# Review of Exploiting CXL-based Memory for Distributed Deep Learning

## Motivation

Deep learning workloads have large memory and storage requirements that typically exceed the limited main memory available on an HPC server. This increases the overall training time as the input training data and model parameters are frequently swapped to slower storage tiers during the training process.

## Contribution

- This paper uses the latest advancements in the memory subsystem, specifically Compute Express Link (CXL), to provide additional memory and fast scratch space for DL workloads to reduce the overall training time while enabling DL jobs to efficiently train models using data that is much larger than the installed system memory.
- This paper implements and integrates DeepMemoryDL with a popular DL platform, TensorFlow, to show that our approach reduces read and write latencies, improves the overall I/O throughput, and reduces the training time.
- The evaluation shows a performance improvement of up to 34% and 27% compared to the default TensorFlow platform and CXL-based memory expansion approaches, respectively.

## Design Objects

- Enable access to the additional memory space over the CXL interface for DL workloads
- Avoid throughput bottlenecks in the I/O pipelines and ensure minimal IO response time with prefetching intelligently and placing data close to the processing threads
- Provide fast CXL-based scratch space to store the training data, thus enabling high bandwidth access to data and eliminating IO access to slower storage tiers

## Design Overview

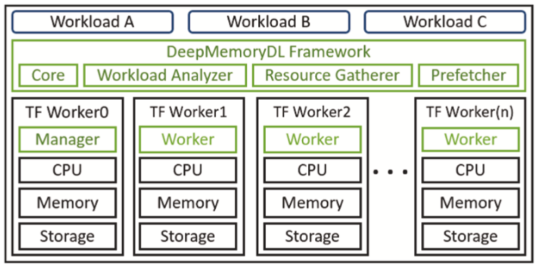
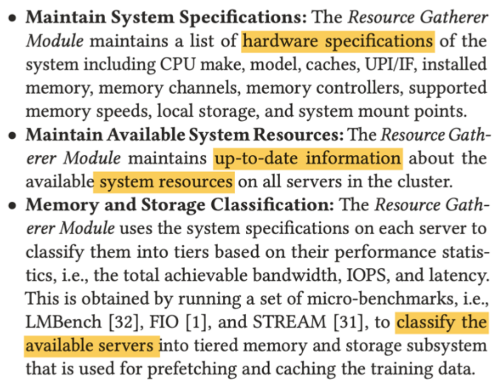


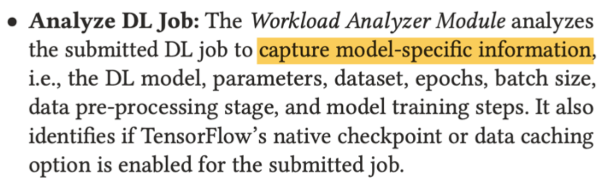
**Figure 4: Proposed architecture of *DeepMemoryDL*.**

- **Resource Gatherer Module** that collects compute, memory, network, and storage resources of all servers included in the cluster.

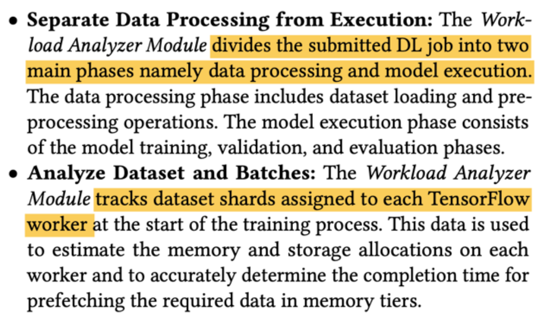


- **Maintain System Specifications:** The *Resource Gatherer Module* maintains a list of hardware specifications of the system including CPU make, model, caches, UPI/IF, installed memory, memory channels, memory controllers, supported memory speeds, local storage, and system mount points.
- **Maintain Available System Resources:** The *Resource Gatherer Module* maintains up-to-date information about the available system resources on all servers in the cluster.
- **Memory and Storage Classification:** The *Resource Gatherer Module* uses the system specifications on each server to classify them into tiers based on their performance statistics, i.e., the total achievable bandwidth, IOPS, and latency. This is obtained by running a set of micro-benchmarks, i.e., LMBench [32], FIO [1], and STREAM [31], to classify the available servers into tiered memory and storage subsystem that is used for prefetching and caching the training data.

- **Workload Analyzer Module** that analyzes the submitted DL workload and breaks down the job in I/O and compute phases.



- **Analyze DL Job:** The *Workload Analyzer Module* analyzes the submitted DL job to capture model-specific information, i.e., the DL model, parameters, dataset, epochs, batch size, data pre-processing stage, and model training steps. It also identifies if TensorFlow's native checkpoint or data caching option is enabled for the submitted job.



- **Separate Data Processing from Execution:** The *Workload Analyzer Module* divides the submitted DL job into two main phases namely data processing and model execution. The data processing phase includes dataset loading and pre-processing operations. The model execution phase consists of the model training, validation, and evaluation phases.
- **Analyze Dataset and Batches:** The *Workload Analyzer Module* tracks dataset shards assigned to each TensorFlow worker at the start of the training process. This data is used to estimate the memory and storage allocations on each worker and to accurately determine the completion time for prefetching the required data in memory tiers.

- **Core Module** has two parts: manager and workers. The manager resides on the same node as the master node in TF and supervises all operations of DeepMemoryDL.

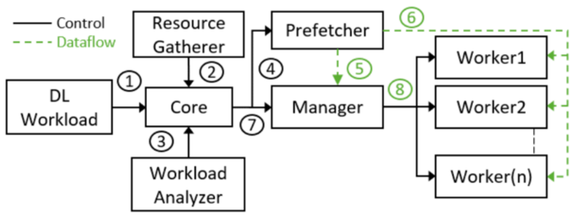
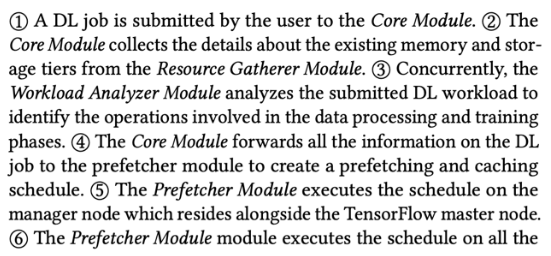
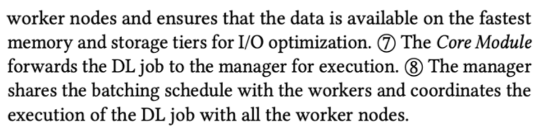


**Figure 5: Control flow between *DeepMemoryDL* components.**

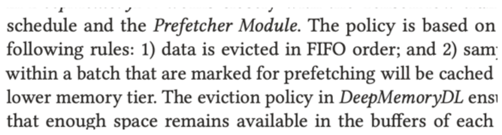


① A DL job is submitted by the user to the *Core Module.* ② The *Core Module* collects the details about the existing memory and storage tiers from the *Resource Gatherer Module.* ③ Concurrently, the *Workload Analyzer Module* analyzes the submitted DL workload to identify the operations involved in the data processing and training phases. ④ The *Core Module* forwards all the information on the DL job to the prefetcher module to create a prefetching and caching schedule. ⑤ The *Prefetcher Module* executes the schedule on the manager node which resides alongside the TensorFlow master node. ⑥ The *Prefetcher Module* module executes the schedule on all the



worker nodes and ensures that the data is available on the fastest memory and storage tiers for I/O optimization. ⑦ The *Core Module* forwards the DL job to the manager for execution. ⑧ The manager shares the batching schedule with the workers and coordinates the execution of the DL job with all the worker nodes.

- **Prefetcher Module** that prefetches data and loads it in the main memory before it is required by the processing thread to minimize I/O stalls.


schedule and the *Prefetcher Module*. The policy is based on following rules: 1) data is evicted in FIFO order; and 2) sam within a batch that are marked for prefetching will be cached lower memory tier. The eviction policy in *DeepMemoryDL* ensu that enough space remains available in the buffers of each

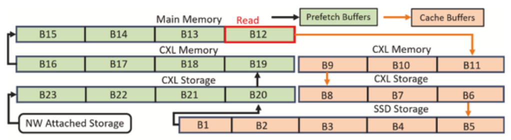
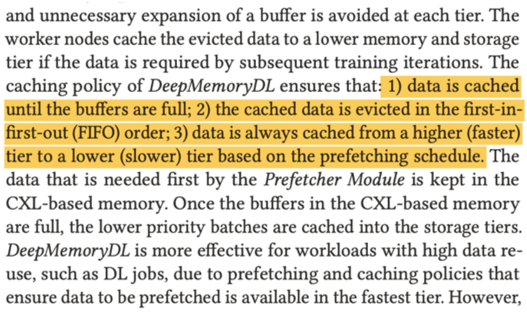


**Figure 6: Dataflow for prefetching and caching in *DeepMemoryDL* using CXL-based memory and storage subsystem.**

- **Caching Policies**


and unnecessary expansion of a buffer is avoided at each tier. The worker nodes cache the evicted data to a lower memory and storage tier if the data is required by subsequent training iterations. The caching policy of *DeepMemoryDL* ensures that: 1) data is cached until the buffers are full; 2) the cached data is evicted in the first-in-first-out (FIFO) order; 3) data is always cached from a higher (faster) tier to a lower (slower) tier based on the prefetching schedule. The data that is needed first by the *Prefetcher Module* is kept in the CXL-based memory. Once the buffers in the CXL-based memory are full, the lower priority batches are cached into the storage tiers. *DeepMemoryDL* is more effective for workloads with high data re-use, such as DL jobs, due to prefetching and caching policies that ensure data to be prefetched is available in the fastest tier. However,

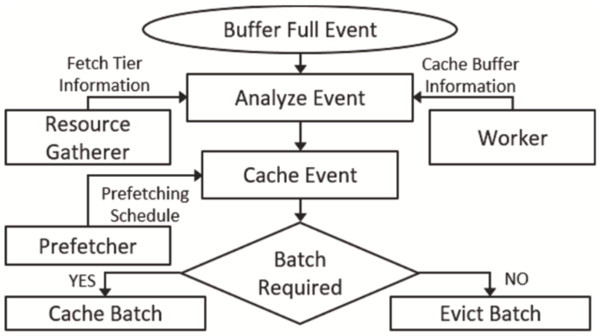


**Figure 7: Caching workflow in *DeepMemoryDL*.**

## Result

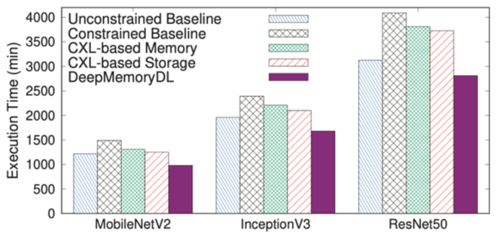


**Figure 8: Total execution time of DL job with batch size of 64 and 3 epochs.**

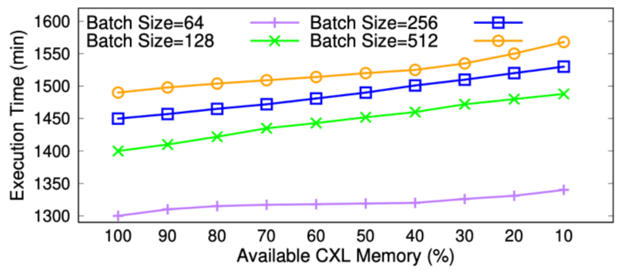


**Figure 11: Impact of CXL-based memory on total execution time of DL job with 3 epochs.**