Eliminating Dark Bandwidth: a data-centric view of scalable, efficient performance, post-Moore

Jonathan C. Beard and Joshua Randall [*]

ARM Research - Austin, TX
{jonathan.beard,joshua.randall}@arm.com

Abstract. Most of computing research has focused on the computing technologies themselves versus how full systems make use of them (e.g., memory fabric, interconnect, software, and compute elements combined). Technologists have largely failed to look at the compute system as a whole, instead optimizing subsystems mostly in isolation. The result, for example, is that systems are built where applications can only ask for a fixed multiple of data (e.g., 64-bytes from DRAM), even if what is required is far less. This is efficient from a hardware interface perspective, however, it results in consuming valuable bandwidth that is never utilized by the core; this hidden bandwidth is effectively dark to the system. The causes of dark bandwidth are systemic, built into the very core of our virtual memory abstractions and memory interfaces. Continued focus on newer, revolutionary memory technologies to improve surface performance characteristics without a systems focus on reducing data movement will simply push this problem off onto future systems. This paper examines the problem of dark bandwidth and offers a holistic approach to reduce overall data movement within future compute systems.

1 The Problem

Computation is typically not the bottleneck that it once was. Computing itself is faster and less hungry (in terms of energy [7]) than the memory and interconnect that supply the data for computation. Much computing research has focused on providing efficient compute, resulting in the compute cores found in today's CPU sockets. The aforementioned compute devices perform very well for SPEC [4] and LINPACK [3] workloads, however, the compute devices and accompanying subsystems optimized for these workloads do not necessarily perform well for the applications executed by many HPC [2] and big data systems [12]. Two things are clear: first, architecture researchers have produced a myriad of ways to compute things efficiently, and second, research into ways to feed compute has not kept up. The technical community has been working on a false thesis: that compute systems efficiently utilize the bandwidth provided to the core by the memory subsystem. Not only does research suggest that this is often not the case, it shows

[*] Partially supported by U.S. DoE FastForward-2 Contract - Subcontract No. B609229

that many applications make use of only a small fraction of the data moved to the compute elements [10]. The trend towards heterogeneous accelerators only serves to exacerbate the problem as bus length and buffering increase. Recovering this lost bandwidth and reducing superfluous data movement gives the system more usable bandwidth for real computing. It is not just the size and speed of the memory technology that matters, it is how you use it that will enable future systems to utilize what is effectively dark or hidden memory bandwidth.

The type of compute elements today range from simple in-order cores to massively complex out-of-order ones. The compute elements within these go from small arithmetic units to heavyweight vector engines. General purpose CPUs are hobbled by the very fact that they must be general. For many applications the general purpose has given way to better adapted hardware [9]. The general purpose graphics processing unit (GPGPU) revolution was launched by the realization that vector units with many threads could operate on specific workloads very efficiently without the constraints of things like having to run an operating system. Many other accelerators have since become mainstream, including FPGAs. Data must still be delivered to the accelerator, just as it must be delivered to the CPU core itself. The main difference between the CPU, and accelerators like the GPU is that memory hand-off must be coordinated by an external agent: the CPU. No matter how efficient engineers and researchers make cores and accelerators, the model of paging in data from memory one DDR burst at a time [5], all coordinated by the general purpose core, limits the efficiency and scalability of all future systems intended for more sparse workloads.

Cache line utilization is one way of measuring the bandwidth utilization of a DRAM burst given that the burst and cache line granularity typically align (e.g., 64-bytes). On average, when measured with profiling tools (e.g., DynamoRio), HPC application utilization for the L1-D cache is between 20-80%, with spikes for kernels like *DGEMM* up to 100% during "hot" loops. The numbers for the L2 cache shift only slightly, with kernels like *DGEMM* exhibiting high reuse at this level as well. The worst offenders are applications like *GUPS* whose L1-D utilization stays at around 20%. Less bandwidth intensive applications like *LULESH* also have room for improvement as elements from a lattice must be gathered to contiguous memory and then scattered again. Simply increasing the available bandwidth, as many memory and interconnect focused technologies do, does nothing to address the critical problem of wasted data movement.

Even when a cache line is fully utilized, often it is quickly evicted and never used again because of high reuse distance. Reuse distance is the amount of relative time or number of bytes, depending on the metric used, between one use of a data element and another. Even when cached data are fully utilized, often the reuse distance is high. Most applications have varying phases where reuse can range from immediate (zero bytes between subsequent accesses), to kilobytes, all the way to infinity (perfectly streaming accesses). To make the most out of a modern hierarchy, it is critical that systems designers find a way to maximize the physical proximity of highly reused dense data to the best compute element possible, while providing the best bandwidth possible for the streaming data (high reuse distance). A cache hierarchy, which on a per component basis is generally static in size, is a poor structure for workloads where reuse distance scales with the data set size.

Sparse applications have both low utilization and high reuse distances. Irregular applications often share the aforementioned properties, but they also typically have unpredictable data access patterns (i.e., they are data dependent). Sparse and irregular applications are found both in HPC (e.g., lattice and geometric multigrid calculations) and big data analytics (e.g., MapReduce, databases). These applications often explore only a few data points within a data region (e.g., 4 KB page). In current systems, this often results in a page-sized region being loaded from network or nonvolatile memory into DRAM (through the main processor core) and then back to the core for computation. This results in a lot of data movement (e.g., from source, to CPU, to DRAM, and then back to core). This is very inefficient, however, the problem is worse. Performance-enhancing technology such as the hardware prefetch unit, as well as the DDR burst length itself, often inadvertently evict useful lines from the cache hierarchy, while allowing useless data to hitchhike into the cache, wasting scarce bandwidth. For off-chip accelerators (e.g., GPGPUs), additional hand-off and coordination of virtual pages must also be managed, adding even more overhead.

Memory technologies are proverbially five years out; that is they often perpetually remain science projects that fail to scale to production. Even when technologies do make it to production, more revolutionary technologies have a hard time competing with incremental improvements on a cost and performance basis. The time necessary to develop revolutionary technologies (e.g., MRAM) often makes possible for incremental improvements in legacy technologies to outpace the improvements that would come through adopting a newer, more revolutionary one. Even when new technologies come to market (dark bandwidth), they will inherit systemic flaws that result in wasted data movement. Ignoring the system deficiencies described in this section when bringing new technologies to market, only means pushing the data movement problem into the future rather than solving it.

2 Solutions

Data movement dominates computation at scale [1]. There are a few options to increase bandwidth utilization and reduce data movement. Some researchers suggest that byte-level addressing is the key to improving bandwidth utilization. While true, when faced with the harsh reality of engineering a system with addressing commands that equal the size of the data requested, at face value, this idea seems quite impractical. A creative solution that arrives at the same effect is in-/near-memory rearrangement [8], which effectively delivers byte-level addressing through bulk data requests. Processing in- or near-memory (PINM) is another solution quickly gaining traction with both academic and commercial researchers. The idea, however promising, is faced with many hurdles. Increased proximity of memory cells and compute elements raise the risk of heat-induced memory leakage, decreasing efficiency. In modern systems, the virtual memory system is also an enemy of those wishing to reduce data movement. What engineers developed to protect systems, improve multiplexing, and ease system programmability, now hobbles scalability and performance.

2.1 Chopping Down Sparse Data

The efficiency of large, heavyweight compute units is extremely hard to beat. A wide vector unit can churn through packed-data computation extremely well. The issue with these is that data often does not come packed, so programmers often use gather-scatter instructions. These are used to pack data from multiple locations into a single vector register and then return it back to non-contiguous memory. In practice, gather-scatter instructions are not as efficient as they could be. The cache lines are still underutilized (only the register is packed) leading to corresponding unused memory bandwidth (see Figure 1). The only way to reduce the data movement for applications in need of heavyweight compute (e.g., vector units) with middle (8KB) to high (> 128KB) reuse distances and potentially low cache line utilization is through in-/near-memory or in-storage rearrangement (there are better techniques for workloads with less compute intensity).

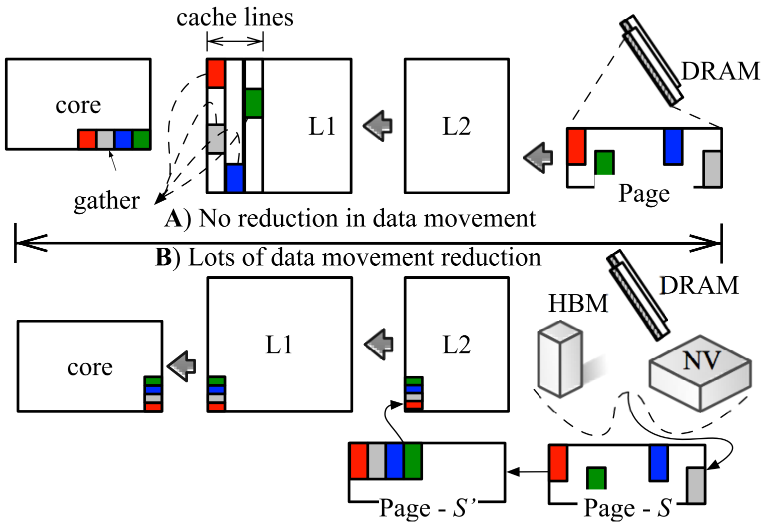

Fig. 1. Image **A** above shows the data movement pattern for a traditional gather instruction as implemented on many architectures. Cache lines are gathered into the cache hierarchy one-by-one, then offsets are accessed to pack data into a vector register. Image **B** shows the potential for in- or near-memory rearrangement, which requires relatively simple logic near memory to compact data from S before it reaches the cache hierarchy S'.

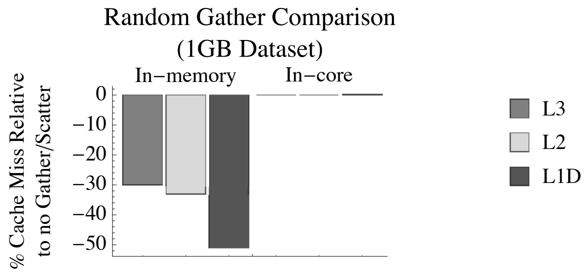

Fig. 2. Benchmark of a random gather across a 1 GB data set using a leading vector architecture's gather instruction to compare to an emulated near-memory rearrangement. Near-memory rearrangement results in a ∼50% reduction in L1-D misses and a ∼30% reduction in LLC misses compared to a standard gather instruction. Both results are normalized to the same code run without a gather instruction. In all cases, the hardware and software stacks are kept the same.

2.2 Processing In- or Near-Memory

For applications with middle to high reuse distances and/or low cache line utilization that can make do with less power hungry compute, PINM is a good option. As an example, if an architecture has a load width of 128b/cycle, the reuse distance of 8KB gives 64-cycles to compute in cache. At a distance of 64KB this grows to 512-cycles, which is more than enough for in-cache processing (assuming a reasonable access time for caches). Many big data applications fall into this category, as do database and string processing (e.g., genomics) workloads. The options for compute elements within a PINM system range from simple fixed-function state machines to putting heavyweight cores in- or near-memory. The line blurs between what is PINM and what is simply a processor with a shorter bus or giant cache made of high bandwidth memory. The definition that best fits is a processor closer to memory that is supplemental to a general purpose processing core (essentially an accelerator for sparse and irregular applications). There are two possible locations for PINM, on- or off-chip. The on-chip devices can be split into in-cache or in-system cache. The main advantage for on-chip is the potential availability of low-latency links to retrieve virtual to physical address translations. Off-chip devices fall into multiple categories as well, in-component devices (e.g., SSD), in-controller (e.g., external memory controller, interconnect), and in-memory devices. A primary disadvantage for off-chip devices: high latency off-chip virtual memory translation.

Functional considerations for on-chip PINM devices are many, especially when coordinated from a general purpose core. Modern processors require things like out-of-order issue, exception handling, and for PINM outside of the coherence network (i.e., either on-chip or off-chip RAM) careful handling of virtual-to-physical address translation is required. Building a PINM in-cache is relatively simple, coordinating it as a system is quite difficult. With current virtual memory systems, off-chip PINM is hobbled by the fact that the software must be coordinated to know where the memory is located. This leads to several issues that must be considered outside of those listed for on-chip PINM. PINM with current virtual memory systems either must rely on the software to place pages statically in memory for the PINM device, restrict operation to huge pages, or rely on an input/output memory management unit (IOMMU). The IOMMU translation bottlenecks for GPGPU found in literature largely apply to PINM (e.g., 20x higher translation cost, 1-20 MPKI [6, 11, 12]), with some exceptions. For PINM, the ideal processor technology is not the heavyweight GPGPU type core, but many small simple cores. The access patterns are also different, instead of lock-step data parallelism, PINM targets very sparse memory access patterns. The usual solution of increasing page size (successful for GPGPU) typically fails for PINM.

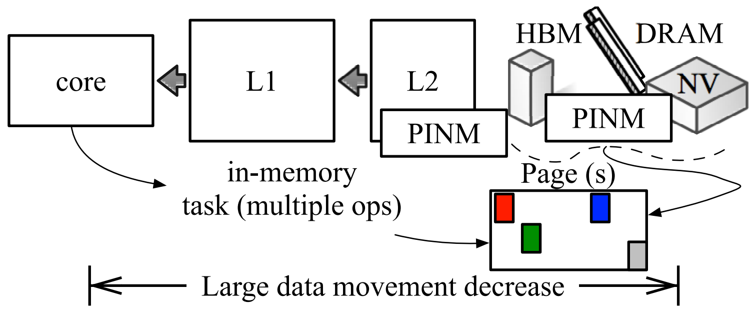

Fig. 3. PINM involves sending instructions closer to where the data rests within the memory device versus bringing the data to the core. Lightweight PINM cores could exist at multiple places in the memory hierarchy.

The simplest embodiment of PINM hands a single page at a time to each in-/near-memory processor (to reduce logic needed to handle contention), the use of huge pages in this scenario limits the number of cores that can be utilized without extra synchronization hardware, limiting the overall parallelism; ideally a smaller page size (e.g., 4 KB) would be used. Figure 4 demonstrates the inefficiencies of virtual memory for a PINM system using the preferred 4 KB page size. Removing the bottleneck of virtual memory would enable PINM technology to be more efficient. Without fixing the memory system, PINM technology will be forced to limit solutions to a select range of applications (e.g., in-memory databases) or force adoption of more restricted programming models (e.g., PGAS) to work around the fundamental limitations of page-based virtual memory. The same solution will likely enable faster, simpler, and easier implementation of a unified memory space for accelerators (including ease of handling pages in virtualized GPGPU clusters). The time is ripe for a rethink of virtual memory and a rethink for the relationship of the operating system, memory system, and runtime.

3 A Common Problem: Translation

The biggest hurdle to implementing the data movement reduction technologies described in the last section is page-based translation. The problem with page-based translation is, first and foremost, the fixed-size pages themselves (pages limit addressability and parallelism for tightly coupled accelerators and PINM devices). Secondly, the reliance on page table caches for non-general purpose compute elements becomes a bottleneck for all out-of-core accelerators [11]. As architectures vie for more efficiency, the trend of late is towards specialized accelerators. In order to maximize the utility of accelerators, something must change in the virtual memory system. Memory fabrics cannot continue to ignore the software co-design to optimize the extension of the virtual memory abstraction to all compute elements. Satisfactory solutions that would enable page-based virtual memory to extend outside of general purpose cores (e.g., accelerators) in a low overhead manner are yet to be found. The current state of the art results in ∼1.5-cycles of translation overhead per cycle of compute for a PINM sparse compute accelerator device (see Figure 4). Nothing short of a rethink of the virtual memory system will solve the problem.

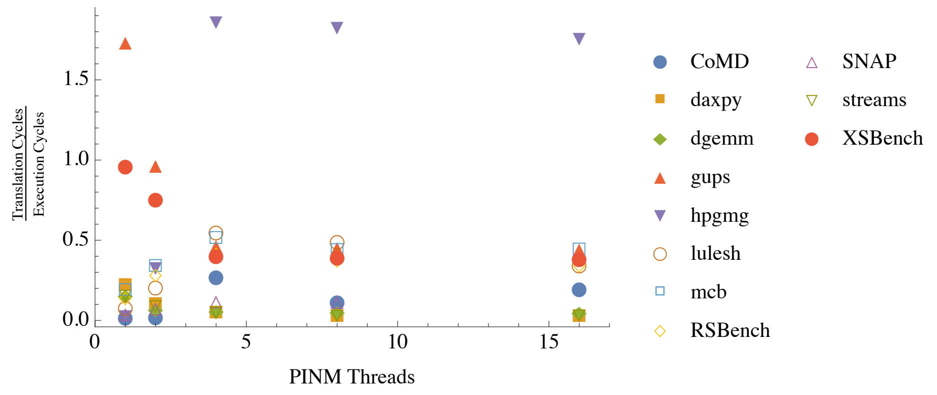

Fig. 4. Projected cycles used on translation versus execution, estimated by measuring actual application miss rates on a leading accelerator architecture (64-entry L1 TLB), assuming a $1GHz$ clocked multi-core main processor, an average of 582-cycles per IOMMU page walk (times above estimated assuming a 60% IOMMU hit rate), PCIe ATS protocol, and 50-cycles PCIe latency round trip (optimistic). No latency is taken into account for page source (e.g., disk, RDMA), nor DRAM page open, making this graph optimistic. Execution time does not include time spent stalled.

Reliance on a set of fixed-size pages (even with an assortment of sizes) has the unfortunate characteristic that each page entry represents only N-bytes of memory and there can only be a set number of entries in the physical hardware. This results in the reach (range of addresses addressable by the translation look-aside buffer, TLB) being fixed by the number of entries in the TLB multiplied by the size of the largest page size supported. TLB size has grown (as well as number of entries and associativity), however, even the largest of TLBs can only address a small fraction of the available address space (e.g., 1TB). What happens when the memory space is a petabyte, then exabyte? It should be apparent that the lack of TLB reach is quickly becoming an issue for general purpose cores in addition to out-of-core devices. Any rethink of page-based virtual memory will clearly pay dividends for all compute elements, not just accelerators.

4 Conclusion: It's the System

Pulling the memory hierarchy into the compute system as a first class citizen, not only to feed cores but as an active participant, will enable extracting more performance from less revolutionary memory and compute technologies. Reducing overall system data movement will likely net system designers far more over the next decade than any revolutionary technology changes. Enabling small amounts of computation in- or near-memory along with fixing the virtual memory system could enable future systems to recapture dark bandwidth. Doing all of this, while not breaking all extant software is a huge, but not insurmountable, challenge. It's not just the memory technology or compute alone that should be the focus, it's how the compute system as a whole uses it.

References

[1] Data Movement Dominates. `https://goo.gl/rro35D`, accessed March 2017
[2] Dongarra, J., Heroux, M.A.: Toward a new metric for ranking high performance computing systems. Sandia Report, SAND2013-4744 312 (2013)
[3] Dongarra, J.J., Moler, C.B., Bunch, J.R., Stewart, G.W.: LINPACK users' guide. SIAM (1979)
[4] Henning, J.L.: Spec cpu2006 benchmark descriptions. ACM SIGARCH Computer Architecture News 34(4), 1–17 (2006)
[5] Jacob, B., Ng, S., Wang, D.: Memory systems: cache, DRAM, disk. Morgan Kaufmann (2010)
[6] Karakostas, V., Gandhi, J., Cristal, A., Hill, M.D., McKinley, K.S., Nemirovsky, M., Swift, M.M., Unsal, O.S.: Energy-efficient address translation. In: High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on. pp. 631–643. IEEE (2016)
[7] Kestor, G., Gioiosa, R., Kerbyson, D.J., Hoisie, A.: Quantifying the energy cost of data movement in scientific applications. In: 2013 IEEE international symposium on workload characterization (IISWC) (2013)
[8] Lloyd, S., Gokhale, M.: In-memory data rearrangement for irregular, data-intensive computing. Computer (8), 18–25 (2015)
[9] Markov, I.L.: Limits on fundamental limits to computation. Nature 512(7513), 147–154 (2014)
[10] Srinivasan, J.R.: Improving cache utilisation. Tech. rep., University of Cambridge, Computer Laboratory (2011)
[11] Vesely, J., Basu, A., Oskin, M., Loh, G.H., Bhattacharjee, A.: Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In: Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on. pp. 161–171. IEEE (2016)
[12] Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., et al.: Bigdatabench: A big data benchmark suite from internet services. In: High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on. pp. 488–499. IEEE (2014)