An Analysis of Shared Library Performance on NUMA Architectures

Neeraj Kumar
neeraj.kumar@uwaterloo.ca

Hemant Saxena
hemant.saxena@uwaterloo.ca

Abstract

Most modern multicore systems these days are Non Uniform Memory Architecture (NUMA), which means they have multiple memory controllers with non-uniform access latencies across them. There has been significant amount of work done in exploring and mitigating performance penalty due to NUMA overhead. In previous works, NUMA-aware schedulers were proposed, sometimes with an objective of keeping data close to the processors working on them and sometimes to attain a trade-off between data locality and cache contention.

However, the effect of instructions on process' performance in context of NUMA has not been explored yet. In this work, we present an analysis about the performance hit on processes due to instructions on NUMA machines. We limit our focus to the impact of shared code, which by definition has a single copy and resides on one of the NUMA nodes. We also discuss various scenarios in which a single copy of these shared objects can harm the performance due to remote fetches and present some ideas to alleviate the problem.

1. INTRODUCTION

With the constantly increasing number of cores in modern multicore systems, non-uniform memory architectures are becoming more and more common. In its simplest form, a NUMA can have two processors with local memories connected to each other via an interconnect. Naturally, access to local memory is faster than the remote by a factor greater than 1, called the numa-factor or numa-overhead. With more processors, the interconnections become more complex, and the numa-factor can also be different for different pair of nodes.

With the introduction of NUMA architectures, problems with respect to data locality becoming bottleneck for application performance has drawn significant academic attention. Previous work by Brecht[10] and Broquedis et al.[11] explore placement decisions in context of data locality. In another recent work [12], the authors claim that contrary to older systems, modern NUMA hardware has much smaller remote wire delays, and so remote access costs are not the main concern for performance, instead, congestion on memory controllers and interconnects, caused by memory traffic from data-intensive applications, hurts performance a lot more.

Another work by Majo et. al. [15] tends to exploit the trade-off between cache contention and interconnect overhead while scheduling threads on NUMA machines. Running two threads of a process on the same socket avoids interconnect overhead but experiences a penalty in terms of cache contention, as last level cache are shared for threads running on the same socket, Whereas scheduling two threads on different sockets can avoid cache contention as two threads will use their local LLCs but the penalty of data fetch over interconnect can hit the performance in this case. Majo et. al. [15] have introduced N-MASS algorithm, which is a cache conscious scheduler for NUMAs.

The problems addressed in previous works mainly deal with the data-section associated with the program. We believe that a similar problem exists for text-section of the program in context of NUMA environment, albeit less aggravated because of the text-section being read-only. In this work, we present a detailed analysis for this problem in context of shared libraries, which by definition has a single copy across all memory nodes, and is loaded into memory when an object linked against it starts execution.

Text-section of a program mostly consists of instructions and read-only global data. In case of shared libraries, this section is unique across all memory nodes and will be shared by all the running processes linked against that library. As the text section is read-only and would be rarely evicted, it makes sense to have an extra cache for instructions. This is accomplished by having instruction only caches at L1 level, called L1-icache. Normally, as code size is dependent upon the program complexity, which is fixed, this means that the growth is not as bad as in the case of data. Additionally, program flow being more predictable than data access patterns, it is normally the case that active code can somewhat fit in the cpu caches. However, it is still not uncommon to find applications using big shared libraries. In cases when the size of shared library exceeds the cache-sizes or for some reason prefetching fails, the instructions will need to be fetched from the main memory. This can result in two problems, increased latencies due to instructions being fetched from remote node and increased overhead on memory controllers and interconnects due to congestion. In this work, we examine and analyze the cost of code on a remote NUMA node from both these perspectives and discuss the results. We also discuss some ideas that can be applied to mitigate the problem.

The remainder of the paper is organized as follows. In the next section, we discuss a little bit about shared libraries

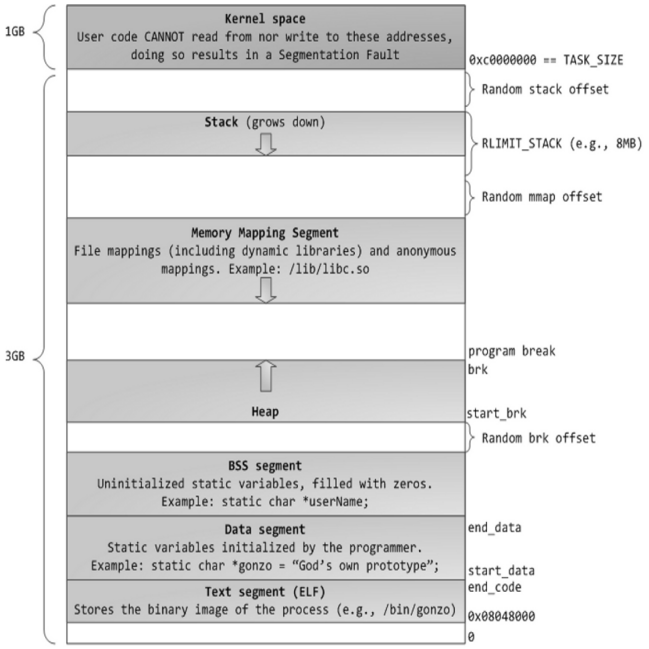

Figure 1: Anatomy of a program in memory, Image Source:[14]

and then talk about our experimental setup. In section 4, we will discuss at length the motivation, results and also the explanation of the results for each of the experiments performed. In section 5, we will discuss some ideas that could potentially improve the system performance with shared libraries on NUMA machines and talk about the roadmap to their implementation. Finally we conclude, highlighting the key observations from our experiments.

2. SHARED LIBRARY

Shared libraries are basically object files, that are loaded, (if not already) at runtime when a process linked against it starts execution. The pages remain in the memory until they are swapped out by pages from another program, based upon the page replacement algorithms[2].

Shared libraries are useful because of multiple reasons. They allow programmers to make use of existing code without increasing the executable sizes. Additionally, as there is a single copy of shared code, there are fewer page faults because its possible that some other process may have already loaded the shared code. In addition to this, shared libraries also make it easy to upgrade to newer versions of the code.

Figure 1 shows how a running program looks like in the memory. Shared libraries, can be loaded elsewhere in the memory, and the address mappings are stored in the memory mapping segment. Size of a shared library depends upon the text-section, (which is mostly the shared-code and read-only data) and the data and bss sections, which correspond to global and uninitialized data respectively. Figure 2 shows the result of *size* command on the libnuma shared library. It can also be seen that, most of the shared libraries on a normal linux-like systems are small, that is, less than 2 MBs. As such, they can easily fit in the processor caches,

```
  text    data     bss     dec     hex filename
 35939    1428    1212   38579    96b3 libnuma.so
```

Figure 2: output of *size* command on libnuma.so

```
text          data    bss    dec          libnames
891635        507856   23792  8423283    ./libqt-mt.so.3.3
11178278      151116   10168 11339562    ./i386-linux-gnu/libQtGui.so.4
18276710         520      16 18277246    ./libicudata.so.48
19267838      605812   32380 19906030    ./i386-linux-gnu/libLLVM-3.1.so.1
22197110     1486376   35184 23718670    ./libwebkitgtk-1.0.so.0
25564235      922020   87484 26573739    ./i386-linux-gnu/libQtWebKit.so.4
40142540     9266320  489648 49898508    ./x86_64-linux-gnu/libgcj.so.12
```

Figure 3: Sizes for some big libraries

and performance impact due to NUMA overhead would be negligible. However, some shared libraries are big, (figure 3 lists some big libraries) and we may have even bigger libraries in future. Therefore, it is worthwhile to evaluate the performance impact in context of NUMA machines and perform possible optimizations.

3. EXPERIMENTAL SETUP

All our experiments were performed on a NUMA machine with 4-sockets each housing a six-core AMD Opteron 8431 processor [8]. Cache related and topology related information about the NUMA hardware can be found at:

`/sys/devices/system/cpu/cpu*/cache`

`/sys/devices/system/cpu/cpu*/topology`

The L1, L2 and L3 sizes for the machine were 128, 512 and 6144 KBytes respectively. The following table shows where the core to socket mapping:

Socket Id	Core Ids
0	0, 4, ..4*i.. 8
1	1, 5, ..4*i+1.. 9
2	2, 6, ..4*i+2.. 10
3	3, 7, ..4*i+3.. 11

3.1 Performance monitoring

For our measurements, we relied on `RDTSC` assembly instruction[5] for getting CPU timing information and `perf stat` [3], for counting the cache misses, page-faults and stalled CPU cycles. We used libnuma, the NUMA policy library, for control and placement of threads and data on the NUMA nodes. [6] lists many functions that can be used to modify the default NUMA policy as required. The command line interface, `numactl` [7] provides various command-line arguments like `cpubind` and `membind` to bind processes and data to specific nodes.

3.2 Interconnect Overhead

Shared code is in essence very similar to read-only data, unless it is self-modifying. Therefore, all the observations with respect to NUMA overheads for read-only data can be generalized for shared code as well. In this section, we compute the overhead of fetching the data over the interconnect by comparing the access times for read-only data for local and remote memory access.

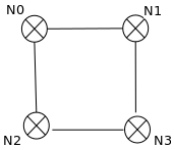

Figure 4: 4-node NUMA topology, N1, N2 and N3 are respectively horizontal, vertical and diagonal neighbours of N0

To make our measurements more accurate, we undermine the effect of cpu caches by flushing the cache line associated with the variable before every memory access. We do this by using the CLFLUSH[1] assembly instruction as follows.

```
// ...
const char* roData = "foo";
clflush(roData);
tmp = *roData;
// ...
```

It should be noted that, if the cpu cache is flushed before the computation, the total ticks elapsed will be the sum of time taken to fetch the operand from memory and execute the instruction.

$$T_{CFlush} = T_{MemFetch} + T_{decodeAndExecute} \quad (1)$$

Whereas, if the cache is not flushed, the data is too small and can easily fit in L1 cache, therefore,

$$T_{NoCFlush} \approx T_{decodeAndExecute} \quad (2)$$

From 1 and 2, we can compute the value for $T_{MemFetch}$, the results for which are labelled in the following table (all entries in clock ticks):

location	T_CFlush	T_NoCFlush	T_MemFetch
Self	944	330	614
Horizontal	1262	330	932
Vertical	1258	330	928
Diagonal	1570	330	1240

From the table above, it can be concluded that adjacent access is 1.5x costly whereas diagonal access is 2x costlier. These observed results agree with the fact that for 4-node NUMA machines, the topology is hypercube with $C = 2$, as discussed here[13].

3.3 Library Setup

We needed to call all the functions in shared library multiple times in different orders. This was difficult to implement with existing libraries as most functions have different signatures, and therefore, generating arguments and calling all of them can be a tedious task. To avoid this, we allowed ourselves the flexibility to generate shared libraries of appropriate sizes using a perl-based code generator or alternatively using C++ Template Metaprogramming.[4]

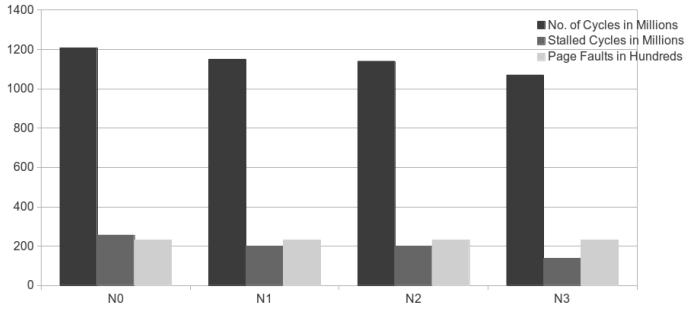

Figure 5: Comparison of remote vs local library. N0, N1, N2, N3 are nodes in NUMA machine. Shared library is at N3

We generated multiple shared libraries, and carried our experiments on them. The results of which are presented in the next section. All the functions in the library were indexed in a *functionTable*, from where they could be called in various different ways.

4. ANALYSIS

The experiments performed by us can be broadly divided into two categories, evaluating performance impact due to NUMA overhead and that due to congestion over interconnect and memory controllers. We discuss each of these separately in following sections.

4.1 Impact of NUMA overhead

4.1.1 Remote vs Local Library

In previous section, we computed the interconnect overhead and claimed that similar overhead should also exist in case when shared code is being fetched from a remote node. To verify our claim, we ran our experiments on a shared library with 0.1 million functions, amounting to about 60 MBs of text section. The scheduler placed the shared library on node N3. Then we created a thread which called functions from this library in sequential order with an offset equal to L3 cache size, which means we called functions in a way that every next function called could be a cache miss. We ran this thread multiple times on each of the four nodes $N0, N1, N2$ and $N3$. The aggregated results are shown in figure 5.

We measured the total number of CPU cycles consumed, stalled cycles and page faults in each case. We observed that in the case when our thread was running locally (i.e. on node N3), it consumed minimum number of CPU cycles. Number of stalled cycles were also minimum when library is placed locally, high number of stalled cycles in remote case indicates that a lot of CPU cycles were idle due slowdown for data fetch over the interconnect. We measured page faults to make sure that we are fetching equal number of library pages in all four cases. As expected, the number of page faults was found to be equal in all four cases.

It can be verified that N0 and N3 being on the opposite corners of diagonal, have the maximum interconnect delays. The results exhibit a similar pattern, as in, maximum cycles are consumed by node N0 and the least cycles for the local node N3. The maximum slowdown is for the farthest

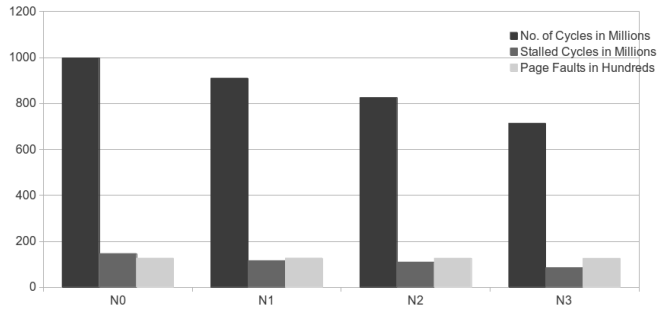

Figure 6: Performance with small sized functions. Shared library is at N3

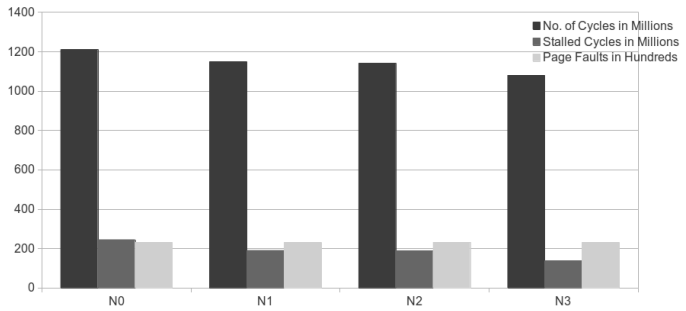

Figure 7: Library functions called in random order by main thread. Shared library is at N3

node and was found to be 12.9%. Based upon the interconnect delays, we may expect the execution on remote node along diagonal to perform 2x slower compared to local execution, which is not the case. This means that CPU caches combined with prefetching, lessen the performance hit that could otherwise be observed.

4.1.2 *Small vs Big functions in Library*

In our previous experiment, size of each function was found to be approximately 600 bytes. We believed that with smaller size functions, there will be less spatial locality and we could see an increased impact on the performance. In this case, the number of functions was increased, but the size of each function was reduced to approximately 140 bytes per function.
The generated functions looked like this:

```
int f_i(){return i;}
```

These functions were indexed as usual and called sequentially with an offset equal to L3 cache size, same as in previous case. Other experimental settings were similar to previous section and shared library was also placed on Node3. The results for this are shown in figure 6.

It can be seen that the overall execution takes fewer cycles. This is so because the functions in this case were simpler and compiler could generate better code for them. The maximum slowdown is again for the farthest node and was found to be close to 39.7%. However, it can also be seen that the effect of library being on the remote node is more pronounced here compared to the previous case. We believe that this is because in the previous case, a prefetched function would exhibit a certain spatial locality of 600 bytes, which is reduced to 140 bytes in this case.

4.1.3 *Various probability distributions of function calls*

Until now we were calling the functions from the shared library in a sequential order. In real application scenarios it is not necessarily the case that an application is using all the functions of a library and that too in sequential order. In an attempt to simulate some real application scenarios, we tried varying our function calling pattern to random and zipfian distributions.

Sequential - Figure 5 shows the results for sequential calling pattern

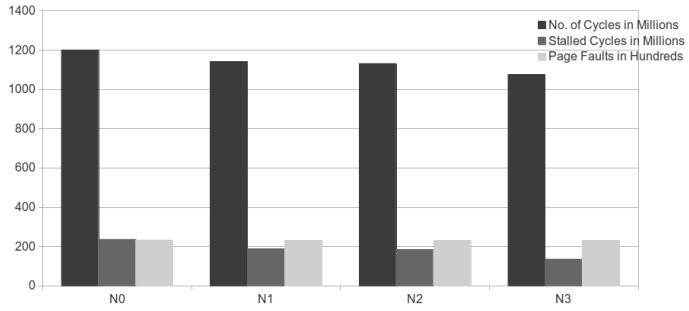

Figure 8: Library functions called in Zipfian order (with alpha=0.5) by main thread. Shared library is at N3

Random - Figure 7 shows the results for calling library functions in random order.
Zipf - Figure 8 shows the results for calling library functions in zipfian order.

The maximum slowdown was again for the farthest node in each of the two cases. The performance impact was 12.05% for the random distribution, whereas it was 11.57% for the zipfian order. According to zipfian distribution, there will be a bunch of functions which will be called more frequently than others. Therefore, the instruction cache can perform better in this case and consequently the performance impact can be lesser.

4.1.4 *Impact of Library Sizes*

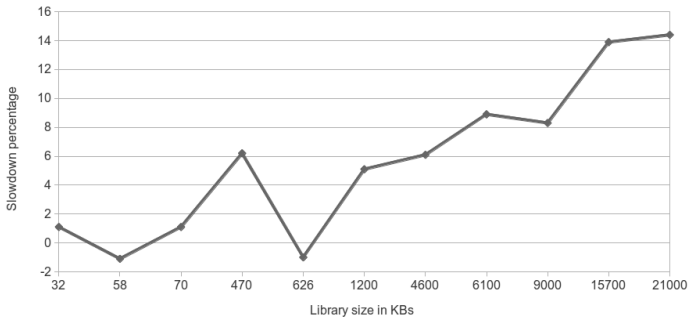

Figure 9: Slowdown with increasing library size

In all of the above experiments, the shared library we used had about 60 MB of text-section. As discussed earlier, most of the modern libraries are lesser than 20 MBs, and very few are of order of 40 MBs. In this section we will vary the text-section size for the libraries and see the impact on system performance.

As the size of text-section is directly proportional to the number of functions, we vary the size by simply increasing the number of functions. We generated libraries of sizes varying from 32KB to 21MB, see Figure 9. We kept the size of each function same at around 600 Bytes, and increased the number of functions available in each library. We believe that when the library will be small enough to fit in the cache, there will be very few instruction fetches across the interconnect. But with the increasing sizes, its likely that there will be more cache misses impacting the performance.

To measure this impact we introduce the following metric:

$$N_{cyclesPerCall} = N_{totalElapsedCycles}/numFunctionCalls \quad (3)$$

This metric is necessary to compare performance across libraries of two different sizes, as with increased sizes, the number of page-faults to load the library into memory will also increase.

The main thread calls the library functions in sequential order, each function being called only once. Setup was very much similar to above experiments, we had our library on one node and main thread was made to run on each of the four nodes. For our comparison, we selected the local node (housing the shared library) and farthest node from this local node, and computed the ratio of their $N_{cyclesPerCall}$ metric as follows:

$$R_{slowdown} = N\_Far_{cyclesPerCall}/N\_Local_{cyclesPerCall} \quad (4)$$

We then plot the values of $R_{slowdown}$ in percentage for different values of library text-section sizes. The result is shown in Figure 9. As expected, we found that when the library is small enough to fit in the cache, not much slowdown is observed. It has even gone to negative in some cases. But as the size of library increases and the cache misses increase, we see the the effect of slowdown becomes more pronounced. The slowdown was found to be about 14% for a library with 21MB of text section, but later stabilizes around this value.

4.2 Impact of congestion

Dashti et al. [12] claim that in modern NUMA systems, wire-delays are not the only source of performance hits, but congestion on interconnect links and in memory controllers can also dramatically hurt performance. They propose an algorithm which avoids, traffic hot-spots to prevent congestion on interconnect and memory controllers.

In case of data, an application may exploit data-level parallelism by splitting the data, such that each thread mostly works on a distinct data-set. In such a scenario, threads are lightly coupled and is ideal for parallelism. It also doesn't make much sense for multiple threads to operate on same data set in parallel, and therefore it is less likely that traffic hot-spots exist in such a system, and even if they do exist, can be avoided by careful allocation of data.
Such a situation, is however more likely to happen in case of shared libraries, as they can only have a single copy across the system, making the NUMA node housing the library, a potential traffic hot-spot.

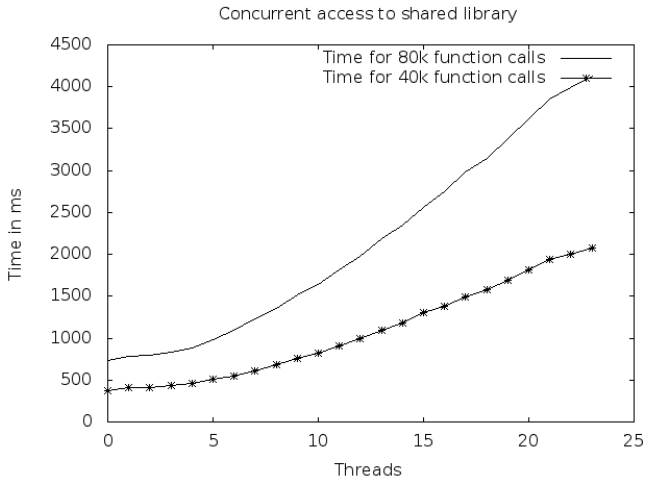

Figure 10: Effect of concurrent access to shared library on increasing the number of threads.

In the previous sections, all our experiments were conducted independently, that is, the shared library was being accessed only by a single thread. In real application scenarios, multiple threads and processes can access shared code concurrently. In this section, we present an analysis of what happens when multiple threads try to access the library concurrently.
For this analysis, we are using a smaller more-realistic shared library with 10k functions amounting to about 20M of text-section. The shared library was placed on Node 0. The driver function that calls the functions from the library can now be executed concurrently by multiple threads. We don't modify any shared-data, therefore we didn't need any kind of locking or synchronization mechanisms.

Figure 10 shows the plot of time taken for n-threads to execute 40k and 80k function calls from the library for different values of n. It can be seen that the curve is nearly flat in the beginning, but picks up later with increase in congestion at the interconnect and/or memory controller.

We also tried various combinations, the results of which are presented below:

```
(Library residing on node 0)
Number of running threads : 1
Core 0 : 375.04 ms
Core 3 : 386.42 ms

Number of running threads : 2
Core 0 and  4 : 387.20 (Local memory + shared cache)
```

```
Core 1 and  2 : 402.50 (Remote memory +
                        No shared cache)
Core 3 and  7 : 405.48 (Remote memory +
                        shared cache)

Number of running threads : 3
Core 0, 4 and 8 : 392.31 (All three local +
                          shared cache)
Core 1, 2 and 3 : 411.31 (All remote +
                          No shared cache)
```

Based upon these values, it can be inferred that there is a performance overhead, whenever a new thread is added to the system. Sometimes, this overhead gets reduced when the cores have a shared cache. This is so because the threads are accessing the same data, hence the caches will collaborate and not contend. The extra performance overhead with introduction of new threads can be attributed to congestion on both memory controller and the interconnect. We believe that the interconnect congestion is less evident with smaller number of threads but becomes more pronounced when the number of threads increase, which explains the initial flat curve in the graph. At later stages, as each node can have multiple cores fetching the shared-code from the memory, their caches may be co-operate, resulting in the an asymptotic curve towards the end.

5. FUTURE WORK

Having analyzed the impact of shared code from the perspective of locality and congestion, we are in a position to suggest some potential improvements that can help to mitigate the overheads due to NUMA architecture in context of shared libraries.

The default NUMA scheduler currently loads the shared library on the node that triggered the page-fault for the first library page, which is random. It can be on any of the NUMA nodes. Such an algorithm is speculative, in the sense that, it expects in future, there may be processes closer to this node which may potentially use this shared library, and the amortized performance penalty is not much. However, this may penalize the currently executing program, in case the library is not loaded on the local node. There are some alternatives, in which we can certainly do better than the default library placement.

Performance-aware loading Whenever a program executes, first of all the dynamic loader is invoked with task to load all the shared libraries the program was linked against (if not already loaded). The programmer knows which all cores he is going to use, for example, if the programmer uses cores from two of the four sockets, and the shared library was not already loaded, it seems reasonable to load the shared library on one of the two nodes. Programmer can explicitly request such a loading or the compiler may identify this during code analysis, and add relevant information in the binary. This algorithm comes with almost no overhead, just a slight optimization to make the loader NUMA-aware.

User text Replication Another plausible alternative is replication, which is essentially making additional copies of the text-section, so that no access for instruction needs to go over the interconnect. The solution sounds great as it completely eliminates the NUMA overhead but it should be noted that it has its own costs. It introduces a new problem of keeping all replicated copies in sync, which is a less severe problem for shared code as it is read-only. The kernel will however, still need to sync every node if any of them undergo a page-fault. The advantage being, the system can be implemented transparently in the kernel and the programmer doesn't has to know about it.

Distributed Loading Another possible solution is loading parts of shared library across all the NUMA nodes. Such a solution will prevent shared library from becoming a traffic hot-spot and also amortize the cost of remote NUMA access. The system however will be more complicated as the kernel will need to keep track of pages on both remote and local nodes and seems worthwhile only if the benefits outnumber the complexities.

In our future works, we aim to further explore these ideas and implement some of these and present the outcome of our implementations and how they fare with the challenges presented in this work. One possible direction of our work can be to use *carefour* [12], and possibly apply their solution in context of shared libraries, as they seem to address similar set of problems.

6. RELATED WORK

The work done by Brecht [10] explains that in the case of NUMA scheduler the question of which processor becomes more important than how many processors to be assigned for a task. His results have shown thread placement decision is critical in large scale multiprocessors. Work done by Broquedis et al [11] is also on the similar lines and talks about dynamic task and data placement over NUMA architecture. In another recent work [12], the authors claim that contrary to older systems, modern NUMA hardware has much smaller remote wire delays, and so remote access costs should not be the main concern for performance, instead, congestion on memory controllers and interconnects, caused by memory traffic from data-intensive applications, hurts performance a lot more. Majo et. al. [15] have exploited the trade-off between cache contention and interconnect overhead for their NUMA aware scheduler.

Apart from these, there also have been some work concerning user-text replication by the Linux community. [9] have performed a similar analysis but in context of system calls, claiming a benefit of about 41% with cold cache.

7. CONCLUSION

In this paper, we presented a comprehensive analysis of shared library performance on NUMA architectures. We begin by stating possibility of a performance hit but based on the results from our experiments, we can conclude that the effect of NUMA overhead due to non-locality of memory is not as pronounced for shared-code as it is for data. This is primarily because shared-code is read-only, and therefore exhibits better cache usage. Based on our experiments, we found that for zipf distribution, (which most closely resembles the practical case) the performance overhead in worst case was found to be about 11%. This was when library and thread were scheduled on farthest nodes. We also found that

NUMA overhead increases with increasing library sizes but never exceeds 15% worst case overhead.

We also found that when multiple threads concurrently try to access the shared-code, the performance drops and keeps on dropping linearly as more threads are added to the system. With 24 threads accessing the library concurrently, the overhead due interconnect congestion, is as much as 5x times the single-thread case. This is in agreement to the observation by Dashti et al.[12], that interconnect congestion due to multiple threads accessing the data can hurt the performance more than NUMA penalty would.

To address both these problems together, page replication happens to be the best solution. We have also discussed couple of other solutions that can possibly improve performance, by solving one or both of these problems. In our future works, we plan to explore these ideas further and come up with a working solution.

8. REFERENCES

[1] clflush instruction `http://x86.renejeschke.de/html/file_module_x86_id_30.html`.

[2] Page replacement algorithms, `http://en.wikipedia.org/wiki/Page_replacement_algorithm`.

[3] Perf wiki `https://perf.wiki.kernel.org/index.php/Main_Page`.

[4] `http://en.wikipedia.org/wiki/Template_metaprogramming`.

[5] `http://en.wikipedia.org/wiki/Time_Stamp_Counter`.

[6] `http://linux.die.net/man/3/numa`.

[7] `http://linux.die.net/man/8/numactl`.

[8] `http://products.amd.com/pages/OpteronCPUDetail.aspx?id=555`.

[9] `http://real-time.ccur.com/docs/default-source/white-papers/kernel-page-replication.pdf?sfvrsn=6`.

[10] T. Brecht. On the importance of parallel application placement in numa multiprocessors. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, pages 1–1, Berkeley, CA, USA, 1993. USENIX Association.

[11] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier. Dynamic task and data placement over numa architectures: An openmp runtime perspective. In M. MÃijller, B. Supinski, and B. Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism*, volume 5568 of *Lecture Notes in Computer Science*, pages 79–92. Springer Berlin Heidelberg, 2009.

[12] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on numa systems. *SIGARCH Comput. Archit. News*, 41(1):381–394, Mar. 2013.

[13] U. Drepper. What every programmer should know about memory, 2007.

[14] G. Duarte. Anatomy of a program in memory, `http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory`, 2009.

[15] Z. Majo and T. R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 11–20, New York, NY, USA, 2011. ACM.