Selective Data Migration between Locality Groups in NUMA Systems

Junsung Yook[0000-0001-8068-4884] and
Bernhard Egger✉[0000-0002-6645-6161]

Seoul National University, Republic of Korea
{junsung,bernhard}@csap.snu.ac.kr

Abstract. Non-uniform memory access (NUMA) architectures exhibit variable memory access latencies that depend on the issuing core and the accessed memory location. To minimize an application's memory access time, the accessed data should be kept as close to the computation as possible. An promising strategy is to deploy groups of threads that access the same data on neighboring cores and close to the accessed data. This not only minimizes remote memory accesses latency but also reduces the amount of accessed cache lines and the traffic incurred by the cache coherence coherence protocol; however, finding and maintaining a good thread group allocation is difficult. This paper presents a novel at-runtime technique that improves application performance through better data locality without prior profiling runs. The presented technique accurately detects accessed memory sections through low-overhead sampling. Sections that are frequently accessed on a remote node are migrated to the local memory node. Migration of unused data such as data streams is avoided by only copying sections that are expected to yield a positive net gain.

Keywords: NUMA(Non-Uniform Memory Access) Architecture, Data Locality Groups, Performance Monitoring Unit

1 Introduction

The memory systems of computer systems are built around the fact that most programs exhibit temporal and spatial data locality. A hierarchy of memories from small and fast to large and slow provides low access latencies and high throughput by exploiting data locality. To achieve maximum performance, a process' working set should be kept in nearby memories. In parallel applications, several threads form locality groups when they access similar data objects. To support the placement of locality groups, NUMA (Non-Uniform Memory Access) architectures provide information on the organization of the physical memory hierarchy [3]. By harnessing this information, it is possible to not only increase the efficiency of caches through improved data reuse but also to reduce the communication overhead in the interconnection network. In other words, achieving better locality increases cache utilization and reduces the memory access latency.

In NUMA architectures, the data access latency depends on the location of the core that requests the data and the physical location of the data. Unlike in hardware-managed caches, the placement of data pages in memory can be controlled in software; we can thus exploit data locality by placing the accessed data in memory local to the core.

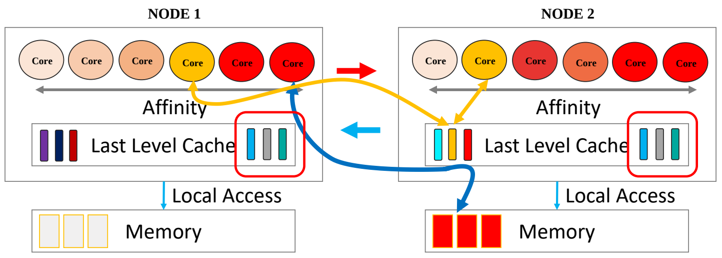

Fig. 1. Illustration of the advantages of local vs. remote memory accesses.

Several studies have proposed techniques to increase data locality to mitigate the cost of remote memory accesses. A common approach is to cluster the threads of an application according to their affinity of shared data [4]. To decide the optimal memory allocation policy, other techniques track memory allocation requests to identify memory objects before profiling the accesses to them [2]. Other approaches aim at preventing congestion in the interconnection network and at the memory controllers by balancing the load and decide the placement of data at runtime [1]. One of the main challenges is to avoid moving data that causes more overhead than benefit. An other difficulty for at-runtime techniques concerns the lack of exact information on memory accesses because they have to rely on sampling to keep the runtime overhead to a minimum. Finally, moving data into local nodes often obstructs load balancing efforts; data should thus be placed in consideration the utilization of memory nodes. All these challenges render low-overhead at-runtime data placement and coordination difficult.

In this paper, we present a technique that can exploit data locality at runtime and with low overhead. A history table is used to identify and migrate only data sections that are expected to yield a net benefit. We employ a technique that is able to infer used or likely-to-be-accessed-next sections of memory objects with a small number of samples and without the need to trace calls to specific routines. Finally, we introduce a placement technique that considers both locality and congestion.

An evaluation on a 72-core Intel NUMA system with 25 real-world programs from three benchmark suites shows that, on average, our migration policy is able to reduce the traffic caused by page movements by 70%. By restoring missing sections of memory objects, it was possible to achieve placements of steady state where threads and data are consistently settled earlier. As a result, the proposed techniques increases application performance by up to 25% and 10% on average over all evaluated parallel programs.

2 Design

The presented technique focuses on (1) relocating processes groups close to their data and (2) balancing the load of requests to the components of the memory subsystem. The thread and data placement is re-evaluated periodically every 100ms. Memory request events such as the number of L2 cache misses are sampled using the Precise Event-Based Sampling (PEBS) capabilities of the hardware performance monitoring unit.

2.1 Inference of Memory Objects

Sampling every n^{th} cache miss does not reveal the full memory access pattern of a group of threads and, in turn, lead to the identification of several small memory segments that are actually part of a single, larger object. To compensate for missing samples, the presented technique speculatively fills gaps between identified memory segments to capture the entire memory object. The heuristic is based on the idea that the intervals between the addresses of the same memory objects tend to be smaller than those to different memory objects.

The average stride between sampled addresses on the heap segment is obtained by dividing the distance between the largest and smallest address by the number of sampled addresses. If all the intervals between addresses in a sequence of sampled addresses are smaller than the average stride, then the entire range is considered a contiguous region of a memory object. To reduce the computational overhead, the minimal detection unit of a memory region is identical to the size of a physical page or larger.

2.2 Selection of Active Data for Migration

Chunks obtained by inference for sections of memory objects are candidate that can be chosen as active data. In this context, selected chunks represent the working set that are likely to be accessed again in the future, that is active data but not dead data such as data stream. The history table constituted by hash table preserves the previous access history of chunks identified by logical addresses. Through this table, the history of existence of the chunks at each epochs is examined. If the chunk has been accessed previously, the chunk is selected as candidate of migration and a counter which indicates how many times the corresponding chunk have existed is incremented. Or if it is the first time to access, the chunk is first considered as inactive data and it is registered in the history table. Also, chunks that are not accessed within a certain number of epochs are evicted from the history table treating as aged. In addition, the higher the number of times a chunk have existed, which indicated by a corresponding counter, the higher priority is assigned to move it first.

3 Evaluation

We evaluate the presented technique on a 72-core (4-node) Intel Xeon E7-8870 v3 processor with 512 GiB of DRAM. All experiments were performed with NUMA-balancing turned off in the Linux kernel to minimize interference between our technique and the operating system's NUMA balancer. We evaluate 25 benchmarks from the NPB, Parsec and Rodinia benchmark suites.

The results are shown in Figure 2. The execution time of all benchmarks is normalized to standard execution on Linux. We plot the runtime for data migration with (`Selective`) and without (`Unselective`) the heuristic to identify memory objects. On average, the presented technique achieves a 10% shorter execution time with individual benchmarks running up to 25% faster. Also noteworthy is the fact that the maximum slowdown for applications that do not profit from the presented technique is consistently below 10% and that all overhead of the presented technique is included in the presented results.

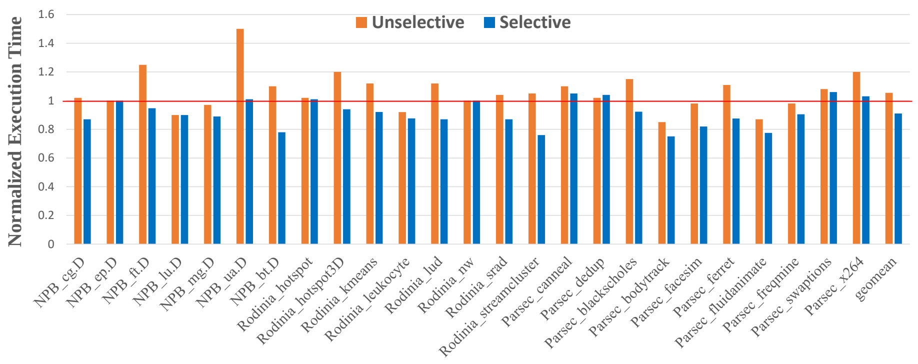

Fig. 2. Comparison of performance with proposed techniques.

4 Conclusion

This paper has presented a low-overhead, at-runtime approach to data migration for higher performance in NUMA architecture. Objects to be migrated are identified through low-overhead sampling, single, small regions extended into regions through a heuristic. By selecting data objects to be moved based on the likelihood of their reuse, redundant migrations can be avoided. Finally, since localization can destroy the load balance of the memory system and cause congestion that results in performance degradation, our technique balances data across multiple memory nodes.

Experiments with 25 parallel applications show an average performance improvement of 10% and a reduction of the number of data migrations between memory nodes by 70%. For several applications, a 20% or higher performance improvement is obtained, demonstrating that low-overhead data placement techniques are an effective way for significant performance gains on existing hardware.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback and suggestions. This work was funded, in parts, by the Korean National Research Foundation through grants 2022R1F1A1074967 and 21A20151113068 (BK21 Plus for Pioneers in Innovative Computing - Dept. of Computer Science & Engineering, SNU). ICT at Seoul National University provided research facilities for this study. B. Egger is the corresponding author.

References

1. Mohammad Dashti, Alexandra Fedorova, and Justin Funston. Traffic management: A holistic approach to memory placement on numa systems. 2012.
2. Renaud Lachaize, Baptiste Lepers, and Vivien Quema. Memprof: A memory profiler for NUMA multicore systems. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 53–64, Boston, MA, June 2012. USENIX Association.
3. ORACLE. Memory and thread placement optimization developer's guide. 2012.
4. David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. 2007.