

MIT Open Access Articles

Optimizing RAM-latency dominated applications

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

Citation: Yandong Mao, Cody Cutler, and Robert Morris. 2013. Optimizing RAM-latency dominated applications. In Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys '13). ACM, New York, NY, USA, Article 12, 5 pages.

As Published: <http://dx.doi.org/10.1145/2500727.2500746>

Publisher: Association for Computing Machinery (ACM)

Persistent URL: <http://hdl.handle.net/1721.1/90488>

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike



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 much 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 hide RAM latency. The evaluation shows that these optimizations improve performance by a factor up to $1.4\times$. Counter-intuitively, even though these programs are not limited by CPU cycles, we found that adding more cores can yield better performance.

1 Introduction

This paper focuses on the performance of applications that follow pointer chains in data structures that are much bigger than the on-chip cache. These programs tend to be bottlenecked by RAM latency because their RAM references do not obey any predictable pattern. For example, if objects within a data structure are allocated with a general-purpose allocator, it is unlikely that algorithms

traversing the data will visit objects in sequential address order; this defeats hardware RAM prefetching. Software prefetching is also hard to use here, since addresses aren’t known in advance: they are only learned as the program traverses the data structure. A RAM fetch on a modern processor requires hundreds of cycles, limiting the performance to a few million pointer dereferences per second. We call this class of programs “RAM-latency dominated.” This paper presents a study of two RAM-latency dominated applications: a Java garbage collector and an in-memory key-value store.

In order to reduce the RAM-latency bottleneck for these applications, we present two general-purpose techniques. First, if the program traverses the pointer graph in a known pattern, one can re-arrange the nodes in memory so that the pattern induces sequential access to RAM; sequential access yields higher RAM performance than random. Second, if the program has many independent tasks that involve following pointer chains, it can interleave them, exploiting the pipelining and banking available in the memory system in order to complete multiple fetches per RAM latency. The interleaving can be implemented by merging the tasks’ instructions into a single stream, or by running different tasks on different cores or hyper-threads.

The paper describes implementations of these techniques for a Java garbage collector and an in-memory key-value store. It presents measurements on an Intel Xeon X5690 processor (six physical cores, each with two hyper-threads; three independent RAM channels, each with two DIMMs). On this hardware, the techniques improve performance by a factor of $1.3\times$ to $1.4\times$.

2 Hardware Background

The techniques introduced in this paper exploit concurrency, pipelining, and buffering in the CPU/RAM interface. This section explains these features.

First, CPUs have a number of ways to start fetching data from memory before it is needed, thus avoiding

stalls. If the compiler generates a load instruction well before the instruction that uses the result (or the hardware speculatively executes the load early), the load may complete before the result is needed. The hardware also prefetches cache lines if it spots a sequential or strided access pattern [4, 9]. Finally, the software can issue prefetch instructions if it knows addresses in advance of need; software prefetch can be useful if addresses are not sequential but are predictable. Section 3 shows an example in which sequential access achieves higher performance than random access. Section 4 shows an example that prefetches data for different tasks in parallel.

Second, the RAM system can perform parallel operations. A system typically has multiple independent RAM channels; each processor chip often has a few separate channels, and a multi-chip system has a few per chip. Different channels can perform memory operations in parallel [8]. If the program can issue concurrent fetches to different parts of memory (for example, by prefetching data for different tasks in parallel), it can exploit the RAM system’s parallelism.

The last relevant hardware feature is the row buffer within each RAM chip. Accessing a RAM row causes the entire row to be stored in the RAM chips’ row buffers. An entire row is typically 4096 bytes, much larger than a CPU cache line. Successive accesses to the same row are satisfied from the row buffer, and are two to five times faster than accessing a different row [8]. The row buffer is one reason that sequential access is faster than random access; simple benchmarks on our hardware show that sequential access provides $3.5\times$ the throughput of random access on a single core. Section 3 exploits this effect.

3 Linearizing memory accesses

In this section, we explore the feasibility and potential benefits of converting unpredictable, non-sequential memory accesses to sequential accesses in garbage collectors.

Garbage collectors are responsible for finding and reclaiming unused memory. The live data is discovered by “tracing”, the process of following all pointers starting at the roots (CPU registers, global variables, etc.) and every 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. Furthermore, tracing

only discovers an object’s address immediately before the object’s content is needed. Since tracing visits memory addresses in a way that is neither predictable nor sequential, prefetching cannot be used. If the live data doesn’t fit in the CPU cache, tracing will incur a RAM latency for each object visited.

Tracing can be made faster if the objects can be laid out in memory in the same order that tracing visits them. This will cause tracing to fetch memory in roughly sequential order, causing the hardware prefetcher and RAM row buffers to yield the highest throughput. As it turns out, “copying” collectors move objects to just the needed order. A copying collector copies each live object when it is first encountered while tracing, resulting in objects laid out in tracing order. However, other collectors preserve object order and thus do not produce sequentially traceable heaps. In the rest of this section, we first measure the benefits of linearizing the live data in another type of garbage collector. Then we discuss a potential linearizing method which exploits the existing free space; the method doesn’t require doubling memory space as the copying collector does.

We examine the impact of arranging the live data into tracing order on tracing times with HotSpot¹, a Java virtual machine, in OpenJDK7u6 [3] and HSQLDB 2.2.9 [1], a high performance relational database engine written in Java. Using HotSpot and its compacting collector only (the “parallel compacting collector”), we run the HSQLDB performance test – a Java program which stresses HSQLDB with many update and query operations. The live data generated by the performance test is approximately 1.8 GB and we configure the garbage collector to use six threads. Throughout the execution of the performance test, the collector is called several times and time spent tracing is recorded. Next, we run the HSQLDB performance test using a modified version of the compacting collector which occasionally copies all live data into tracing order in the same way as a copying collector. We record the time spent tracing and observe that copying the objects into tracing order yields a $1.3\times$ speed improvement in tracing time.

Although our experimental compacting collector uses copying collection to arrange the live data in tracing order, a different method may be needed in practice. For instance, a compacting collector could take advantage of free space between live objects (before actually compacting) by filling it with subsequent live objects according to tracing order. The more unpredictable accesses replaced by sequential accesses, the faster tracing will be,

¹revision 3484:7566374c3c89 from Aug 13, 2012

resulting in reduced overall time spent in the garbage collector.

4 Alternating RAM fetch and computation

If an application traverses many different paths through a data structure, it will not be possible to lay out the data in memory so that it is always accessed sequentially. For example, looking up different keys in a tree will follow different paths. No layout of the tree can linearize the memory accesses of all tree lookups.

We can borrow an idea from the way I/O systems cope with latency. If many concurrent independent requests are outstanding, we can issue them in a batch, and pay just one latency in order to fetch multiple results. This doesn't reduce per-request latency, but it might increase overall throughput. This section describes how we apply this approach to coping with RAM latency during lookups in the Masstree key-value store.

Masstree [6] is a high performance key-value store for multi-core. Each core has a dedicated thread processing get/put/scan requests. Masstree stores all key-value pairs in a single in-memory B+tree variant shared by all cores. Masstree scales well on multi-core processors through optimistic concurrency control. We discuss concurrency on multiple cores in the next section, and consider Masstree on a single core in this section.

Despite 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. This design reduces RAM dereferences for key comparisons. However, RAM latency still dominates Masstree's performance. Each key lookup follows a random path from the root to a leaf, incurring a RAM fetch latency at each level of the tree. This limits the per-core performance to about a million requests per second.

To get higher throughput, we modify Masstree to interleave multiple lookups on each core (we assume requests arrive in batches, which is reasonable in practical systems [7]). We call this version Interleaved Masstree. Figure 1 shows the lookup pseudocode. For each request at each tree level, Interleaved Masstree identifies the relevant child (line 16) and then prefetches the child (line 17). It only proceeds to the next level in the tree after prefetching for each of the requests. In this way, the CPU processing to identify the next child for the requests is

```

1  class lookup_state:
2      Key k // the key to lookup
3      Node n // current node. Initially the tree root
4      bool ok // if the lookup is completed. Initially false
5
6  void multi_lookup(lookup_state[] requests):
7      int completed = 0
8      while completed < len(requests):
9          completed = 0
10         for req in requests:
11             if req.ok:
12                 completed += 1
13             else:
14                 if req.n.is_leaf():
15                     req.ok = true
16                     req.n = req.n.find_containing_child(k)
17                     prefetch(req.n)

```

Figure 1: Interleaved key lookups of Interleaved Masstree

overlapped with the prefetching of those children. In addition, the children are prefetched nearly concurrently, so that little more than a single RAM latency is required to fetch an entire batch's worth of children. With a large enough batch, the cycles spent processing each child may cover the entire RAM latency, thus entirely hiding the RAM latency. This technique allows Interleaved Masstree to descend one level of multiple lookups in each RAM latency, whereas non-interleaved Masstree descends a level for only one lookup in each RAM latency. The idea is similar to one used by PALM [5]. For our workloads, we find that batching at least four requests achieves the best performance. On a read only workload, this technique improves the single-core throughput by a factor of $1.3\times$.

5 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 the RAM busy.

Parallelization has much the same effect as the interleaving technique described in Section 4. Both improve performance by doing more work per RAM latency. The difference is that parallelization use multiple cores in order to issue many concurrent or interleaved RAM requests, while the interleaving technique achieves the same goal on a single core.

To demonstrate the benefits of parallelization, we measured the performance of Masstree on multiple cores.

Figure 2 shows the throughput of Masstree on a read-only workload with small keys and values. Each key or value is one to 10 bytes long. As the result shows, with the number of cores increases, the performance of Masstree increases. The reason is that, with more cores, the CPUs could issue more RAM loads per second as suggested by the “Total RAM loads” line.

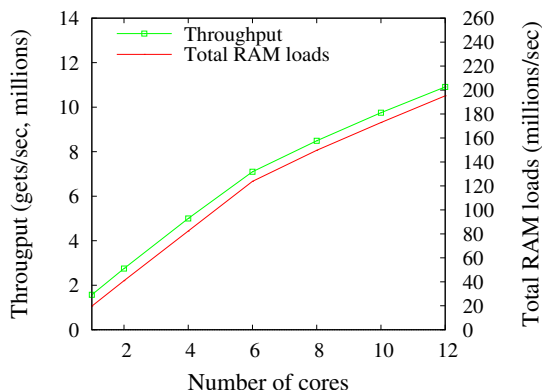


Figure 2: Throughput of Masstree and RAM loads on a read only workload

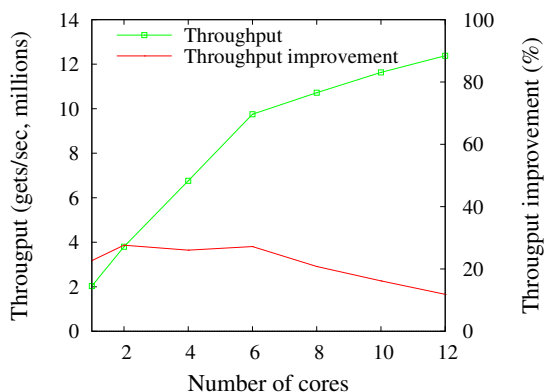


Figure 3: Throughput of Interleaved Masstree and improvement over Masstree on a read only workload

Our experience suggests that parallelization and cache-consciousness techniques could be complementary to each other. Figure 3 shows the throughput of Interleaved Masstree on multi-core on the same workload described above. As the number of cores increases, the performance of Interleaved Masstree increases since the RAM controller is busier with more cores. Interleaved Masstree outperforms Masstree consistently by 12-30%. However, the performance improvement decreases as the number

of cores increases. The reason is that, with enough cores, parallelization alone can saturate the RAM system, so there are no idle RAM resources for Interleaved Masstree to exploit.

6 Discussion

We don’t claim that the three techniques above can be applied to any RAM-dominated program. In fact, for RAM-latency dominated programs that are not parallelizable and difficult to pipeline, the RAM-latency bottleneck can be hard or impossible to hide or avoid, limiting the overall throughput. For example, suppose a program performs a single random walk over a graph for a fixed number of steps, and returns the weight of the edges it travels along. It doesn’t seem that any of the three techniques is applicable to such a program. Nevertheless, these techniques work well for programs that with the right properties. For example, the interleaving technique applies to programs that serve many independent requests. For such programs, we can batch multiple requests and interleave the memory accesses, preventing the DRAM latency from limiting the overall throughput.

Applying the interleaving technique can be difficult. One way to reduce the effort is to automate the interleaving. Compilers do this for loops in some circumstances, but typically not for complex situations involving pointers. Automatic interleaving may be difficult or impossible since it requires detecting accesses to shared data and resolving conflicting accesses. Conflict resolution may be impossible without the help of the programmer.

On the other hand, interleaving certain operations on certain data structures is possible. It might be interesting to identify applications that are RAM-latency bottlenecked, what operations and data structures need to be interleaved, and how much that helps performance. One potential application is Memcached [2], which provides a multi-get API at the client side. With the multi-get API, the client batches multiple get requests, and sends them to the server at once. This opens an opportunity for the server to interleave the processing of multiple gets in the same batch. Some preliminary measurements show that, for a Memcached-like hash table which uses linked-list-based chaining, interleaving multi-get can improve the single-core throughput by more than 30% on a read-only workload. We intend to further investigate the feasibility and measure the benefit of this modification to Memcached.

One difficulty that arises when optimizing these applications is the lack of tools. We use Linux *perf* to find

hot spots and manually inspect the code to see if a RAM fetch bottleneck seems likely. This can be misleading since it can be hard to tell if the hot spot is busy doing computation 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.

7 Conclusion

This paper identifies a class of applications that suffers from RAM latency bottlenecks. We describe three techniques to address these bottlenecks. We implement and evaluate these techniques on several applications, and observe a significant performance improvement. We hope this work will inspire people to optimize their applications in a similar way.

References

- [1] HyperSQL HSQLDB - 100% Java Database. <http://hsqldb.org/>.
- [2] memcached - a distributed memory object caching system. <http://memcached.org/>.
- [3] OpenJDK. <http://openjdk.java.net/>.
- [4] Intel 64 and IA-32 Architectures Optimization Reference Manual, section 2.1.5.4. 2012.
- [5] S. Jason, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel architecture-friendly latch-free modifications to b+trees on many-core processors. *Proceedings of the VLDB Endowment*, 4(11), 2011.
- [6] 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.
- [7] 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.
- [8] H. Park, S. Baek, J. Choi, D. Lee, and S. H. Noh. Regularities considered harmful: forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems, ASPLOS '13*, New York, NY, USA, 2013.
- [9] 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.