Understanding the Behavior of Pthread Applications on Non-Uniform Cache Architectures

Gagandeep S. Sachdev, Kshitij Sudan, Mary W. Hall, Rajeev Balasubramonian
School of Computing, University of Utah

Abstract

Future scalable multi-core chips are expected to implement a shared last-level cache (LLC) with banks distributed on chip, forcing a core to incur non-uniform access latencies to each bank. Consequently, high performance and energy efficiency depend on whether a thread's data is placed in local or nearby banks. Using compiler and programmer support, we aim to find an alternative solution to existing high-overhead designs. In this paper, we take existing parallel programs written in Pthreads, and show the performance gap between current static mapping schemes, costly migration schemes and idealized static and dynamic best-case scenarios.

***Keywords*-Data locality; static-NUCA; Performance;**

I. Introduction

Future high-performance processors are expected to accommodate many cores sharing a last-level cache (LLC), partitioned into many banks and distributed on chip. Every core will therefore be in close proximity to a subset of LLC banks, while access to other banks will require long-distance traversal over the on-chip network. This leads to a *non-uniform cache access (NUCA)* architecture [1]. The overall performance and energy-efficiency of an application depends on the underlying system's ability to keep a core's relevant data in nearby banks.

Because of the communication bottlenecks, many studies have attempted to improve locality in NUCA caches, primarily with hardware-based techniques. Since dynamic-NUCA [1] is evident to have significant complexity, more recent studies have adopted a simpler static-NUCA architecture [1] that employs a unique mapping between a block and a cache bank. To improve locality in these architectures, OS-based page coloring can be used to control a page's address and hence the bank that it maps to. Most recent papers [2] primarily rely on first-touch page coloring to guide a page towards the core that is likely to be its dominant accessor. If a page is initially mapped to a sub-optimal bank, it must be moved with migratory techniques. Such migration is expensive and complex as it requires cache flushes, TLB updates, and in some cases, DRAM page copies.

This work was supported in parts by NSF grants CCF-0811249, CCF-0916436, SHF-1018881, NSF CAREER award CCF-0545959, HP, Intel, SRC grant 1847.001, and the University of Utah.

Ultimately, for widespread commercial adoption, the problem of data locality in LLCs must be solved with techniques that are simple and yet flexible. We believe that the use of programmer or compiler hints, based on an understanding of the application's data structures, can guide the OS page coloring process. This helps correct the many incorrect initial page assignments and eliminates the need for complex migratory mechanisms. In this paper, we characterize the programs based on their performance on a static placement versus a migration scheme followed by a discussion on the access patterns of data and the stratergies to reduce remote cache accesses.

II. Experimental Characterization

The simulation infrastructure uses Virtutech's Simics platform [3]. We model a 16-core CMP running linux. Each core is an in-order x86 processor and has private 16 KB L1 caches. The LLC is a shared 16 MB L2 cache divided into 16 banks with a 4×4 2D mesh on-chip network. Each core is attached to private L1 caches and one bank of shared L2 cache, and the on-chip network router. PARSEC [4] applications with *Simlarge* input are measured over the *Region of Interest (ROI)*.

Figure 1 shows the performance impact of optimally placing data in the correct last-level cache (LLC) bank. Results for ten PARSEC benchmarks are shown. *Blind* relies on the default linux kernel to allocate physical pages. *First Touch* binds the page mapping to the core that makes the first access to the page. For *Migration*, we combine first-touch allocation with migrating a fixed number (ten) of highly accessed pages every epoch (10 million cycles) to the core accessing it the most. For Oracle, we perform a two-pass simulation where the first pass determines the number of total accesses to each page from each core, and the second pass measures performance by using information from the first run to guide its page placement. In the first version (*Oracle_High*), a page is placed in the bank that yields the most local hits while in the second one (*Oracle_COG*), it is placed at the center of gravity of the sharers. *Ideal* ignores placement and assumes local bank access times for every LLC access to provide an idealized upper bound on performance.

The performance of First Touch is not much higher than Blind, on average only 2%. On the other hand, Oracle and

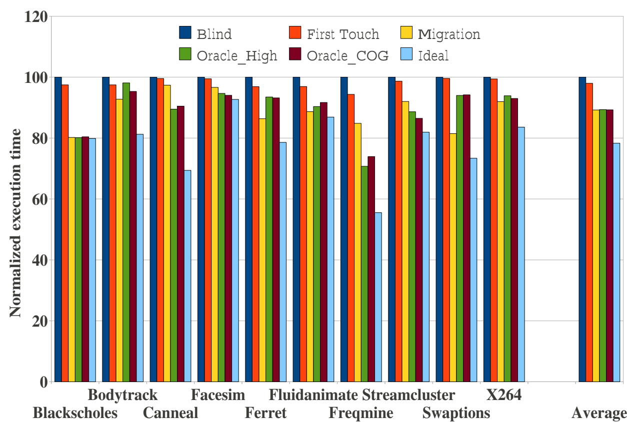

Figure 1. Execution time for programs on a NUCA CMP with the allocation schemes defined in Section II.

Ideal yield much higher performance than First Touch, on average 12% and 27%, respectively. Migration can improve upon First Touch by 9.8%. However, migration schemes carry with them a performance penalty and the energy demands of moving data across the chip. These overheads cause a 1.8% performance degradation when compared to a hypothetical migration scheme with no penalties from DRAM page copies and increased miss rate. In future systems, the overheads are expected to increase and thus, we believe that it is not a scalable solution. We have shown that First Touch is inadequate, there is significant room for improvement, and Migration has high overheads. The next section examines these results in more detail, and discusses some common patterns in the codes that suggest the potential for improvement with compiler or programmer hints to the operating system, thus avoiding the need for migration.

III. Discussion

Looking at these results we observe several classifications of the codes.

A. The programs where both Oracle and Migration perform close to Ideal (`blackscholes`) signify that a good static mapping would achieve the same performance as migration. The threads in this program work on independent partitions of a large data structure allocated in main. If the application can convey this information to the OS, it can allocate the pages in the optimal banks. Intuitively, we would expect First Touch to work well too, but because the application allocates and initializes a large global array in the initial thread, First Touch maps all pages to an LLC bank closest to the main thread.

B. The programs where the Oracle static mapping is much better than Migration (`canneal`, `freqmine`, `streamcluster`) suggests that migration is not beneficial, or there is a better static mapping. For `canneal`, a thread can work on any element in the large data structures suggesting uniform sharing. However, in `streamcluster`, a thread primarily works on its partition of the global data structures but also makes accesses to other partitions. Migration will keep migrating pages unnecessarily based on access patterns in an epoch. Again, if the OS knows the sharing pattern and partition sizes, it will map the pages to their dominant accessor.

C. The programs where Migration is better than the Oracle static mappings (`bodytrack`, `ferret`, `fluidanimate`, `swaptions`, `X264`) suggest that static mapping is sub-optimal as written. This can be due to task parallelism without affinity or excessive false sharing at page granularity. Two of the programs, `bodytrack` and `ferret` use dynamic scheduling of their threads. In both cases, a thread accesses a global queue to get the next unit of work, without taking affinity into account. As written, the thread which accesses a page of data is determined at run time, making a static mapping unsuitable. Performance could be improved if the system, rather than using a first-in, first-out mapping of work to threads, would instead consider affinity in both allocation and scheduling. When pages allocated dynamically through *malloc* are given to different threads during execution (such as in `swaptions`), a static mapping leads to significant false sharing of the page. Using a thread-private heap whenever safe, we can achieve a much better static mapping that eliminates this false sharing.

We intend to pursue the following optimizations to improve the performance of the code using a software-only strategy: (1) using thread-private storage when safe for stack and local, dynamically-allocated data; (2) partitioning global shared objects into separate pages according to their affinity; (3) explicitly marking migratory data; and, (4) affinity scheduling of parallel tasks.

IV. Conclusion

In this paper, we have performed a study to prescribe a set of optimizations for Pthreads benchmarks targeting NUCA architectures. We consider how architecture, operating system, compiler and application developer can work together to yield significant performance gains while reducing overall system complexity. As the complexity of memory hierarchies grows in future architectures, optimizations such as these will become increasingly critical to overall performance.

References

[1] C. Kim, D. Burger, and S. Keckler, “An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches,” in *Proceedings of ASPLOS*, 2002.

[2] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, “Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches,” in *Proceedings of HPCA*, 2009.

[3] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A Full System Simulation Platform,” *IEEE Computer*, vol. 35(2), pp. 50–58, February 2002.

[4] C. Bienia, S. Kumar, J. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” Department of Computer Science, Princeton University, Tech. Rep., 2008.