Memory Matters

Tim Kaldewey[1,2], Andrea Di Blas[1,2], Jeff Hagen[2], Eric Sedlar[2], Scott Brandt[1]
[1]Computer Science & Engineering, University of California Santa Cruz
{kalt, andrea, scott}@soe.ucsc.edu
[2]Oracle Server Technologies – Special Projects
{tim.kaldewey, andrea.di.blas, jeff.hagen, eric.sedlar}@oracle.com

Abstract

When predicting application performance, allocating resources, and scheduling jobs, an accurate estimate of the required resources is essential. Although CPU and disk performance is relatively well understood, memory performance is often ignored or considered constant. Our research shows that memory bandwidth can vary by up to two orders of magnitude depending upon access pattern, read/write ratio, and number of cores accessing the memory. We believe that resource management can be improved by accounting for these factors, especially for data-intensive applications.

1 Introduction

Real-time resource management depends upon knowledge of the resources needed for each task. Significant research has been performed on WCET estimation, admission control, and scheduling with a strong focus on CPU performance requirements. Newer research has examined real-time resource management for networking [8] and disk I/O [9]. Relatively little research has focused on the impact of memory performance for three primary reasons: memory bandwidth is assumed to be constant; variations in memory bandwidth are thought to be small; and it has been assumed that real-time processing is generally not memory-bandwidth bound.

With the rapidly decreasing costs of memory and storage, applications manage increasingly large volumes of data and real-time and non-real-time processing naturally becomes more data-intensive. Rapidly growing main memory sizes eliminate disk I/O as a bottleneck even for traditionally data-intensive applications, *e.g.* databases. In-memory databases [3, 13] put main memory performance in the spotlight.

High-definition video processing, image processing, real-time data collection and analysis, and other applications similarly put an increasing burden on the memory.

Making matters worse, server virtualization, service level agreements, and other types of QoS now require many traditionally best-effort applications to be managed according to real-time principles. With processor performance increasing much faster than memory performance, memory has become the bottleneck for many data-intensive applications. This issue was already of concern to computer architects more than a decade ago: "It's the Memory, Stupid!" [12].

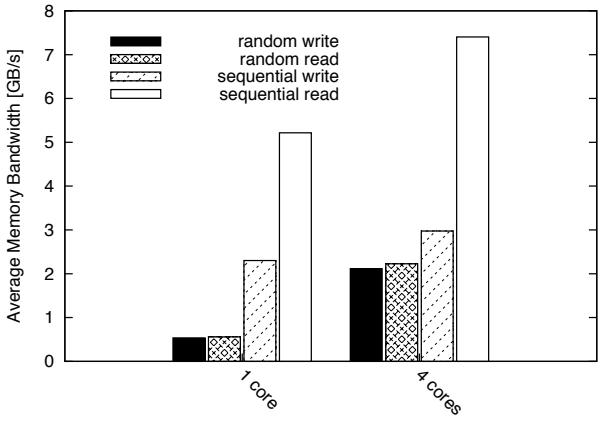


Figure 1: Measured memory bandwidth on a quad-core CPU for different access patterns.

We decided to investigate the relationship between application behavior and memory performance. We show the effects of data access pattern, read/write ratio, data types, and number of concurrently active cores/threads. To our surprise, every single one of these has an impact on memory performance, some in the opposite way than expected (Fig. 1).

While memory access time increases linearly with the amount of data accessed, we found that sequential access patterns yield an order of magnitude higher memory bandwidth than random accesses. The use of larger data types yields more than an order of magnitude speedup. In general, performance increases with the number of cores.

Overall, we found two orders of magnitude difference between best- and worst-case memory performance. We believe this must be accounted for in resource estimation, provisioning, allocation, and scheduling of data-intensive

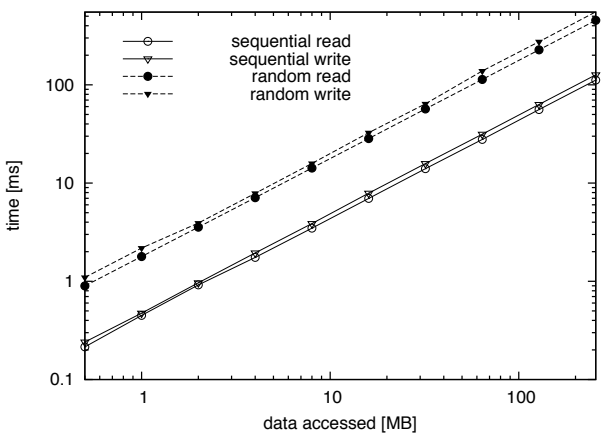


Figure 2: Execution times for accessing increasing amounts of memory using 8-bit data types. Other data types show the same behavior.

applications.

2 Dissecting Memory Performance

After establishing that access time is proportional to the amount of memory accessed, we evaluate memory bandwidth along three axes: access pattern, data type, and the number of cores and threads concurrently accessing the memory

Experimental setup. Experiments were conducted on an x86 system running Oracle Enterprise Linux 5.2 64-bit, Kernel 2.6.18. The hardware consisted of an Asus P5Q mainboard with an Intel P45 chipset, an Intel Core 2 Quad Q6700 CPU 2.66 GHz, and 2 GB of DDR2-1066 RAM with 5-5-5-15 latency. The system was dedicated to the experiments; no other applications or users were active.

Measurements were conducted in user space with the `gettimeofday()` function, to include all overhead a real application would incur when accessing memory. The C and Assembly code used to generate the desired memory access patterns was designed to minimize computational overhead: reads add the requested memory value to a register variable; writes write the sum of 2 registers to memory; (pseudo) random memory locations are generated using an inline function; and aggressive compiler optimizations turned on (`gcc -O4`).

Memory accesses are always aligned to the respective word size boundaries, *i.e.* all n-byte words are aligned to n-byte boundaries. In order to avoid caching effects for small data sets or paging to disk for large data sets, all experiments allocate exactly 512 MB of memory. When the amount of memory accessed exceeds 512 MB, we iterate over the memory. When using multiple threads, each thread is assigned its own memory to eliminate any potential caching effects due to locality of reference.

Amount of memory accessed. Our first set of experiments shows that the time required to access main memory is nearly linear in the amount of memory requested (Fig. 2). Using a single CPU core we measured the access time for requests ranging from 512 KB up to 32 GB with different access patterns. Figure 2 shows results for 8-bit character data types. Results were similar across data types, *e.g.* 32-bit, 64-bit and 128-bit. Therefore, we use bandwidth as the metric for the remaining experiments.

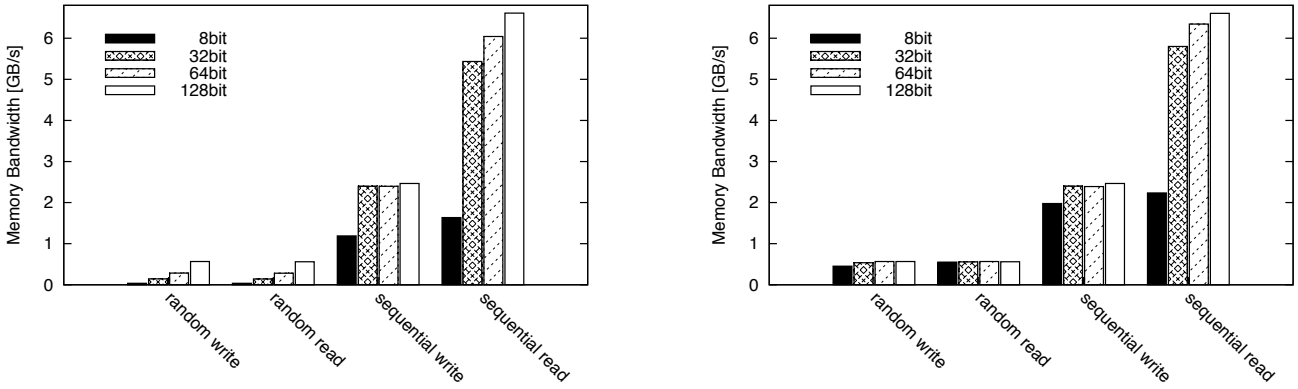
Data Types. The size of data types accessed has a major impact on memory performance. Figure 3a shows more than an order of magnitude difference in memory bandwidth between small and large data types for random access patterns, and 2–4×, for sequential access patterns.

Smaller data types require more memory operations to access the same amount of memory. Thus, for a given amount of memory, random accesses of smaller data types experience a higher degree of randomness than larger data types. As a result, random access bandwidth increases proportionally to data type sizes. For example, 128-bit random accesses receive twice the bandwidth of 64-bit ones. Sequential access patterns are less dependent on data types as sequential access patterns benefit from prefetching, omnipresent in caches. However, the 2–4× performance drop for 8-bit data types shows a limitation of the memory controller, *i.e.* the number of outstanding requests, which logically increases by 4× when comparing 32-bit and 8-bit data type accesses.

As we normalize the size of accesses to the largest data type, (*i.e.* comparing a 128-bit access with two sequential 64-bit, four sequential 32-bit, or sixteen sequential 8-bit accesses) the randomness is the same across data types. As one would expect, random access performance is now similar across data types (Fig. 3b). While the bandwidth of 8-bit sequential writes is now similar to the other data types, there remains a 3× performance difference for reads.

Access Pattern. Although memory is often referred to as Random Access Memory, memory access patterns, including read-write ratio, affect memory bandwidth. Figure 3b shows the effect of different memory access patterns, with the effects of different data types (mostly) eliminated by normalization as discussed above. The bandwidth for sequential accesses is up to one order of magnitude larger than random ones, *i.e.* 13× when reading from memory and up to 5× when writing to memory.

The 2.5× difference between sequential read and write bandwidth is surprising, as from the perspective of the physical DRAM memory module, read and write operations take approximately the same amount of time. To understand why writes take about twice as long as reads it is necessary to look at the way the cache is integrated into the memory subsystem.



(a) Individual data-type sized accesses

(b) Accesses normalized to 128-bit

Figure 3: Memory Bandwidth for different access patterns. Values are averaged from accessing amounts of 512 KB up to 32 GB.

All data transfers to and from memory will always transfer an amount of data equal to a *cache line*, 128 bytes in our system. When the CPU reads a word of any size, the whole 128-byte line containing the requested word will be read from memory into the cache. If the CPU then requests multiple words within the same line, as in a sequential access pattern, there is no need for additional memory transfers since the subsequent words are already in the cache.

This strategy, designed to exploit *spacial locality*, makes writes more complex. The CPU can only modify part of a cache line at a time, as it is impossible to write 128 bytes with a single machine instruction. But since transfers to main memory require writing a full 128-byte cache line, before modifying a word in a certain cache line it is necessary to first read that full line into the cache. There it can be modified in any part before being written back to memory. Therefore a write operation will in practice involve both a read and a write, approximately doubling the access time.

Concurrency All recently released CPUs comprise multiple cores, and multi-threading is omnipresent, so we investigated their effects on memory performance. As the increased number of memory requests due to small data types had a significant impact on memory bandwidth, one would expect similar results from multiple cores issuing requests in parallel. Multiple concurrent threads issuing sequential requests may together produce a random access pattern with correspondingly poor performance. Surprisingly, performance generally improves with multiple cores. Increasing the number of threads beyond the number of available cores had a negligible effect on memory performance.

For this experiment we measured the memory bandwidth for increasing numbers of threads accessing 32GB total, each thread operating on its private memory to eliminate caching effects caused by locality of reference across threads. Random accesses and sequential 8-bit reads benefit from multithreading, with bandwidth increasing proportionally to the number of active threads up to the number of available cores (Fig. 4). The performance gains for sequential writes using larger data types are small, as the bandwidth for a single core is already close to peak performance.

Multiple cores implies multiple memory controllers, each with its own queue for handling cache misses. Thus, we see bandwidth increase proportional to the number of cores where this was the bottleneck: random accesses and 8-bit sequential reads.

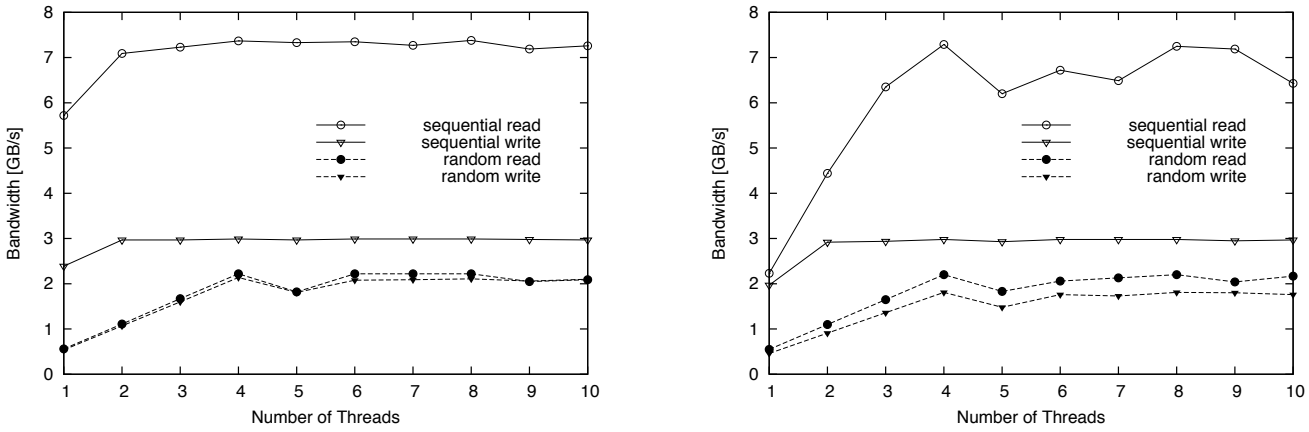
The peak bandwidth measured using 128-bit sequential accesses and all four cores was 7.5GB/s. Although the theoretical bandwidth of the memory in our system is approximately twice as much, we were unable to achieve higher performance despite trying several optimizations.

3 Related work

Computer architects have been trying to overcome the CPU-memory performance gap for decades. It was first identified as the "von Neumann bottleneck" in 1977 [2] which was ameliorated by incorporating caches and prefetching in newer CPU generations. With the ever-widening gap between CPU and memory performance, in the mid-90's the term "memory wall" was coined [14] which remains an accurate description of the situation today [7].

Within the database community, memory performance was identified as a bottleneck a decade ago [1]. Optimizations such as data structures aligned with cache lines [11] and using larger, native vector data types [15] have been proposed to increase memory performance. However, scheduling queries based on their predicted memory behavior to optimize memory performance and providing performance guarantees has not yet been attempted.

Relatively little real-time research has tackled the memory system. Marchand *et al.* [6] suggest tightly managing the available amount of memory in order to allow real-time performance guarantees with virtual memory. Fisher *et al.* suggest including cache space constraints in the scheduling decisions to avoid performance degradation due to thrashing [5]. Calandrino *et al.* provide heuristics to improve



(a) 32-bit data types (64-,128-bit are identical)

(b) 8-bit data type

Figure 4: Memory bandwidth with increasing number of threads. 32GB read total.

co-scheduling tasks competing for cache space in multicore CPUs [4]. Worst case execution time analysis has shown that it is possible to account for locality of reference [10]. However, in the context of large-scale multi-user applications this becomes difficult to manage.

4 Work in Progress

We are validating our results on other architectures, *e.g.* experiments on an AMD Quad-Core show similar behavior. As we were primarily investigating large datasets, 64× larger than the cache, we did not consider locality of reference and individual request latency. With increasing cache sizes, more applications will be able to fit their working set (partially) in the cache. Thus, we will conduct a similar analysis for cache memory.

We are evaluating options for incorporating our findings into a scheduler. One possibility is to make reservations in terms of CPU time adjusted for the expected memory performance. According to our results, accessing even small amounts of 8-bit randomly placed data, e.g 1 MB can require up to 30 ms, which is significant if sub-second accuracy is required.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proc. of 25th Int. Conf. on Very Large Data Bases (VLDB'99)*, pp. 266–277, 1999.

[2] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. of the ACM*, 21(8), pp. 613–641, 1978.

[3] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, and S. Haldar. Datablitz storage manager: main-memory database performance for critical applications. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'99)*, pp. 519–520, 1999.

[4] J. Calandrino and J. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Proc. of the Euromicro Conf. on Real-Time Systems (ECRTS'08)*, pp. 299–308, 2008.

[5] N. Fisher, J. Anderson, and S.Baruah. Task partitioning upon memory-constrained multiprocessors. In *Proc. of the 11th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pp. 416–421, 2005.

[6] A. Marchand, P. Balbastre, I. Ripoll, M. Masmano, and A. Crespo. Memory resource management for real-time systems. In *Proc. of the 19th Euromicro Conf. on Real-Time Systems (ECRTS'07)*, pp. 201–210, 2007.

[7] S. A. McKee. Reflections on the memory wall. In *Proc. of the 1st Conf. on Computing Frontiers (CF'04)*, p. 162, 2004.

[8] T. Okumura and D. Mossé. Virtualizing network I/O on end-host operating system: Operating system support for network control and resource protection. *IEEE Transactions on Computers*, 53(10), pp. 1303–1316, 2004.

[9] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with Fahrrad. In *Eurosys'08*, April 2008.

[10] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *Proc. of the 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS'05)*, pp. 148–157, 2005.

[11] J. Rao and K. A. Ross. Making B+trees cache conscious in main memory. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'00)*, pp. 475–486, 2000.

[12] R. Sites. It's the memory, stupid! *Microprocessor Report*, 10(10), pp. 2–3, 1996.

[13] T. T. Team. High-performance and scalability through application tier,in-memory data management. In *Proc. of 26th Int. Conf. on Very Large Data Bases (VLDB'00)*, pp. 677–680, 2000.

[14] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1), pp. 20–24, 1995.

[15] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'02)*, pp. 145–156, 2002.