NOS\textsubscript{2}: Network Optimized Scale-Out Servers

Anirban Nag, Chandrasekhar Nagarajan, and Karl Taht

Abstract—Scale-out servers is term coined to express the scalability of servers in the cloud domain. While at the high-level this simply means increasing storage, memory and processing as the number of users increase, there are limitations in the current architecture which prevent such a straight-forward approach. Namely, latency is a critical aspect which does not improve in the same fashion as throughput by simply adding more nodes to a cluster. As the data center grows, communication networks grows, the number of hops grow, and the ultimately it is the user who incurs this penalty. As architects, it up to us to design a new framework which can support this growing data center framework.

NOS\textsubscript{2} seeks to reduce the cost of network traffic by restructuring the memory hierarchy. The approach employs an emerging memory technology, hybrid memory cubes (HMC), as a high-capacity cache. This allows the memory system to leverage the spatial and temporal locality present cloud servers workloads, even though it is typically in the order of gigabytes. In addition, NOS\textsubscript{2} leverages the HMC device’s logic layer to work directly with the NIC, further reducing latency and bandwidth usage within individual servers. After exploring several potential architectures, we find that NOS\textsubscript{2} has a potential latency benefit of X% over traditional network servers and X% over the previous state-of-the-art network server which employs HMC memory.

Index Terms—HMC, NIC, DRAM, latency, DDIO

1 INTRODUCTION

The traditional study of computer science can be thought of as a chicken-and-egg problem. We design hardware based on the needs of software, yet, software is designed based on the capabilities of hardware. Because of software's flexibility, it seems that the architecture can be considered the "egg" and the the software is the "chicken" modeled after it. As the cloud domain grows, what does this mean? As Ferdman et al \cite{1} have pointed out, modern server architecture does not differ drastically from consumer personal computers. In fact, ignoring the scale, the only key difference between rack-based cloud servers and commodity desktop computers are the lack of graphics and audio processors, and the addition of high-end network interface cards \cite{1}. Despite, software architects have done a good job of utilizing this lackluster architecture.

While at some point many of these design choices may have made sense, they now seem to be penalties in the scale-out server domain. Alternatively, NOS\textsubscript{2} asks the question if we design a new type of server with a network and memory system designed specifically for cloud domain. The goal is to combine the state-of-the-art software solutions, with a state-of-the-art hardware solution. The benefit of this approach is two fold: from the software developer’s perspective, they no longer have to design around server limitations; from the architect’s perspective, the change is incremental, only the memory subsystem and NIC are changed, the core architecture remains consistent.

Specifically, NOS\textsubscript{2} seeks to improve network direct-memory accesses by nearly an order of magnitude, while simultaneously improving performance of on-chip workloads. Firstly, the networking component is addressed by a technique which mirror’s Intel’s DDIO technology. DDIO typically improves performance 10-15%, despite using just 10% of the LLC. On a modern processor, this is typically just a few megabytes. While diminish returns will of course exist, by increasing this region to tens or hundreds of MBs performance improves further. Additionally, the NIC relocated so that network traffic does not use a PCIe bus, but rather a much higher-bandwidth 3D memory TSVs, improves both latency and bandwidth for such networking accesses. The auxiliary benefit is that NOS\textsubscript{2} architecture also provides a high-capacity cache for other workloads running on the server. Specifically, this cache is in the order of GB, which allows it to capture the locality present in cloud computing \cite{2}.

2 BACKGROUND AND RELATED WORK

Because our key idea expands off a previous work, soCache \cite{2}, we leave the description of the previous hardware architectures to section 3. Here we instead take time to explain scale-out server workloads. Firstly, let us examine a few common classes of these cloud computing workloads. Data Serving. A large portion of the cloud domain is data serving, such as NoSQL databases including Facebook, Google and Amazon. For the sake of our purposes, we lump this category together with media streaming as well. While not only are they intuitively similar, empirical evidence also suggests they have similar performance characteristics \cite{1}. Examples of media streaming include YouTube, Netflix and Hulu. At the high-level, all of these type of workloads have the same problem to solve, a very large database (in the order of TB or PB), and many clients issue requests from the database, often with parallelism. While of course the implementations vary, at the high-level, this class of applications solves this issue by partition data into “shards”. In this way, each server can service requests to a particular shard. By making the shards of reasonable size, more precisely, a subset of the database which can fit into a single server’s memory, requests can be processed efficiently.
SAT Solving. The propositional satisfiability problem (SAT) is a common computer science problem. Traditionally, these problems were very large and done within HPC clusters with special high-bandwidth, low-latency network interconnects. While this offered good performance, it comes at great cost. The cloud domain has introduced the concept of low-cost server rental; however, these resources are not only made up of general commodity servers, they also use traditional IP-based networking. To compensate for this, software has been developed for more complex partitioning and load balancing to compensate for network resources. While the solution is feasible, and still provides substantial economic benefit, one can start to see the impacts of the general commodity-based servers for the usage of broader domains.

Web Search. Much like data serving, web search also uses the shard model. For web search applications, the “database” comprises of website indexes, which are then partitioned among nodes. When a search request is made, a front end machine must propagate this request to nodes in parallel, then collect and sort the data before finally responding. Because web search is a particularly latency critical component, again the size of each shard is limited by an individual node’s memory capacity. The larger issue, however, is the many parallel requests. Because the network within web-search nodes is relatively high latency compared to on-chip communication, and modern server networks typically optimize for larger chunks of data, software chooses to “overfetch” data, and rely on the front end to aggregate and process all the received data.

All of these workloads have specific methods to work around limitations of current cloud-server architecture. However, these methods incur additional penalties like increasing total network traffic, or sacrificing latency for throughput. Alternative approaches to handling large in-memory databases have been proposed. Particularly, RAMCloud provides a networking infrastructure that is latency, rather than throughput optimized [3]. This framework provides the opportunity to completely restructure the method in which requests, such as search queries are handled within the network. NOS2 seeks to push this latency-oriented networking concept one step further, and back it up with proper hardware support to boost performance yet another order of magnitude.

3 OVERVIEW

The key idea of NOS2 is massively improve network latency, bandwidth, and improve overall scale-out server workloads. NOS2 makes combines two state-of-the-art technologies: Intel DDIO technology and HMC caching. When combined, these concepts allow NOS2 to cater to both network I/O bound processes and memory bound cloud server workloads.

3.1 Intel DDIO

Network Interface Controllers (NICs) are networking components which are connected to the CPU via PCI or PCIe bus. They serve various purposes such as indicating the availability of I/O to the host processor (polling based or interrupt driven), transferring data packets to or from the host (Programmed I/O or Direct Memory Access (DMA)) and processing network traffic using the TCP offload engine.

Traditionally while servicing I/O requests, the NIC would perform DMA to the host memory for incoming or outgoing data because of cache space being a scarce resource. With the advent of modern generation of processors with larger last level cache (up to 55MB), Intel came up with the Data Direct I/O (DDIO) technology for its Xeon E5 and E7 line of server processors, which makes the processor cache as the source and destination of I/O data as shown in Figure 1, thus eliminating memory accesses. This greatly reduces the latency of I/O bound processes, increases the utilization of network bandwidth and also reduces power consumption. Intel reports a 3× increase in data rate and 10-15% decrease in latency over previous generation Xeon processors due to DDIO technology.

Fig. 1: Left: Traditionally NIC I/O requests serviced from memory; Right: Intel DDIO makes LLC the source and destination of I/O data

Intel DDIO does not use a dedicated cache for I/O operation and hence the data in the cache can be used by active threads. For outgoing traffic, packet creation is done with data in the cache if already present and sent to the NIC. Often, the data to be sent is in the recently used working set with data in the cache if already present and sent to the NIC. During incoming data traffic, the NIC uses two modes of operation for data delivery: write update and write allocate. If the data being delivered to the processor results in cache hit, it is updated in place, else space is allocated in the LLC for the data to be written. Now write allocation may lead to cache pollution if data streams are not consumed by running threads or when rate of incoming data is higher than what threads can handle. So Intel only reserves 10% of the cache for write allocation.

Thus, if cache size where to increase, there would be more room for write allocation, and more cache hits for outgoing packet creation and write updates. As we have seen already, DDIO effects greatly improve I/O bound processes with as high as 2× improvement in I/O bandwidth utilization even with relatively small 20MB LLC. This demonstrates the potential of DDIO using larger caches for scale-out servers with high memory footprint.

3.2 Scale-out Cache using HMC

Scale-out workloads have high request level parallelism because of which many-core architecture is an ideal candidate for them. This leads to a significant memory bandwidth requirement which can be satisfied with modern
We state the opportunities and limitations for both HMC and NIC and justify our design choices in the following subsections:

4.1 Hybrid Memory Cube (HMC)

Hybrid Memory Cube or HMC is a 3D memory architecture where multiple DRAM dies are stacked on top of each other, each die being a rank. It also consists of a logic layer which has the memory controllers and serializer-deserializer blocks for four differential SerDes links that connect the memory module to the processor and to other HMC modules. The DRAM dies are accessed using through-silicon-vias (TSVs). The dies are partitioned in 16 blocks, and one such block from each die constitute together as a vault. Hence there are 16 vaults with each vault having a separate memory controller. The HMC provides high bandwidth with 512 TSVs fetching data from the 16 vaults, with high bank level parallelism options. In order to use the HMC as a cache, the tag array needs to be stored as well. Storing tag array in the HMC memory adds to the access latency, hence they are stored in SRAM arrays on the logic layer of the HMC. This is possible because there is enough room in the logic layer for such SRAM arrays to be implemented once unnecessary SerDes blocks are removed from the logic die. In this case, the HMC stack needs just one SerDes block to be connected to the processor and two DDR controllers for the HMC cache to be connected to DIMMs. Storing data in HMC in normal cache line granularity prohibitively increases the tag array size such that it cannot be implemented on the logic die. Hence data is stored in page level granularity within the HMC.

Due to higher access latency of vaults (20ns) in comparison to SRAM based LLC (15ns), we incorporate the NIC in the logic layer of the HMC such that the extra latency of traversing the PCIe bus is avoided. This dramatically reduces the latency and also provides an internal bandwidth of 128 GB/s which is much more in comparison to PCIe 3.0 (15.75 GB/s). Although this might seem to be an easy solution, there are other problems from the NIC’s perspective as well.

4.2 Network Interface Controller (NIC)

The Network Interface Controller (NIC) is composed of two parts: the MAC, which deals with Media Access Control Layer of the networking, and the PHY, which is essentially the Physical layer. The MAC is responsible for multiple access protocols, collision control, remote DMA (supporting kernel bypass and zero copy which are critical for decreasing latency), etc. Whereas the PHY is used for data encoding and decoding, and the hardware send and receive functions of Ethernet frames; it interfaces between the analog domain of Ethernet’s line modulation and the digital domain of link-layer packet signaling. Traditionally, the NIC is manufactured separately by different vendors and plugged into the motherboard on PCI or PCIe slots. The NIC board consists of separate MAC and PHY chips connected via Media Independent Interface (MII). THE MII
interface can either be parallel (32-bit data-path working at 156.25 MHz DDR) or serial (differential pairs working at 625 MHz DDR).

The problem with integrating the NIC in the logic layer of the HMC is that the NIC as a whole requires high chip area infrastructure. This is mostly due to the massive PHY chip. This is why we implement only the MAC part of the NIC in the logic layer of the HMC, shown in Figure 3. Although this might seem that the packets undergo the latency of being transferred outside the HMC to the PHY part of NIC, this latency was present in the baseline as well.

These simple design choices lead to some trade-offs:

- The HMC capacity which was dedicated entirely to working set of scale-out workloads now need to be shared for networking workload as well, specifically for the write allocate scheme. Intel’s decision to allocate 10% of the LLC for write allocate makes sense for a small 50 MB cache but seems too much for a large HMC cache. We believe a much smaller percentage of HMC cache is enough to avoid cache pollution with incoming data.
- Even though HMC provide very high bandwidth, they are not operated at their peak bandwidth due to adverse effects on memory latency caused due to heavy contention on memory resources. When performance of these servers are characterized by tail latencies, queuing delay becomes a critical factor. When HMC is used as a cache for networking workloads, they tend to eat up on the allocated HMC bandwidth and can suffer more queuing delay in comparison to traditional LLC. We believe that a balance of throughput and latency oriented workloads along with the exclusion of added latency of PCIe would greatly help in circumventing such scenarios.

In Figure 4 we show the layout of the logic layer of HMC, which consists of the SRAM tags for each vault, the individual vault controllers, a SerDes block which connects the HMC to the socket, two DDR controllers for two DDR channels connected to the HMC cache, and the NIC MAC chip.

5 Evaluation

5.1 Cloud Workload Performance

We project workload performance by building a model extended from soCache [2]. Recall that the primary benefit of using the HMC cache is its high bandwidth compared to DRAM. We consider three difference setups, a baseline DDR only system (Figure 5), soCache (Figure 2) and NOS (Figure 3). We consider that each system as a dual-socket, 4-channel per socket server. We consider the baseline with DDR4-2400 memory, but use DDR3-1600 memory in soCache and NOS to help offset the cost of the HMC devices. Note that because of this difference in DDR baseline (DDR4-2400 vs DDR3-1600), our baseline performs significantly better than the baseline present in soCache [2].

From data present in soCache, we are able to define a ratio of bandwidth utilization to latency. This allows us to build projections of average off-chip latency. We see that allocating 384MB for DDIO effects has a minimal impact,

<table>
<thead>
<tr>
<th>TABLE 1: My caption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Type</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
</tr>
<tr>
<td>Memory Capacity</td>
</tr>
<tr>
<td>Cache Capacity</td>
</tr>
<tr>
<td>Memory-to-Cache Ratio</td>
</tr>
<tr>
<td>Additional Miss Penalty</td>
</tr>
</tbody>
</table>
and on average DRAM utilization and latency by increases only 8.8% compared to allocating the full 4GB of HMC devices as cache. HMC cache misses result in a DDR3 bandwidth utilization of 37% (up 3% from soCache) and DRAM latency increases from 57.6ns (up from 55.1ns) on average for CloudSuite benchmarks \[4\]. It should also be noted that our absolute bandwidth numbers used for a calculations are validated by two separate methods. On one hand, the data is directly present in Volos et. al. work \[2\], however, it is also stated that in single-thread, OOO, 3-way ARM core memory pressure range from 0.4GB/s to 1.2GB/s \[4\]. Alternatively, it modern servers comprise of two sockets, each in the range of 20 cores, and hyper-threading support, resulting in roughly 80 threads total. Extrapolating these numbers, we can again validate that memory bandwidth for CloudSuite workloads should be in the range of 30-120GB/s on a modern Intel-based server \[5\].

5.2 Networking Performance

5.2.1 The Gem5 way

We started off evaluating our project idea using gem5 which seemed to be an easier option at first due to object-oriented and modular code, and extensive documentation but later proved be a major hurdle. There were several modules in gem5 that didn’t work and fixing them was our first challenge. For example, the HMC memory module only worked for the given test bench and couldn’t be attached off-the-shelf to any design. We have fixed that such that the HMC can be used in both system emulation mode and full system mode. The cache hierarchy was also not properly configured. We modified the scripts to configure a system with a shared last level cache. To use HMC as a cache, we instantiated instances of HMC vault controllers such that it is invisible to the system memory map. This can be achieved with the help of two flags: in

\[
\text{addr map and conf table reported. In order to use SRAM tags, to support any arbitrary cache line size and to remove MSHRs and coherency, we had to significantly change the cache code which doesn’t inherently support non-coherency. We have made some progress in that direction. The memory mapping policy also had to be changed to map an entire paged-sized cache line of HMC cache to one DRAM row. The whole endeavor for such simulation was to find the effect of queuing delay in HMC cache on networking latency and the effect of reduced cache size on capturing the spatial locality of scale-out workloads.}
\]

5.2.2 The RPC workload profiling way

We also tried a parallel approach to evaluate through trace-based simulations. The idea was to capture memory access requests generated by the CPU. Inspired by the RAMCloud project \[6\], we set up an experiment on the CloudLab server nodes to emulate remote procedure calls (RPC) over the network. RPCs can be thought of as API requests to a binary which is hosted on a remote machine. Essentially, client nodes can establish connection and communicate with a server node which holds a particular executable binary. The network traffic can be parametrized in terms of number of `chunks` per message and size of every chunk. RPC applications, like database workloads mentioned above also possess a Zipfian skew in the pattern of data accessed \[7\].
The nodes were equipped with an Intel Xeon CPU which had 8 cores per socket and could execute two threads per core. For monitoring peak performance, we decided to instantiate enough clients to run the CPU on high throughput while reserving one thread space for the server’s internal operations as well. Therefore, a set of 16 nodes were requested from the node pool and a star based connection of 15 clients communicating with a single server was established.

There are multiple options to get a memory profile from a setup like this. *pmu-tools* is one of such valid candidates but it does not provide the address locations that were queried from uncore events or over the memory bus. We decided to explore using the PIN binary instrumentation tool. The PIN tool can essentially probe and instrument events in a CPU and also has access to the data which was related to the event. The two main elements in a PIN tool are the instrumentation tool and the analysis tool. The instrumentation tool can be used to profile requests on the bus and helps categorize these into different types of events (data cache request, instruction cache request etc.). The analysis tool on the other hand is used for the user specific probing methods which were executed based in instrumentation ‘interrupts’.

To analyze the networking performance, we project the networking operations as well. Therefore, a set of 16 nodes were requested to run the CPU on high throughput and thus cannot be obtained either in the given time frame.

This proved to be the roadblock in trying to profile multi-threaded code. There have been other efforts in academia to tweak the PIN tool to function for a multi threaded system such as this. CMP8SIM \[8\] is one such work which essentially instantiates multiple parallel running PIN tool binaries on top of a CPU. Results from this direction could not be obtained either in the given time frame.

In Table 2, we report the area and power breakdown of the HMC logic layer. The only addition to the soCache baseline in terms of the logic layer layout is the NIC MAC chip which takes up 6.73 mm\(^2\) area. Table 2 shows that NOS\(_2\) layout is well within the budget of the logic layer.

<table>
<thead>
<tr>
<th>Component</th>
<th>Power (mW)</th>
<th>Area (mm(^2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>SerDes Block</td>
<td>1445</td>
<td>6.3</td>
</tr>
<tr>
<td>DDR Controller</td>
<td>4590</td>
<td>10</td>
</tr>
<tr>
<td>NIC MAC</td>
<td>1565</td>
<td>6.73</td>
</tr>
<tr>
<td>SRAM Tags</td>
<td>300</td>
<td>14</td>
</tr>
<tr>
<td>Vault Controller, Crossbar Switch</td>
<td>2890</td>
<td>42.7</td>
</tr>
<tr>
<td>Total</td>
<td>6680</td>
<td>79.73</td>
</tr>
</tbody>
</table>

TABLE 2: Power and Area breakdown of HMC logic layer

6 Conclusion

We investigate the potential of a server solution which caters to both scale-out workloads and networking latency. This work borrows its idea from two independent solutions and integrates them in the same server design. In terms of networking latency, we project the potential benefits and point out the problems in implementing such a project in gem5. On top of the said problems encountered, the other major hurdle is the warm up such a large scale cache.

References


