Optimizing RAM-latency Dominated Applications

Yandong Mao Cody Cutler Robert Morris
MIT CSAIL

Abstract

Many apparently CPU-limited programs are actually bottlenecked by RAM fetch latency, often because they follow pointer chains in working sets that are bigger than the CPU's on-chip cache. For example, garbage collectors that identify live objects by tracing inter-object pointers can spend much of their time stalling due to RAM fetches.

We observe that for such workloads, programmers should view RAM much as they view disk. The two situations share not just high access latency, but also a common set of approaches to coping with that latency. Relatively general-purpose techniques such as batching, sorting, and "I/O" concurrency work to hide RAM latency much as they do for disk.

This paper studies several RAM-latency dominated programs and shows how we apply general-purpose approaches to hiding RAM latency. The evaluation shows that these optimizations improve performance by a factor up to 1.4×. Counter-intuitively, even though these programs are not limited by CPU cycles, we found that adding more cores can yield better performance.

1 Introduction

Profilers observing applications with little I/O typically show CPU utilization at 100% under peak workload. However, some of these applications are actually limited by RAM stalls. We call this class of applications "RAM-latency dominated." The stall occurs when the processor cannot continue execution until the RAM loads complete. As the processor is wasting CPU cycles spinning on the stall, it appears to the profiler as if the processor is executing instructions when it is actually waiting for RAM loads.

One type of RAM-latency dominated applications are those following long random pointer chains in working sets that are much bigger than on-chip caches. Random pointer following makes it difficult to predict the access pattern. As a result, each pointer dereference is likely to miss in the cache and requires fetching data from RAM rather than on-chip cache. RAM fetches on modern processor require hundreds of cycles, limiting the performance to tens of millions of pointer dereferences per second. This paper focuses on this type of application; dealing with applications with small working sets may require a different set of techniques [4, 6] than those presented here. The first contribution of this paper is a study of several applications of this type including a Java garbage collector and an in-memory key-value store.

To deal with the RAM-latency bottleneck, we compare it with disk and network latency. We found that the RAM latency problem is similar, and so are the corresponding optimization techniques. As a second contribution, we implement and evaluate three techniques, which improve the performance by a factor of 1.3× to 1.4×. Just like the OS disk scheduler obtains higher disk throughput by converting random access to sequential access, we can make memory access sequential to get higher throughput

from RAM too. Similar to the networked applications, we can also overlap computation and RAM latency by concurrent batching of unrelated operations. In analogy, we believe these techniques are universally applicable to this class of programs.

The rest paper describes and evaluates three techniques to address the RAM latency bottleneck. We run the experiments on an Intel Xeon X5690 processor, which has six physical cores with hyper-threading enabled. The processor is connected to a single RAM node of 48GB. The RAM node consists of three DRAM channels, each equipped with two DIMMs.

2 Linearizing memory accesses

Despite RAM being initially designed for random access, sequential access yields better performance on modern architectures. RAM provides optimal performance for random reads when RAM latency matches the speed of CPU. However, with today's technology, RAM latency is too high to drive the CPU busy. Similar observations have been made between disk and memory.

To fill the performance gap between the CPU and RAM, modern processors are equipped with hardware prefetchers to benefit sequential RAM access. The prefetcher speculatively issues RAM fetches during instruction execution, and reduces latency if the prefetched data is immediately requested by the CPU. Since hardware prefetchers *only* fetch if the recent memory accesses are sequential [13], sequential access could yield better performance than random access. For example, on a micro benchmark which keeps loading from RAM, sequential access provides 3.5× the throughput of random access on a single core. Profiling shows that, with sequential access, hardware prefetcher pro-actively fetches from RAM and reduces the cache misses by a factor of 4× compared to random access. In the rest of the section, we describe the random access pattern in a garbage collector, and how we convert it to sequential access to improve performance.

Garbage collectors are responsible for finding and reclaiming unused memory. The live data is discovered by "tracing", which follows all pointers starting at the roots (CPU registers, global variables, etc.) and follows every unvisited pointer in every object discovered. All objects not visited are unreachable and are therefore free memory. Because the live data (which can be thought of as a graph) is created and arbitrarily manipulated by the application, the addresses of the live objects have no correspondence to their position in the live data graph. This results in random memory accesses (a pattern that is unprefetchable as it is unpredictable) during tracing, incurring stalls on RAM accesses. Furthermore, each live object is read only once making caching useless.

Memory for long running programs will eventually become "fragmented", meaning that free memory is interspersed among live objects, making it impossible to satisfy some allocation requests. To remove fragmentation, moving garbage collectors will move the live data, compacting it together so the free memory becomes a contiguous block. Moving garbage collecting also provides an opportunity to improve the application's locality [3, 5]. Nevertheless, tracing compacted live objects still requires random memory access. To our knowledge, opportunistically moving live data to reduce garbage collection tracing times has not been explored. If the collector rearranges the live data such that the objects are in the same order that the tracing phase will read them, the new live data would be traced sequentially, providing an opportunity for the pefetcher to help, eliminating RAM stalls and improving tracing times.

We examine the impact of moving the live data into tracing order on garbage collection times with HotSpot, a Java virtual machine, in OpenJDK7u6[1] [2] and a small benchmark we wrote. The benchmark builds a Red-Black tree with 10 million nodes, performs 10 million inserts or updates with random keys, and then calls the garbage collector. Time spent tracing in the final garbage collection is recorded. The total live data used by this benchmark in HotSpot is approximately 1.4 GB. Since we are

[1] revision 3484:7566374c3c89 from Aug 13, 2012

interested in measuring tracing time only, all collections are single-threaded. The garbage collector is called several times during execution of the benchmark and the final tracing of live data takes 4.16 seconds on average. Next, we run our benchmark in HotSpot using only a modified semi-space copying collector which copies objects into tracing order. The final live data trace takes only 2.93 seconds on average after ordering, yielding a 1.4× speedup.

3 Alternating RAM fetch and computation

One memory layout benefits only certain memory access patterns. Some applications have more diverse access patterns and it is impossible to layout the memory to linearize all memory accesses. For example, looking up different keys within an ordered tree may follow different paths. No layout of the tree could linearize the memory accesses of all tree lookups.

Similar constraints apply to I/O subsystems such as the disk and network. Despite programmers' attempts to avoid disk seeks, sometimes seeks are unavoidable. For example, lookups of different files require reading dependent and non-contiguous disk blocks of directory nodes, each read of which is likely to cause a disk seek. While there is not much to do to reduce the per-lookup latency, we can overlap the computation with access latency through batching and asynchronous I/O, which results in higher throughput. This section describes how we adopt similar approach to address RAM latency using Masstree as an example.

Masstree [8] is a high performing key-value store for multi-core. Each core has a dedicated thread processing get/put/scan requests one by one. Masstree stores all key-value pairs in a single in-memory B+tree variant, which is shared by all cores. Masstree scales well on multi-core processors through optimistic concurrency control. We discuss concurrency on multiple cores in next section, and consider Masstree on a single core in this section.

Despite its careful design to reduce RAM references, the performance of Masstree is still limited by RAM latency. Masstree avoids RAM references by storing key fragments and children pointers within the tree nodes. This is possible by using a trie of B+trees such that each B+tree is responsible for eight bytes of the key. Such design avoids additional RAM dereferences for key comparisons. However, since Masstree has to lookup the targeted key for each request, RAM latency still dominates the performance. Each key lookup follows random pointers to B+tree nodes from root to a leaf, which is likely to miss in the cache and causes RAM fetches. This limits the per-core throughput to tens of millions requests per second.

To get higher throughput, we modify Masstree to process requests in batches and interleavely on *a single core* (we assume requests arrive in batches, which is reasonable in practical systems [9]). We call this version Interleaved Masstree, which descends tree lookups level by level for each batch of requests. At each level, Interleaved Masstree finds the children node to follow for each request, and then issue a prefetch of the children.

This approach allows Interleaved Masstree to descend one level of many lookups in at most one RAM latency, whereas non-interleaved Masstree descends a level for only one lookup in a RAM latency. By the time Interleaved Masstree inspects a tree node to look for the child, the node may have already arrived at cache due the prefetch issued when inspecting the parent. The technique is only effective for a degree of concurrency equal to the number of RAM banks [12]. For our workloads, we find a batch of eight requests achieves the best performance. On a read only workload, this technique improves the single-core throughput by a factor of 1.3×.

4 Parallelization

It turns out that, even for applications that are limited by random-access RAM latency and not by CPU cycles, running applications in parallel on multiple cores can help performance. The reason is that

the multiple cores can keep multiple RAM operations in flight in parallel, and thus can keep RAM busy. This is similar to the way how web servers address latency of network communication to the backend database server to get higher throughput, i.e. they use multiple processes to keep the backend busy.

At a low level, parallelization is identical to the interleaving technique described in Section 3. Both improve performance by exploiting the parallelism of the memory subsystem. The maximum parallelism is decided by the number of memory channels and RAM banks per channel [12]. Modern implementation allows independent access to each memory channel and multiple RAM banks within each channel [1]. The difference between parallelization and the interleaving technique is that the latter generates concurrent RAM accesses from single-core, while parallelization generates them from multiple cores. Programmers may prefer parallelization because it may be easier to parallelize than interleaving within a single core.

The degree of parallelism is application and implementation dependent. Some applications [10] have little global shared state and are inherently parallel; some applications have more sharing and require more effort to parallelize [7, 8, 11]. Since multi-core synchronization techniques are out of the scope of this paper, the rest of this section considers applications that are easy to parallelize.

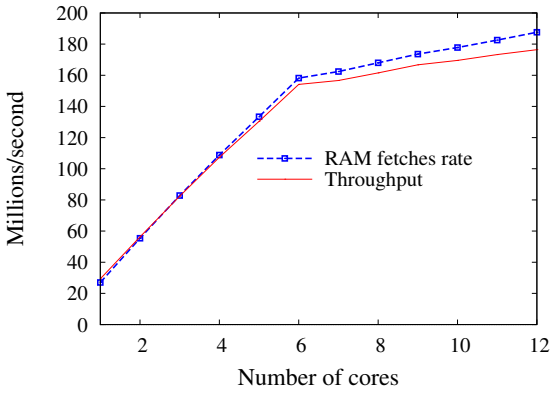

Figure 1: Throughput and RAM fetches rate varying the number of cores

To demonstrate that parallelization could improve performance via issuing more concurrent RAM fetches, we measure a memory intensive micro benchmark. Each operation of the benchmark reads eight bytes of a random cache line. Each core reads a different memory chunk of 100MB. The benchmark scales well since nothing is shared among cores. Figure 1 shows the result. The throughput (operations per second) increases thanks to the RAM fetches, which are in turn increased with the number of cores. The throughput increases less after six cores because each of the first six cores sits in a different physical core; beyond six, each additional core collocates with one of the physical cores due to hyperthreading.

5 Discussion

Our experience suggests that programmers should pay attention to both parallelization and cache-conciousness on multi-core architectures in order to achieve the best performance. While the first two techniques improve single core performance, parallelization is still required to achieve the best performance on a multi-core processor. On the other hand, parallelizing a cache-unconscious program may not yield the best performance. For example, loading memory randomly with all 12 cores can only achieve half of the maximum bandwidth.

One difficulty that arises when optimizing these applications is the lack of tools. Currently, we use Linux *perf* to find the hot spot and manually inspect the code to see if there is a RAM fetch bottleneck. This can be misleading since it is hard to tell if the hot spot is busy doing computation using data from on-chip cache or stalling on RAM fetches. A better tool that identifies RAM stalls spent in each instruction could help find such bottlenecks faster and more accurately.

6 Conclusion

This paper identifies one class of applications that suffer from RAM latencies. We describe three tech-

niques to address the RAM-latency bottleneck. We implement and evaluate these techniques on several applications, and observed a significant performance improvement. Since these techniques are similar to those used to address I/O latency bottleneck, we believe they are generally applicable to this type of applications. We hope this work could inspire people to optimize their applications in a similar way.

References

[1] Multi-channel memory architecture. `http://en.wikipedia.org/wiki/Multi-channel_memory_architecture`.

[2] OpenJDK. `http://openjdk.java.net/`.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, New York, NY, USA, June 2004.

[4] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, Monte Verit, Switzerland, 2009.

[5] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the International Symposium on Memory Management*, 1998.

[6] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the ACM EuroSys Conference (EuroSys 2013)*, Prague, Czech Republic, April 2013.

[7] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th Usenix NSDI*, 2013.

[8] Y. Mao, E. Kohler, and R. T. Morris. Cache Cratfiness for Fast Multicore Key-Value Storage. In *Proceedings of the ACM Eurosys Conference (Eurosys 2012)*, Zurich, Switzerland, April 2012.

[9] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[10] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA)*, Feburary 2007.

[11] J. Ryan, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009.

[12] R. Scott, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. *ACM SIGARCH Computer Architecture News*, 28(2), 2000.

[13] M. Wall. Multi-core is here! But How Do You Resovle Data Bottlenecks in Native Code, 2007. `http://developer.amd.com/wordpress/media/2012/10/TLA408_Multi_Core_Mike_Wall.pdf`.