replication and migration to ensure memory affinity in NUMA system. However, none of these solutions provide portability as memory affinity control. To achieve this, memory affinity is ensured by applying a memory policy for an entire process. First touch algorithm was proposed in Linux 2.6.24 kernel which was the default policy to manage memory affinity. It places the page with respect to thread on the NUMA node that access it first. By this policy data is allocated by thread or master thread to its local memory which reduces remote accesses. However first touch policy apply on application that access symmetric data (regular). If thread does not access similar data, it leads to high number of remote access.

To reduce remote access the page should be migrated between NUMA nodes to make sure that data is closer to the processor that access it. Before page migration the following sequence of step has to be followed

- The victim page which is to be migrated should be removed from the Least Recently Used lists.
- All the references of PTE (page table entries) to the old page are freed and the page is put into sleep till the migration of the page is over.
- If suppose any kernel references are made to the page, then the page migration is aborted and should be further processed after the kernel usage is over.
- New kernel references are to be made for the new page after migration.

Next touch policy was introduced to provide automatic page migration. This policy is applicable for regular and irregular data. Next touch policy provide automatic dynamic page migration using "copy on write "(COW). Using COW page migration occurs only it is essentially needed. Generally page table entry (PTE) contain protection bit (write access and read access bit) to check the page detail. COW is implemented in LINUX to modify the write access bit of the page from the page table entry which indirectly generates a page fault on a write access. The Figure-2 explains the flow of process occurred in the next touch policy. Initially by using first touch policy, thread associates/binds page with respective NUMA node. In next touch policy the page is marked by flag bit and indicates that it will be used in near future. In order to mark the page using flag bit in LINUX it is accomplished using madvise () system call.

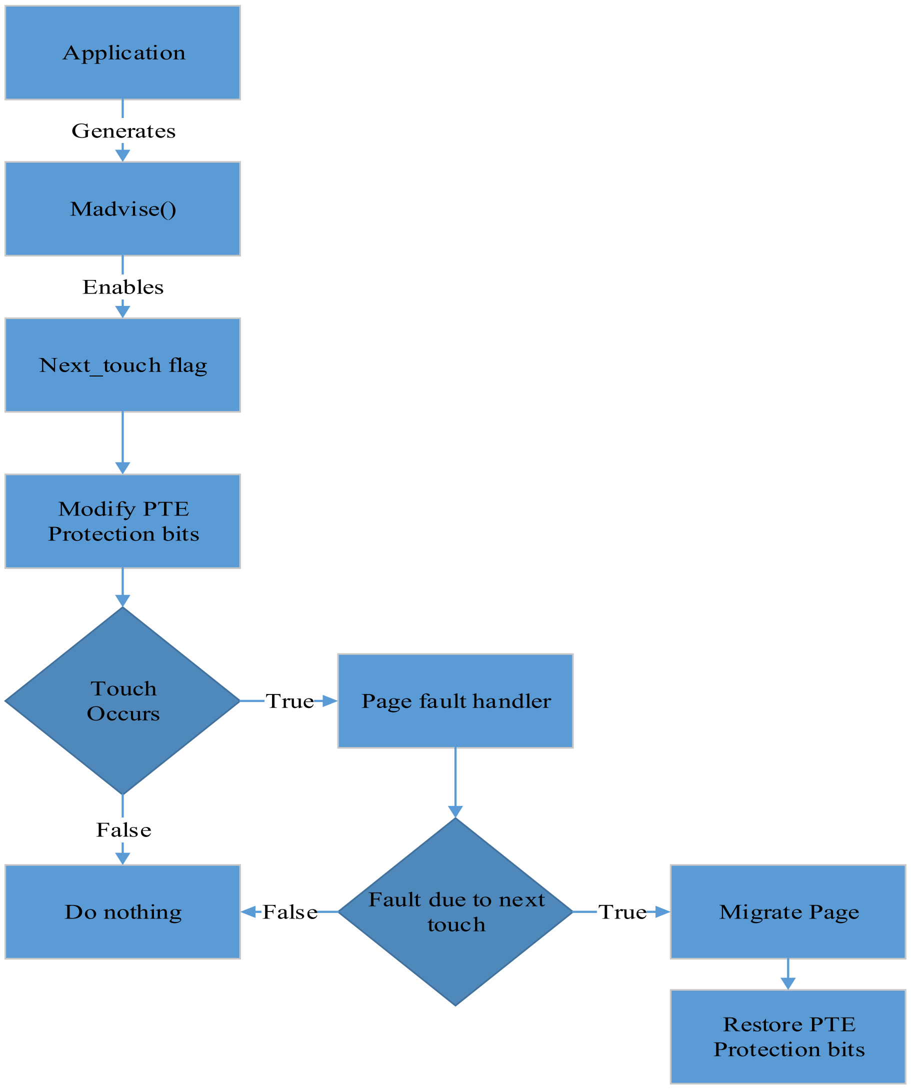

Figure-2. Next touch policy implementation.

Whenever we want to set the next touch flag, we have to modify the PTE protection bit. Due to the load balancing there is need to migrate the threads from one NUMA node to another irrespective of the data location of the thread. This leads to page fault whenever touch occurs for the page. When this happen page fault handler take care of the faults by checking the next touch flag of the page. It also sees that actual page fault is occurred because of Next Touch policy or a real page Fault. If the fault is due to the Next touch, then page will be migrated. After the page migration, the Next touch flag is removed from the corresponding buffer and enables its original protection bits in the PTE. Same procedure happens if Next touch faults happens. In general some pages are protected. This can be done by using mprotect () system call which prevents application to access that page. If that page is to be accessed, it leads to segmentation fault which is handled by custom signal handler [4]. This custom handler removes the protection temporarily, then migrates the resultant buffers and restores its protection back. This mechanism is not flexible to implement as we are calling the mprotect () (system call) twice to handle the segmentation fault, resulting in changing the TLBs frequently.The next touch policy produces a better performance when compared with numa_migrate_ pages()(libNUMA's API) because numa_migrate_pages() Shifts entire process address space on to new NUMA node which is not necessary as discussed in the previous sections.

The proposed design addressed all the above mentioned policies along with replication in next touch policy.

4. DESIGN AND ARCHITECTURE OF THE PROPOSED SYSTEM

Our proposed system is trying to provide a replication mechanism for the aforementioned next touch policy with some threshold constraint. Our algorithm tries to give an effective mechanism for Memory affinity which targets in reducing the remote accesses. The proposed mechanism also tries to incorporate all the LibNUMA support (migration, replication, preferred and interleaved) automatically.

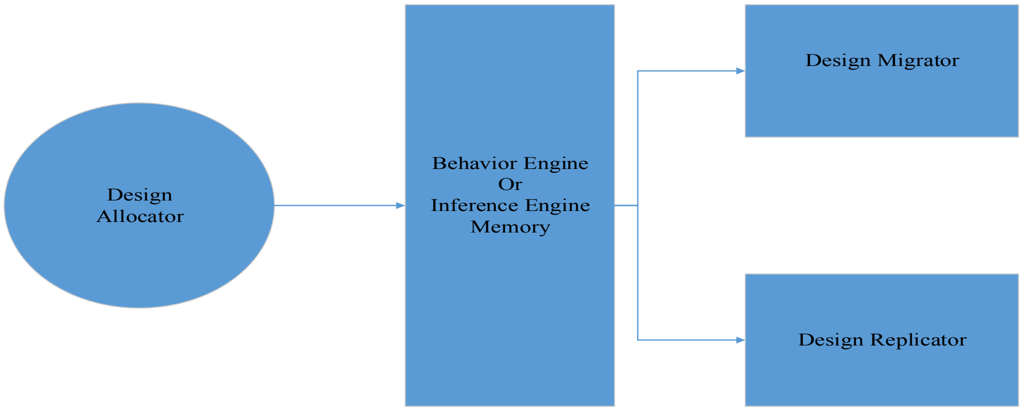

Figure-3. Design and architecture of the proposed system.

Module 1: Design allocator

Design allocator allocates memory to a node by default policy (First touch). According to first touch policy; memory will be allocated to the thread that first touches it. It sets the behavior of the shared page. Then it sets the next touch flag indicating that the page will be accessed in the future. Before doing his we must check whether the memory is available for migrating or replicating the pages to the destination nodes or not. If page fault value is less than the threshold value and memory is available, migrate the page. In case if the memory is not available then memory reclaim has to be done by using madvise_free ()

Module 2: Behavior engine

Behavior engine decides whether page candidate is fit to Migrate or Replicate based on the threshold value. The threshold value is to be checked with page fault counter. It also provides a access mechanism for pages (e.g. More write access implicitly disables the page from replication)

Module 3: Design migrator or replicator

This forms the core enabler for Migration or Replicator. Before the migration or replication happens again we have to check for sufficient memory availability. If the sufficient memory is not present memory reclaim (madvise_free ()) has to be happened. If the page fault count is greater than threshold frequency data replication has to be done. Page fault counter should be incremented only on the next touch. Whereas if the page faults count is less than threshold frequency data migration has to be done. Figure-4 depicts the overall proposed methodology flow discussed above.

Table-1. Key terminologies for a proposed system.

Key parameters	Semantics
Threshold value	Number of faults for a page to decide whether to replicate or migrate.
Reset interval(RST)	Reset the counter after certain number of clock cycles.
Write frequency threshold	Maximum number of write access allowed to a page after which a page is not considered for the replication.
Page fault counter	Number of page faults for a page occurs across the NUMA node with in reset interval.

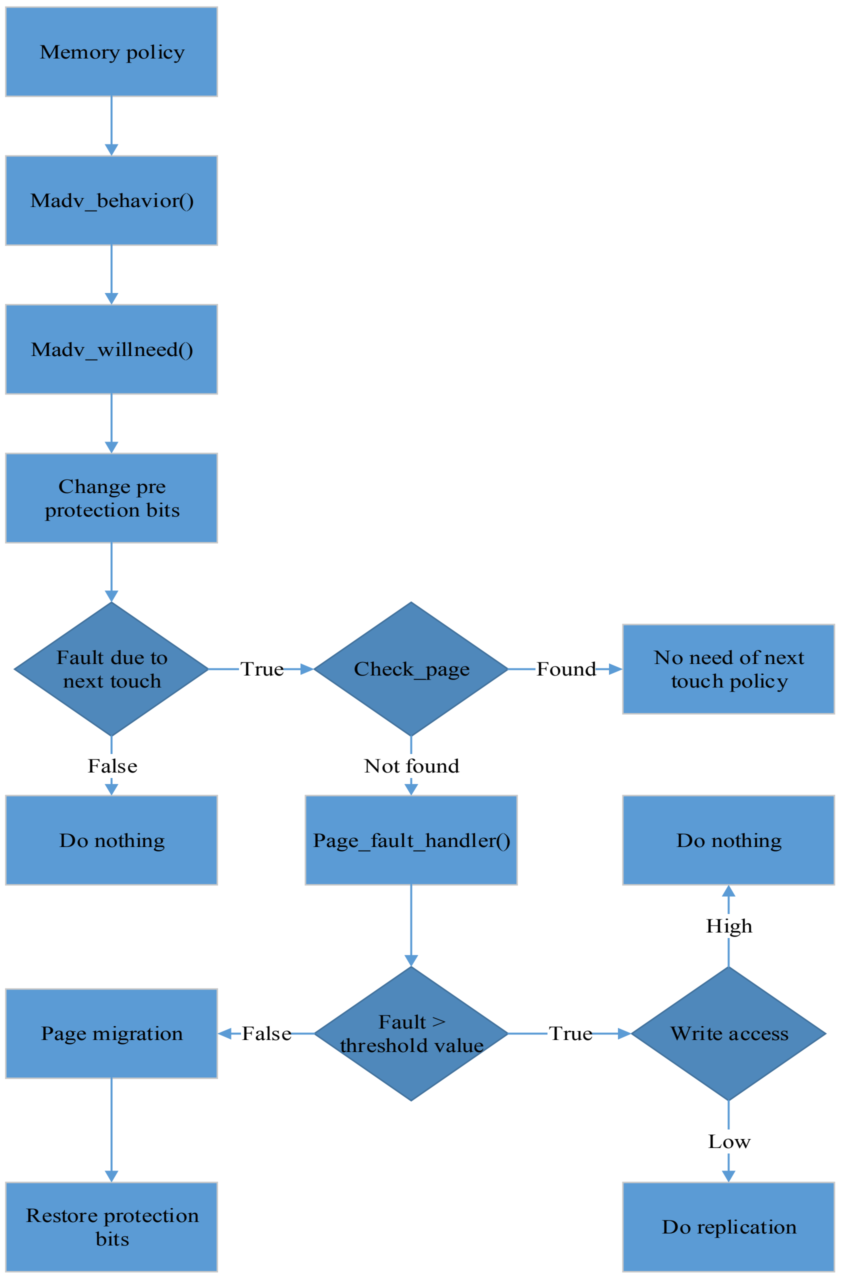

Figure-4. Next touch policy with replication.

To include replication mechanism we introduced some variables as threshold, reset (RST), and write frequency threshold. The threshold frequency checks the no of access of the page for a certain period across then NUMA nodes. The reset flag resets the page fault counter value after a particular clock cycle time (as of Table-1). The write frequency threshold finds how many write access happened to a page. If it is high, then check the write frequency threshold, if the number of write access is greater than write frequency threshold, do nothing. Else check for the memory again and replicate the page.

Algorithm

Init_configuration_cum_allocator_NUMA ( )

Step 1: Allocate a page in NUMA node as follows:

Step1.1.a Choose default memory policy ( )

Step 1.1.b Initialize the Memory Access parameters (like page_table_entry, protection bits, pid, bitmask, write access bit, vma, main memory etc…)

Step1.2 set the behavior of the shared space using madv_behavior ()

Step1.3 set the next touch flag using madv_willneed ()

```
Step1.4Change the page table entry protection bits i.e. pte_modify (page table entry, get_protection (0)).
Step1.5 Initialize threshold and reset value
Shared behavior process
Step 2 Check the recent access to Page i.e.
If (touch==true) do the following steps
{
/*checking whether the page fault occurred from the next touch or not*/
2.1. Check the pages for its local or remote access using Page_found=check_page(pid,bitmask,pageaddress)
2.2. If page is being from local node i.e. if (Page_found==1)
Do nothing and abort ()
/* next touch policy not needed*/
Else
/* page fault handler mechanism and increment the page fault count */
 2.2. Generate a page fault using
intpage_fault_count=page_fault_handler (addr, writeacess, pagetableentry, vma, mm)
2.2.1Increment the page fault using page_fault_count++;
 /*until reset values doesn't cross the periodical time value, if crosses reset the page fault values*/
Step 3Replication or Migration
3.1. Check the Page faults limit& compare the same with threshold limit set using.
 If (page_fault_count>= threshold value)
{
3.2. Replicate the page using page_replicate (node mask_all, vma, page table entry, Pd, mm)
}
Else
/* (if page_fault_count< threshold value)
{
3.1 Migrate the page using next touch policy as page_migrate (mm, pte, ptl, mm, vma)
/*provide migration*/
```

Algorithm for main design block design for data placer ()

```
Automatic_data_placer_for_NUMA ()
{
Step a) Configure the NUMA nodes & allocate the pages;
Init_configuration_cum_allocator_NUMA ( );/* step 1 above*/
Step b) Organize the NUMA shared process behavior;
Perform step2 above;
Step c) Based upon the threshold frequency    value
Either Migrate or Replicate;
 /* step 3* above */
}
```

The algorithm describes the next touch policy with replication. Initially every application obeys the default memory policy as first touch. Madvise () system call gives a prior knowledge to kernel of how an application expects to use the memory. The following table describes the different functionalities of madvise () system call.

By using madv_willneed () system call for which we change the page table entry bits (read/write access flag), indicates that page will be used in the future. Now whenever fault occurs, kernel confirms that this fault is by next touch policy and decides to take decision of migration or replication.

Table-2. Different functionalities of madvise ().

System call	Description
madvise_normal	Default kernel way of accessing the addresses.
madvise_sequential	Informs kernel that application will access a listed range addresses in a successive way.
madvise_random	Tells kernel that page references are in a random manner.
madvise_willneed	The specified address range will be referenced in future.
madvise_dontneed	The specified address range will not be referenced in future.
madvise_free	Intimating kernel that the specified range of addresses are no longer important(these addresses are freed when the memory pressure is high)

5. COMPARATIVE ANALYSIS AND DISCUSSIONS

In this proposed design we have investigated and addressed the key concerns or limitations imposed by the existing works. Our Algorithmic outline proposed is by all accounts beating as for LibNUMA particularly.

Consider an application which is generating 200 threads, where these threads are randomly distributed in parallel among the NUMA nodes. In LibNUMA using mbind () we can allocate memory to any node. Cpubind () allows process to execute on a set of CPUs for a specified node. LibNUMA also provides page migration using numa_migrate_pages () where a large buffers are to be moved which leads to increase in latencies (unnecessary pages are also transferred) [15].For the LibNUMA support the programmer should alter the application code manually which is a complex task [9]. By first touch policy adopted by us, the thread which touches the page first will be allocated to the corresponding node to make the access local. If other threads from different nodes need to access the same page (remote) very frequently in a short period, then migration becomes a tedious job where in TLB values are to be flushed frequently. So instead of migration we replicate the page based on the page fault count (if page fault is greater than the threshold count) in our proposed system addressing the key concern. Replication may also cause memory congestion. Duplicate copies indeed result in redundancy of data. So in order to make the memory available we use madvise (madv_free) resets in a periodical interval and the replicated page is removed based on the necessity. For every migration or replication the status of the page is checked in advance. Hence we ensured that our proposed system provides a better alternative to the existing works with dynamic processing built-in.

NUMA architectures has multi cores for each node. The thread scheduler is required to bunch on the similar node threads which are getting to the same information, for occurrence inside of an OPENMP parallel segment. We in this manner hope to have simultaneous accesses from various threads for a buffer to be moved.

Figure-5 shows the movement throughput on our experimentation stage when a few threads are certain to NUMA 1st node and moving memory from 0th node. It first demonstrates that parallelizing the movement does not bring any change for buffers littler than 1 MB. We feel that this is identified with lock dispute in the kernel in both executions. The figure additionally demonstrates that both methods accomplish somewhere around 55 and 65% change when moving huge buffers with 4 threads (one for each core). Lazy migration looks to scale somewhat better since despite everything it enhances a bit (+7 %) when including a fourth thread, accomplishing up to 1.5 GB/s. This throughput stays much lower than a normal memory duplicates between NUMA nodes, yet it must be noticed that a page-deficiency and serious page-table securing are included every hidden page relocation. Regardless of the possibility that the general throughput is restricted, strung movement still shows up as an intriguing arrangement when various strings taking a shot at the same dataset are relocated to another node in the meantime. Our kernel base implementation seems 33% speedier than the user-space model.

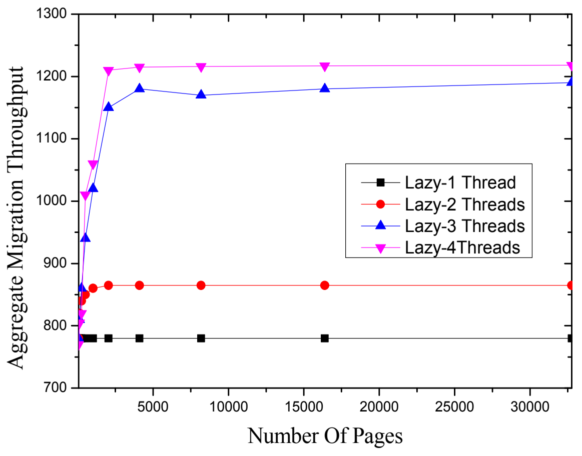

Figure-5. Throughput of Lazy threads.

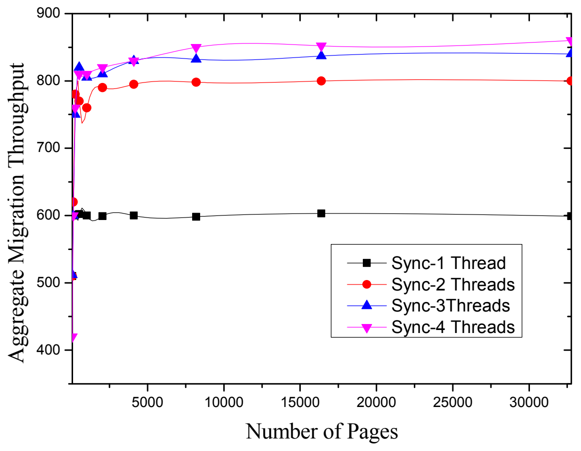

Figure-6. Throughput of Sync threads.

6. CONCLUSION AND FUTURE WORK

Scalability being a major concern in UMA based system NUMA addressed this key concern very effectively. With the advent of many core environments where memory is distributed among the different cores, it is challenging to design a thread scheduler along with proper data distribution across different nodes in an efficient manner. Since the modern architectures has increased there complexity by adding more number of cores and shared memory there are some problems in schedule planning between data and threads. Running dynamic applications with the help of OPENMP threads causes' unbalanced access for threads i.e. some threads will be sharing more data and some threads will be sharing less there is no proper load balancing. We have thoroughly investigated existing solution with respect to data placement and explored their limitation in current context (lack of dynamic adaption to run time directives in OpenMP and lack of automatic placer in the current LibNUMA APIs). We have proposed, investigated and evolved an efficient design for NUMA based machine for automatic data placement in scalable fashion, there by addressing the key concern (dynamic migration and replication), for improving Dynamic Multithreaded applications. This examination is done in the setting of planning a productive OPENMP runtime framework for various levelled NUMA architectures. A tight coordination of our Next-touch support inside of the NUMA-aware MARCEL user level threading library [19] is relied upon to establish the frameworks for dynamic scheduling threads and setting memory supports relying upon their affinities. It ought to empower a sharp dispersion of work and information inside of our FORESTGOMP OPENMP runtime which has been intended to keep running on these architectures [14].

REFERENCES

[1] C. Compiler. 2010. Thread affinity interface. http://software.intel.com/en-us/intel-compilers/.

[2] G. C. Compiler. 2010. Thread affinity interface. http://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html.

[3] AndiKleenSUSE Labs. 2005. An NUMA API for Linux" August 2004 A. Kleen. A NUMA API for Linux Tech. Rep. Novell-4621437.

[4] Kleen. 2005. A NUMA API for Linux. Tech. Rep. Novell-4621437.

[5] D. S. Nikolopoulos, E. Artiaga, E. Ayguadé and J. Labarta. 2001. Exploiting Memory Affinity in OpenMP through Schedule Reuse. SIGARCH Computer Architecture News. 29(5): 49-55.

[6] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson and C. D. Offner. 2000. Extending OpenMP for NUMA Machines. in SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Dallas, Texas, USA.

[7] H. Richardson. 1996. High Performance FORTRAN: history, overview andcurrent developments. Tech. Rep. TMC-261, http://hpff.rice.edu/publications/index.html

[8] S. Benkner and T. Brandes. 2002. Efficient Parallel Programming on ScaleShared Memory Systems with High Performance Fortran. Concurrency: Practice and Experience. 14: 789-803.

[9] Christiane Pousa Ribeiro, Jean-François Méhaut. 2010. MAi: Memory Affinity Interface. inria-00344189, version 6 -14.

[10] Charles Koelbel, David Loveman, Robert Schreiber, Guy Steele and Mary Zosel. 1994. The High Performance FORTRAN Handbook.

[11] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, Pierre-André Wacrenier. Dynamic Task and Data Placement over NUMAArchitectures: an OpenMP Runtime Perspective. published in International Workshop.

[12] T. Mu, J. Tao, M. Schulz, and S. A. McKee. 2003. Interactive Locality Optimizationon NUMA Architectures. In: SoftVis '03 Proceedings of the 2003 ACM Symposium on Software Visualization. New York, NY, USA: ACM. pp. 133.

[13] J. Marathe and F. Mueller. 2006. Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems. In: PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. New York, NY, USA: ACM, pp. 90-99. Online Available: http://portal.acm.org/citation.cfm?id=1122987.

[14] Christoph Lameter. 2006. Local and Remote Memory: Memory in a Linux/NUMA System. In Linux Symposium (OLS2006), Ottawa, Canada.

[15] Christian Terboven, Dieter anMey, Dirk Schmidl, Henry Jin and Thomas Reichstein. 2008. Data and Thread Affinity in OpenMP Programs. In: Proceedings of the 2008 workshop on Memory access on future processors (MAW '08), pp. 377-384, New York, NY. ACM.

[16] http://www.hpcsociety.org/Resources/Documents/6-9NOV2011-AMD Best%20practices%20for%20programming%20with %20openMP%20on%20NUMA%20systems.pdf.

[17] Marongiu, A. Capotondi, S. Member, G. Tagliavini, L. Benini and A. Multiprocessor. 2015. Simplifying Many-Core-Based Heterogeneous SoC Programming With Offload Directives. 11(4): 957-967.

[18] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, F. M. Quintão Pereira and F. Magno. 2014. Compiler support for selective page migration in

NUMA architectures. Proc. 23rd Int. Conf. Parallel Archit. Compil. - PACT '14. pp. 369-380.

[19] S. Thibault. 2005. A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines.

[20] S. Thibault, F. Broquedis, B. Goglin, R. Namyst, and P. a. Wacrenier. 2008. An efficient OpenMP runtime system for hierarchical architectures. Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics). Vol. 4935, LNCS, pp. 161-172.