SOLVING THE MYSTERY OF TRANSFERRED OF CACHED DATA IN A MULTIPROCESSOR, OLTP ENVIRONMENT

Brittany Kwait
Fordham University
kwait [at] fordham.edu

Dr. Kelly Shaw
Department of Mathematics & Computer Science
University of Richmond
kshaw [at] richmond.edu

Abstract

Commercial databases run on multiprocessor systems exhibit particularly high levels of false data sharing resulting from the low cache coherency present in such systems. These databases suffer from cache-to-cache data transfers. Until now, research has only focused on hiding or accelerating these cache transfers, not reducing or eliminating them. Yet, to do this it must first be understood why and how the data is transferred.

I. Introduction/Motivation

In the realm of database implementation and usage, large, commercial workloads (OLTP) frequently serve as a company's foundation more than bricks and mortar ever could. Regardless of purpose, many businesses, such as online bookstores and airline reservationists, operate on the premise of constantly retrieving, adding, and modifying enormous amounts of data.

By its very nature, OLTP demands fast, concurrent access to lots of data. For this to be achieved, programs are run on shared-memory multiprocessor systems. Yet, beyond all the perks of such a system comes one debilitating problem: cache coherency.

Cache coherency results when the processors have learned to successfully "communicate"; that is, if one processor is writing to a data block, some internal memory procedure precludes the threads from other processors from either reading or writing to that same block. In reality, multiprocessing systems are notorious for their high levels of false sharing (in this case, the false sharing stems from cache-to-cache transfers). This occurs when multiple threads attempt to access the same block of data. Because OLTP programs exhibit a great amount of thread parallelism, computer architects have recently been focusing on how these applications performed in the shared-memory environment. In summary, the research has shown is that these systems are extraordinarily inefficient. In fact, commercial applications spent 30 -35% of overall execution time on coherent read stalls, which are more prevalent as more users try to access data[2].

It is our aim to locate the source of these transfers by observing when they are occurring and then mapping these instances back to a invocation in the program.

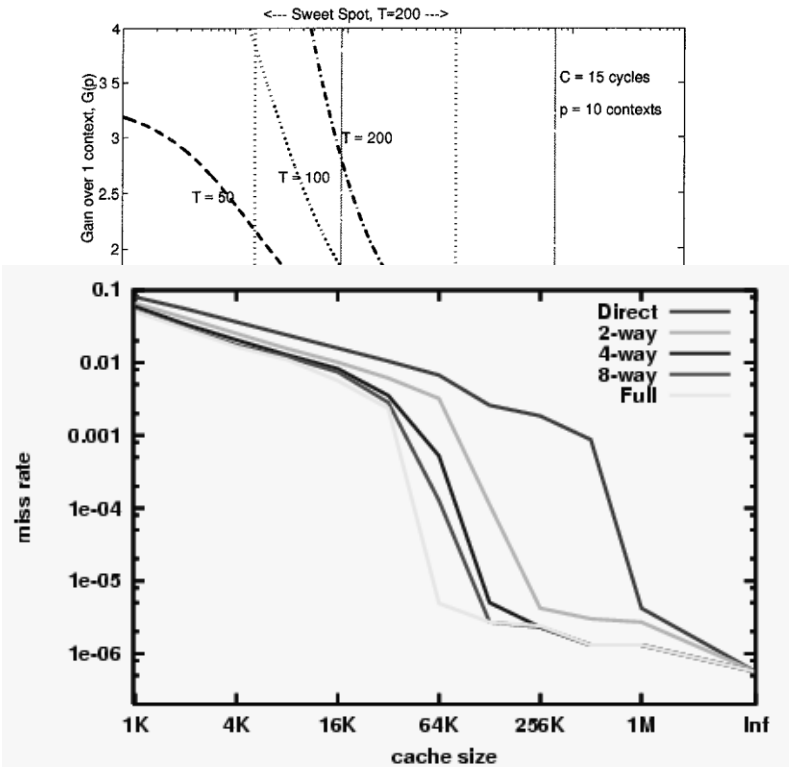
II. Background

Over the past decade, much research has been done to improve the efficiency of

databases, yet very little of this has targeted, or is even applicable to, online transaction processing environments. Despite its growing importance, research has been stymied because of the immense size, complexity, mutability, and primarily proprietary software licensing characteristic of OLTP databases[1]. Furthermore, most of the work available is somewhat dated—by computer industry standards; most of the papers being published in the late 1990s.

Due to database schema and the mystery of OLTP data transfers, the bit of previous work done has been polarized. Software scientists attempt racing the issue of cache-to-cache data transfers by raising the degree of multithreading. A multithreaded microprocessor is good for "hiding" latency misses, but the quantity of threads considered most advantageous is disputed, as their effects eventually level off [Figure 1, 3, 4]. The counter camp has tried improving OLTP performance by improving the hardware. Considered methods include multiprocessor architecture and expanding cache size. Yet these tactics are approaching their limits as well, and subsequent research has suggested a bigger cache may only cause bigger problems [Figure 2, 1, 5]. It is our understanding that discovering what is causing the transferring of cached data requires an analysis of the stratum linking hardware and software, as will its (hopefully) eventual elimination.

What data is chosen to actually be stored on the cache must follow at least one of two rules. First is what is known as spatial locality: if a piece of data is used, it is likely that other data nearby will also be used. The second is temporal locality: if a piece of data is used once, it is likely to be used again. Interestingly, of the coherency-camouflage methods previously described, the prefetching method utilizes spatial locality while the snooping method relies on temporal locality. "Multiple instruction issue and out-of-order execution provide only small gains for workloads such as OLTP due to the data-dependent nature of the computation and the lack of instruction-level parallelism." [4] For this reason, and other gleaned from other research, it was decided that temporal locality was much more relevant for OLPT environments and, consequently, our research. Moreover, one way to gauge the project's success would be to evaluate the levels of spatial locality present in the cache. If the data is able to



properly be stored near data that is commonly used with it, then the relevance of spatial locality will as a memory reference will become moot and threads will be able to execute their computations at chip level.

Due to database schema and the mystery of OLTP data transfers, the bit of previous work done has been polarized. Software

scientists attempted to race the issue of cache-to-cache data transfers by raising the degree of multithreading. A multithreaded microprocessor is good for "hiding" latency misses, but the quantity of threads considered most advantageous is disputed, as their effects eventually level off [3, 4]. The counter camp has tried improving OLTP performance by improving the hardware. Considered methods include multiprocessor architecture and expanding cache size. Yet these tactics are approaching their limits as well, and subsequent research has suggested a bigger cache may only cause bigger problems[1, 5]. It is our understanding that discovering what is *causing* the transferring of cached data requires an analysis of the stratum linking hardware and software, as will its (hopefully) eventual elimination.

III. Methodology

III.A. Summary

To best locate the program code instigating the transfer of cached data, the use of open-source software would be imperative. Our research plan entailed investigating how and what OLTP environments were implemented in previous papers and then modeling our OLTP database to adhere to these specifications.

Using the same method as above, a standard query set would be developed and executed N times. A trace file would be produced during runtime. Through the use of a specially-developed program, the trace file will be analyzed to find where data transfers were happening, computer performance statistics, and, perhaps, perform some rudimentary data mining (see Section V, "Future Work").

III.B. Specifications

To track program threads and memory accesses, it is crucial for all of our software to open-source. Our platform was the latest version of the Linux kernel. Incidentally, for OLTP "operating-system activity is non-negligible, but it also does not dominate the memory system behavior"[1]. The database used was PostgreSQL 8.1, an open-source database with the commercial workload to be generated based on the Transaction Processing Council's Benchmark C (TPC-C).

The program written split the trace of the running application into two pivotal pieces: threads and addresses. Included in each line of the trace file is a thread ID, program counter (hexadecimal), destination address (hexadecimal), thread size (bytes), and whether the instruction is a read or a write. Statistics compiled include: examining the frequency of address, thread, and instruction invocations, thread lifetime, and first-access writes, etc . Understanding first-access writes is crucial since these will always result in false sharing, by definition. Since this project is merely an auxiliary job in a larger body of research, a number of statistics that may not seem pertinent to the task were implemented that may later prove useful.

III.C. TPC-C, in brief

There is some animosity TPC-C. Some say the benchmark is outdated and does not meet the high-volume data and traffic needs of today's commercial world. Regardless, the aim was to approximate how a real OLTP environment would operate, not to measure system performance. The TPC-C acts as a warehouse of a wholesale distributer of 100,000 unique items with queries designed to mimic the order-entry queries that would be used in any business environment that manages and sells a product. TPC-C includes the following mix

of five randomized, synchronized database transactions: 43% new 10-item orders, 43% payments, 4.3% deliveries, 4.3% stock level checks, and 4.3% order-status checks. Within each of these transactions are even more stipulations, making the benchmark thorough and all-encompassing, perfectly adequate for a complete trace file.

IV. Implementation/Dilemmas

Trouble began with what was originally planned as an intermediary step—dragging a three-day task past three months. The TPC-C standards need to be created, requiring the use of either a database generation/execution program or the coding of such a program, which time would not allow. Finding software that could run in a Linux environment and be synchronized with our PostgreSQL database was difficult. Eventually, a kit released by Open-Source Development Labs, Inc. was found called Database Test Suite 2 (DBT2). Unfortunately, poor documentation with this program caused weeks of grief as we tried to figure out how it worked.

Additionally, it was learned that due to the vast numbers of instructions processed on a multiprocessor machine in just a small amount of time, an adequately-sized trace file would be too large to process. The trace file generation code and analysis program would need to be modified to allow execution lines to be buffered shortly and piped in.

V. Future Work

Foremost, a solid and comfortable grasp be attained on the operation and execution of DBT2 so the TPC-C benchmark may be met for the generation of a trace file. Also, pattern "mining" techniques for finding what data (memory addresses) are used together will be beneficial in the processing of this, and future research.

VI. References

[1] L. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998. http://portal.acm.org/citation.cfm?id=279363&dl=ACM&coll=ACM

[2] T. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal Streaming of Shared Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture,* June 2005. http://portal.acm.org/citation.cfm?id=1080695.1069989

[3] L. Barroso, K. Gharaschorloo, R. McNamara, A. Nowatzyk, S. Qadeer, *et al.*. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. *ACM SIGARCH Computer Architecture News*, May 2000. http://portal.acm.org/citation.cfm?doid=342001.339696

[4] J. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. 1998. http://portal.acm.org/citation.cfm?id=279367&coll=portal&dl=ACM

[5] L. Barroso, K Gharachorloo, A Nowatzyk, B Verghese. Impact of Chip-Level Integration on Performance of OLTP Workloads. In Proceedings of the 6th International Symposium on High Performance Computer Architecture, January 2000. http://www.ri.cmu.edu/pubs/pub_4936.html

[6] B.H. Lim, R. Bianchini. Limits on the Performance Benefits of Multithreading and Prefetching. *Sigmetrics* Vol 5, Issue 96, 1996. http://portal.acm.org/citation.cfm?id=233021&coll=portal&dl=ACM

[7] Wikipedia contributors, "CPU cache," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/w/index.php?title=CPU_cache&oldid=92419183

[8] M. Martin, D. Sorin, A. Ailamaki, A. Alameldeen, R. Dickson, *et. al*. Timestamp Snooping: An Approach for Extending SMPs. *ACM SIGPLAN Notices* Vol 35, Issue 11, 2000.

See Also

www.osdl.org

www.postgresql.org

www.tpc.org