Characterization of cache performance interaction on NUMA systems

Andrew Heard
University of Waterloo
David R. Cheriton School
of Computer Science
Email: andrew.heard@uwaterloo.ca
Website: cs.uwaterloo.ca/~asheard/

Behrooz Shafiee
University of Waterloo
David R. Cheriton School
of Computer Science
Email: bshafiee@uwaterloo.ca
Website: cs.uwaterloo.ca/~bshafiee/

***Abstract*—In recent years, Non-Uniform Memory Access (NUMA) systems, which partition memory amongst physical processors, have become the de facto standard. In these systems, accessing memory local to a processor incurs a lower performance penalty than remote memory. This study measures application performance based on three cache access patterns, sequential, uniform random and Zipf (at different α values), as well as different dataset sizes.**

Cachetest was used for benchmarking and it was modified to accommodate the Zipf distribution. A combination of the sequential and random uniform patterns was used to model this distribution, as well as a Zipf random number generator. Benchmarks used the Linux numactl command to allocate memory locally or remotely.

For the sequential pattern, we found that the most dramatic decreases in performance were seen when dataset sizes approached and exceeded the L3 cache size in the system. Performance slowdowns of over 60% were seen when accessing remote, instead of local, memory. For the random uniform pattern, performance decreased gradually as the dataset size increased but diverged significantly as it approached the L3 cache size. In this range, slowdowns of over 40% were observed. For the Zipf pattern, performance decreased significantly as the α value became smaller, resulting in a more random distribution. Performance also decreased as the dataset size increased. Trends for the Zipf pattern were less clear.

Results from this experiment demonstrate the impact of remote memory access on NUMA systems and that this impact is generally a factor of the dataset size. We also found that sequential access patterns suffered larger performance penalties than random, when running remotely. Zipf performance was largely a factor of the α value. Although we hypothesize that the Zipf pattern most closely models real-world applications, further studies will be needed to confirm this.

I. Introduction

In the past, symmetric multiprocessing (SMP) systems, which share a single memory controller amongst all cores, were the dominant architecture for multiprocessing. As the number of cores has increased, these systems have become impractical for scaling due to memory contention [4]. In order to address this problem, non-uniform memory access (NUMA) systems, which partition the available physical memory amongst CPUs, have become more prevalent. In NUMA, each CPU's cores have high-speed access to their local memory through the chip's memory controller. Access to remote memory is performed through a cross-chip interconnect, which increases the access latency and bandwidth is shared.

NUMA systems present several new challenges with regards to which CPU to schedule a process on. Running several memory-bounded processes on the local CPU can result in cache contention. Running them on a remote CPU could alleviate cache contention but also reduces memory access performance. This presents a trade-off between data locality and avoiding cache contention [6]. The purpose of this course project is to understand how much the NUMA penalty affects application performance. We hypothesized that the NUMA performance penalty may not be significant; however, as it has been measured and shown in our results section, the NUMA penalty can be significant and can severely degrade applications' performance. In order to measure the effect of the NUMA penalty, we will set up different

experiments with varying factors to evaluate their effect on application performance.

For the remainder of this paper, Section 2 explores the limited related work available, including a model of cache access patterns of real world applications using stack distance as a metric and another paper, which explored workload performance using various memory mapping strategies. Section 3 describes several distributions used to model cache access patterns, including sequential, uniform random and Zipf. Section 4 describes the setup used to evaluate application performance based on these cache access patterns using a benchmark tool called Cachetest. This section also explains how memory was allocated locally or remotely, and at what sizes. In Section 5, we present the results of these experiments, including performance differences between local and remote memory allocation. In Section 6, we conclude with general patterns of the performance penalty introduced by remote memory under these different distributions and finally, we propose future work that would allow these results to be generalized to real world applications.

II. Related Works

There is limited directly related work, which explores cache access patterns and their performance in the context of NUMA systems. Kotera *et al.* have analysed cache access locality of several benchmark applications from SPEC CPU2006 using Stack Distance Profiling (SDP) [5]. SDP estimates the temporal re-use of a thread of an application using counters that get incremented for cache hits and decremented for cache misses [2]. It was found that almost all of the Stack Distance Distributions (SDD) for these benchmarks could be modelled using Zipf's Law [5].

However, Alexander Szlavic, points out that true SDP has only been implemented in simulation and that SDP estimations using hardware counters only apply to fully-associative caches [8]. In contrast, our system's L3 cache is a 48-way set-associative cache. Nonetheless, the SDD results provide some basis for the hypothesis that real-world applications follow a Zipfian cache access pattern, hence our use of Zipf in this study.

To provide background on NUMA performance penalties, Majo and Gross measured remote memory latency to be roughly double that of local memory, while bandwidth was roughly equal [6]. However, they noted that this bandwidth was shared between all processors in the system, whereas the local access through the on-chip memory controller was not [6]. This study also provided motivation on the possible benefits of running an application remotely, in order to exploit the trade-off of latency vs. memory contention [6].

III. Memory Access Patterns

It is important to specify the pattern in which cache lines are accessed. To the best of our knowledge, there has been no previous study on this subject. Therefore, we decided to use three different distributions including: Sequential, Uniform Random, and Zipf. Although, these different distributions provide a wide range of different access patterns, it would still be beneficial to measure and map the cache access patterns of real workloads. In order to access the memory with a specified distribution, we generated samples from a specified distribution and stored them in an array. Memory is then accessed according to the array elements. In this way we don't introduce the overhead of sample drawing during the execution loop. In the following, each of the access pattern distributions are briefly explained.

A. Sequential

In the sequential access pattern, cache lines are accessed sequentially, that is to say one after another, in order. First, the head cache line is accessed, next the second cache line is accessed, and this continues until the last cache line. Finally, after reaching the last cache element, the first element is accessed again and this will repeat until the benchmark has finished. Figure 1 depicts the Sequential access pattern.

It's clear that in this scenario, the application does not significantly benefit from the cache and we expect this access pattern to have the lowest performance relative to the other patterns. We believe that this access pattern is far from a real application's access pattern and consider it as a boundary. However,

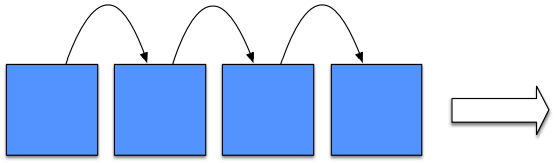

Fig. 1. Sequential Pattern

one factor, which might increase the performance under the sequential access pattern is the prefetching effect. Due to the predictability of the next cache line required by the sequential access pattern, the hardware prefetcher may be able to prefetch the next cache line, which would be beneficial to performance.

B. Uniform Random

In the uniform random distribution, each member of the distribution has equal probability to be accessed. In the context of caches, it means that every cache line has the same chance to be accessed next. Figure 2 shows the uniform random access pattern.

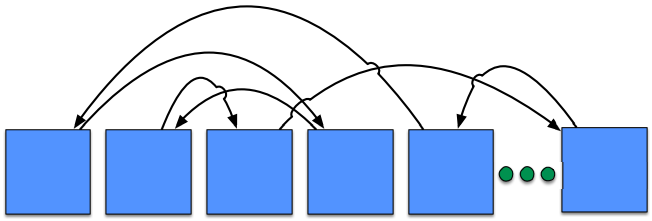

Fig. 2. Random Pattern

Although the random uniform distribution is a closer guess to the access patterns of real world applications, compared to sequential, it still does not completely reflect the access patterns of these applications because in real workloads, some data are often more popular than others, and thus, accessed more frequently.

C. Zipf

There are some events and quantities, which usually have a typical value. For example, human height is usually around 180 cm and although there is a small amount of deviation, it's not usually significant. This means that it is possible to see humans with heights of 150 cm or 220 cm, but it is not possible to find a human with a height of 400 cm or 10 cm [7]. These types of events follow Zipf's law and can be modelled by a Zipfian distribution, which was first proposed by Auerbach [1]. Zipfian distributions have an important parameter, which is called the power of distribution or α. In order to understand the effect of α on the distribution, we draw 50 random samples from the Zipf distribution with four different values of α. As it can be seen in Figure 3, the smaller the value of α, the more scattered the samples will be. As α becomes bigger, the samples will be more concentrated around a popular value.

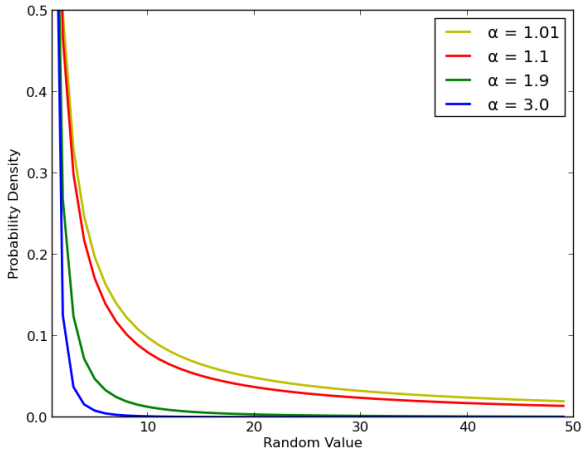

Fig. 3. 50 different samples drawn from Zipf distribution with different α values

We believe that a Zipfian distribution is the closest access pattern to that used by real world applications, because in real applications, some data are often more popular than the rest. In a related study, Kotera *et al.* investigate the distribution of cache access behaviour using stack distance profiling [5]. Stack distance profiling approximates cache hits and predicts when an access will result in a cache miss. They showed that the stack distance distributions almost all conform to Zipf's law; however, their metric is different from ours (accessing a cache line) and we can't definitively conclude the same pattern. In the following, we show how we implemented the Zipf access pattern. The Zipf access pattern is similar to the random access pattern, except the fact that each random cache line has an associated Zipf

value as well. The Zipf value is applied using the following method: for each Zipf value n, the first n cache lines will be accessed, and then the next random element will be followed. In this way, we assume the most popular cache line is the first cache line and the second most popular is the second cache line, and the same pattern for the rest of elements. This scenario is shown in Figures 4 and 5.

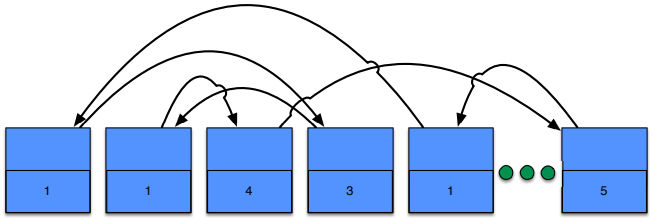

Fig. 4. Zipf access pattern, Step 1

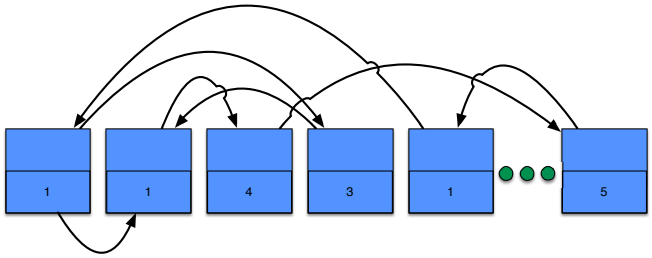

Fig. 5. Zipf access pattern, Step 2

IV. Experimental Setup

A. Cachetest

In order to measure application performance under the described cache access distributions, the Cachetest tool was used. As described by the author of the tool, "Cachetest provides the infrastructure for machine-level profiling of custom memory access patterns" [8]. Cachetest is unique in the way that it allocates elements to be accessed, such that they will be mapped to different cache lines. Contrast this with simply allocating an array of 32-bit integers on a machine with 64 byte cache lines, where 16 array indexes are mapped to the same cache line.

Cachetest is composed of a buffer allocator, a distribution generator, an execution engine and a profiler, as shown in Figure 6. The buffer allocator allocates memory of a specific size, to be accessed during execution. The distribution generator generates a pattern of memory accesses, such as sequential, uniform random or Zipf. The execution engine iterates over the generated pattern to perform the memory accesses. The profiler uses the hardware performance counters to generate statistics, such as cache hit, miss or access rates for each of the levels of cache.

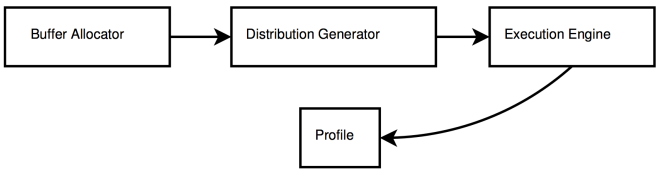

Fig. 6. Cachetest Architecture [8].

B. Implementation Changes

In order to perform this experiment, we needed to modify the distribution generator so that it could generate a pattern that matched the Zipf distribution. This involved generating a random uniform backbone pattern, much in the same way as was done for the uniform random distribution, but also adding a random Zipf value to each node. This Zipf value was generated using an algorithm based on source code provided by Christensen [3]. The Zipf value was generated in the range of one to the total number of nodes in the uniform pattern, minus one.

The execution engine was modified to traverse the nodes in the uniform random pattern but, for each node, sequentially traverse the sequence of nodes, starting from the beginning node each time, by the Zipf value. Based on the range of Zipf values generated, it was possible for the entire sequence of nodes to be traversed although this is less likely to happen than shorter traversals. The nodes at the beginning of the sequence are accessed more frequently, which models the Zipf distribution. As the α value increases, these Zipf values become less random and are, or are close to, one.

C. Experimental Factors

For our experiments, we adjusted the following factors:

- Local vs. Remote memory allocation.
- The size of the dataset, or buffer.
- The generated distribution:
  - Sequential
  - Uniform Random
  - Zipf (With Different α Values)

Cachetest provides a command-line interface and the following template was used:

```
cachetest -c {CPU core #}
          -e {recorded CPU events}
          -d {time in seconds}
          {memory size in KB}
          {output directory}
```

For each distribution, the following additional parameters were added:

- Sequential: none
- Random: -r random #, which was generated using the bash variable $RANDOM.
- Zipf: -z α value, the z argument is not part of the standard Cachetest tool and was added to allow us to enable the use of the Zipf distribution and set its α value.

The duration, set with the argument *d*, was fixed at 20 seconds for all experiments.

The dataset sizes used were 500, 1473, 2447, 3420, 4394, 5368, 6341, 7315, 8288 and 9262 KB.

The numactl command was used to control where memory was allocated. All Cachetest commands were prepended with *"numactl –membind=0"*, which ensured that memory was allocated in Node 0 of the system. In order to change whether this memory was accessed locally or remotely, the cachetest command's *c* argument was changed to either a CPU # that belonged to Node 0 for local memory or one that belonged to a different node for remote memory.

The α values used for the Zipf distribution were 4.05, 3.05, 2.05, 1.55, 1.45, 1.35, 1.25, 1.15 and 1.05.

Each experiment was run 20 times and the average was computed. Initially, we ran each experiment 100 times but reduced this number to 20 when it was found that the standard deviation was still less than 1%. This reduced the total amount of time to perform all experiments by a factor of 5 and allowed us to deal with such a large number of permutations of variables.

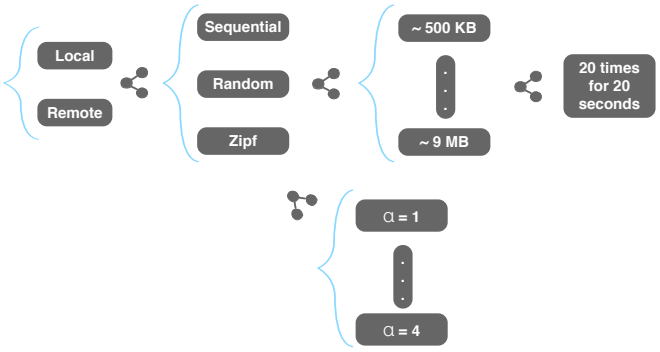

Fig. 7. The total set of experiments included all permutations of the above variables.

D. System Specifications

The experiments were run on a system with four AMD Opteron Six-Core Processors running at 2.4 GHz each. Each core had 128 KB of L1 cache and 512 KB of L2 cache, which were both private to the core. Each processor had 6 MB of L3 cache that was shared among the cores. The system contained 64 GB of main memory, with 16 GB local to each of the four NUMA nodes. The Operating System used was Ubuntu 12.04.1 LTS using version 3.2.0-29 of the Linux kernel.

V. Results

A. Sequential Pattern

1) Absolute Performance: The performance of sequential access, as shown in Figure 8, was the worst case scenario in this experiment, especially as the data size became larger than the caches. Performance levelled off during two distinct ranges of data sizes. The first range was when the dataset size was greater than the L2 cache of 512 KB but less than the L3 cache of 6 MB. Secondly, performance bottomed out at a dataset size of roughly 7 MB and was roughly constant up to the maximum value tested of just over 9 MB. The most dramatic decreases in performance were observed just after the dataset size

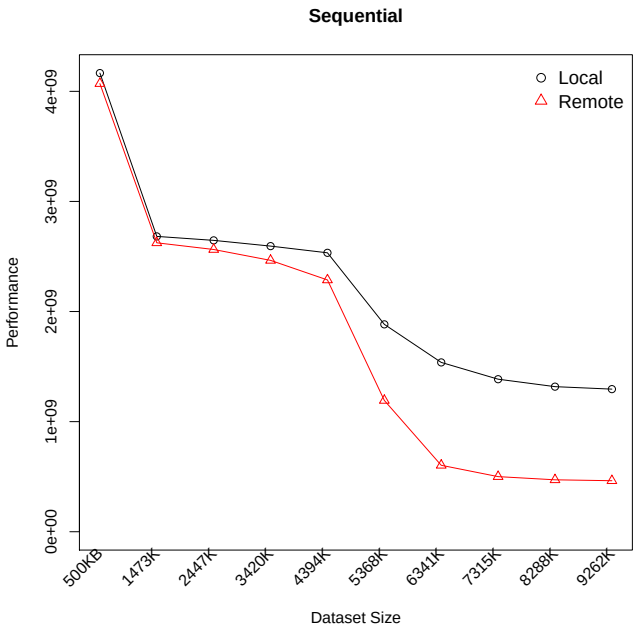

Fig. 8. The performance, measured in number of loop iterations completed, of local and remote sequential cache accesses using varying dataset sizes.

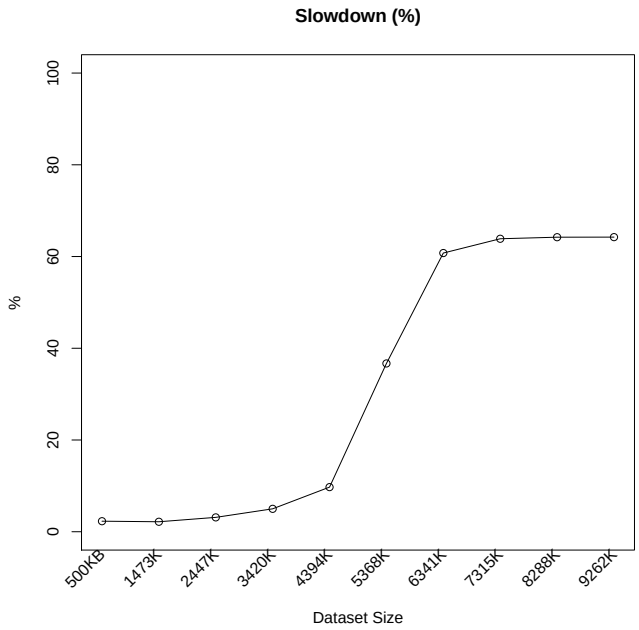

Fig. 9. The performance difference, measured in percentages, of local vs. remote sequential cache accesses using varying dataset sizes.

exceeded the L2 cache size and as it approached the L3 cache size between 4 and 6 MB.

2) Relative Slowdown: As shown in Figure 9, the greatest differences in performance of local compared to remote memory accesses were observed as the dataset size approached the L3 cache size. This is likely due to a much higher L3 cache miss rate, though this was not measured. The miss rate is likely to have increased due to the nature of sequential accesses, where cache lines at the beginning of the loop iteration have been evicted by the time a single iteration has completed. Thus, all memory accesses require fetching the data from main memory, which in this case incurs the performance penalty associated with remote access.

B. Uniform Random Pattern

1) Absolute Performance: Similarly to sequential accesses, Figure 10 shows the large decrease in performance that was observed as the dataset size exceeded the size of the L2 cache. However, unlike sequential accesses, no levelling off of performance occurred. Instead, performance gradually decreased as dataset sizes approached, and eventually exceeded, the size of the L3 cache.

For uniform random, it is likely that some memory accesses were fetched from caches, even when the dataset size increased past the L3 cache size. This would explain why we don't see a rapid change around this range.

2) Relative Slowdown: Performance differences between local and remote accesses, as shown in Figure 11, were very low up until around a dataset size of 5 MB, when it was approaching the L3 cache size. However, it is at this point where the performance of the local and remote executions began to diverge significantly, which we hypothesize as being due to the many requests requiring remote memory accesses, instead of from the caches.

C. Zipf Pattern

1) Absolute Performance: The performance of the local Zipf access pattern, as shown in Figure 12, behaved as expected with respect to α values. As α values decreased, the Zipf values generated became more like a uniform random distribution. At α values above 2, virtually all Zipf values generated

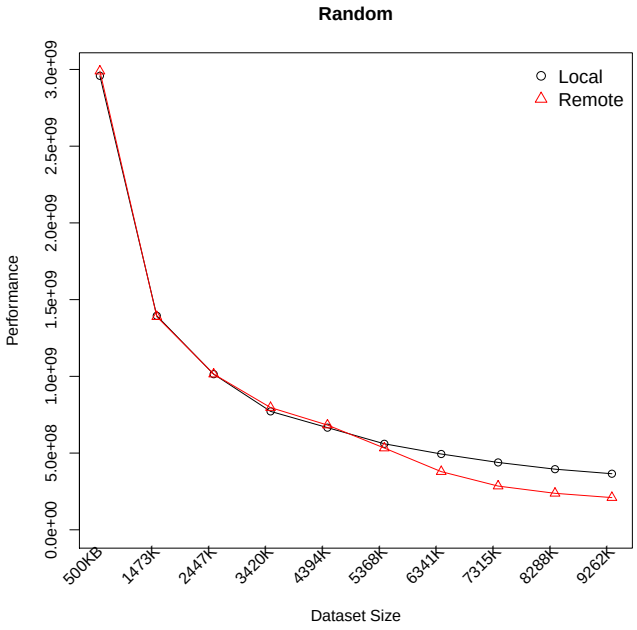

Fig. 10. The performance, measured in number of loop iterations completed, of local and remote uniform random cache accesses using varying dataset sizes.

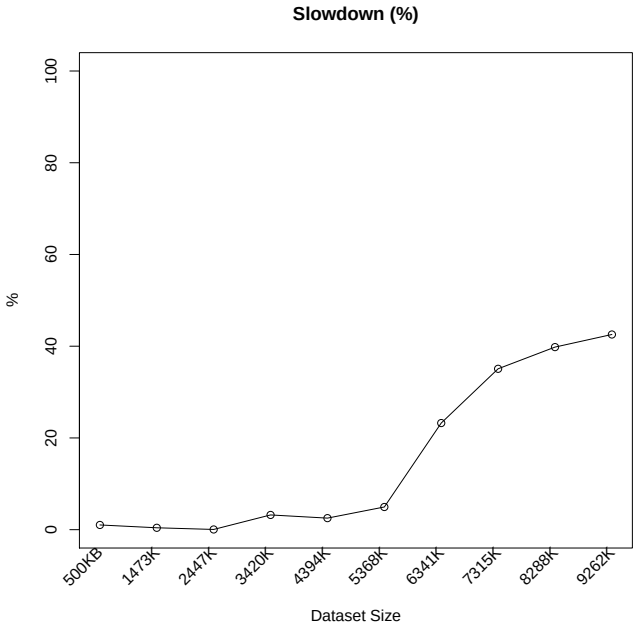

Fig. 11. The performance difference, measured in percentages, of local vs. remote uniform random cache accesses using varying dataset sizes.

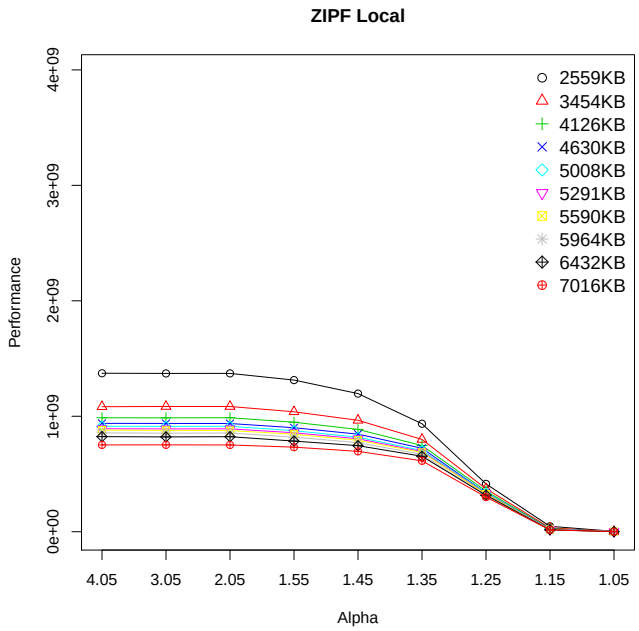

Fig. 12. The performance, measured in number of loop iterations completed, of local Zipf random cache accesses using varying dataset sizes and Zipf alpha values.

were 1, meaning that the sequence of nodes only had to be traversed sequentially for a short path, which is reflected in the performance results. As α values became smaller than 2, the probability of a long sequential traversal greatly increased, which could explain the significant decreases in performance that were seen after this point.

The performance of the remote Zipf access pattern, as shown in Figure 13, followed similar trends to the local results, which can be seen by the same curve shape as that in Figure 12. However, the absolute performance values were lower for the remote experiment. Presumably, this is due to the performance penalty incurred for accessing remote memory during the L3 cache misses.

2) Relative Slowdown: The slowdown incurred by remote access of the Zipf pattern, shown in Figure 14, was generally more severe for larger dataset sizes. However, this finding is not as clear as with the other access patterns. For example, for some α values the slowdown with a dataset size of 6432 KB was less than that of one with 5964 KB. The same can be said for many other pairs of dataset sizes.

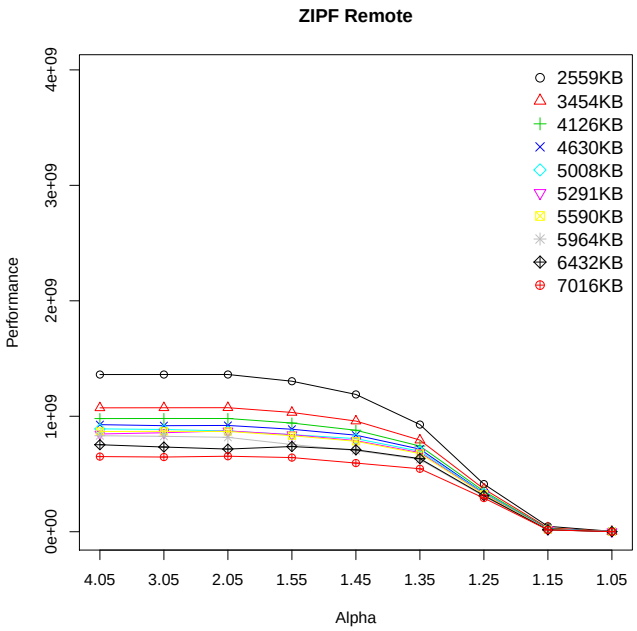

Fig. 13. The performance, measured in number of loop iterations completed, of remote Zipf random cache accesses using varying dataset sizes and Zipf alpha values.

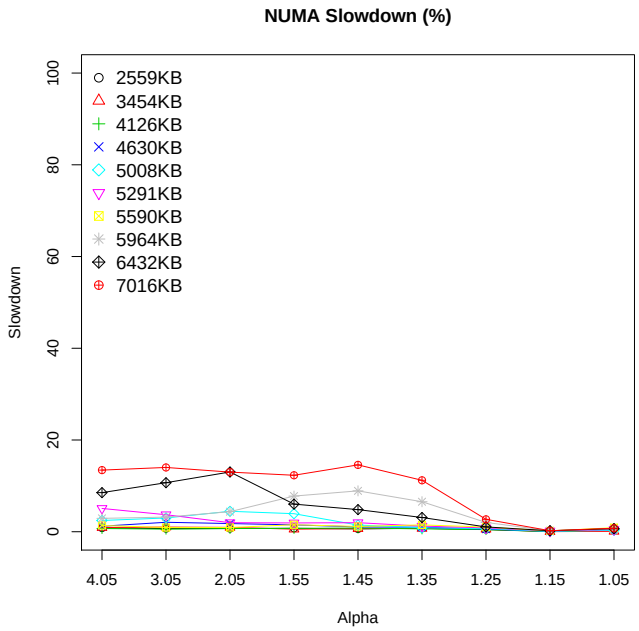

Fig. 14. The performance difference, measured in percentages, of local vs. remote Zipf random cache accesses using varying dataset sizes and Zipf alpha values.

As the α value became less than 1.25, differences in performance between local and remote access experiments became virtually negligible. Interestingly, this is contrary to the results seen for sequential and uniform random access patterns in Figures 9 and 11, where the largest slowdowns were seen at these large dataset sizes.

D. Access Pattern Comparisons

1) Sequential and Random Uniform: Comparing the absolute performance of sequential and uniform random access, in Figures 8 and 10, sequential accesses were faster than random for all dataset sizes. This could be due to prefetching performed by the processor. However, when comparing the slowdown of remote memory in Figures 9 and 11, the sequential pattern suffered a much greater slowdown than random.

2) Zipf and Others: The absolute performance of Zipf should not be compared with the results for sequential or uniform random because of the metric used for measuring performance, the number of loop iterations. Each loop in the Zipf experiments performs more memory accesses, due to both the uniform random backbone and the sequential traversal of the sequence based on the Zipf random value. Thus, it cannot be expected that the Zipf experiments should perform at the same levels as the others.

Comparing the slowdowns of remote memory in Figures 9, 11 and 14, Zipf suffered a much smaller decrease in performance when dataset sizes approached the size of L3 cache. This was especially apparent when α values of 1.25 or less were used, where running with remote memory resulted in almost no performance degradation.

VI. Conclusion

The NUMA architecture was introduced to alleviate the memory contention issue of the SMP architecture. However, NUMA has its own challenges including cache contention, which is the result of scheduling many memory bounded processes on the same socket. Therefore, it is necessary to schedule some of the processes on a remote socket, which will result in experiencing a remote NUMA penalty. The NUMA penalty is the delay which is due to accessing memory of a remote socket through

interconnects. It is clear that the NUMA penalty hurts application performance but this effect has not been studied and quantified before. In this project, we measured the application performance slowdown due to the NUMA penalty with several microbenchmarks using the Cachetest tool. In order to mimic real applications' memory access patterns, we considered three different memory access patterns including sequential, uniform random, and Zipfian access pattern. The results of the experiments indicated that, generally, when the size of the dataset is greater than the last level cache, the NUMA penalty becomes highly significant. Moreover, in the Zipf case, when the value of α, which indicates how scattered the samples are, decreases and samples become more scattered, the performance decreases and the NUMA penalty becomes significant. Conversely, when the α increases, the samples become more concentrated and repetitive resulting in less slowdown.

VII. Future Works

Although we believe a Zipfian distribution is more representative of real workloads' memory access patterns, compared to sequential and random distributions, it is still required and would be interesting to study this problem and confirm which distribution is the most representative distribution of real workloads' memory access patterns. We plan to study this issue as a future work and determine memory access patterns of different workloads of the SPEC CPU benchmark suites.

Acknowledgment

The authors would like to thank Dr. Martin Karsten for his advisement and support in performing this study.

References

[1] F. Auerbach, "Das gesetz der bevölkerungskonzentration," *Petermann's Geographische Mitteilungen*, vol. 59, pp. 74–76, 1913.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 340–351, IEEE, 2005.

[3] K. J. Christensen, "Program to generate zipf (power law) distributed random variables," 11 2003.

[4] A. Kleen, "A numa api for linux," tech. rep., Novell, Inc., 04 2005.

[5] I. Kotera, R. Egawa, H. Takizawa, and H. Kobayashi, "Modeling of cache access behavior based on zipf's law," in *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*, pp. 9–15, ACM, 2008.

[6] Z. Majo and T. R. Gross, "Memory management in numa multicore systems: trapped between cache contention and interconnect overhead," in *ACM SIGPLAN Notices*, vol. 46, pp. 11–20, ACM, 2011.

[7] M. E. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary physics*, vol. 46, no. 5, pp. 323–351, 2005.

[8] A. Szlavik, "Cache-aware virtual page management," 2013.