EE482C Project Proposal: Stream Cache Architectures for Irregular Stream Applications

Timothy Knight Arjun Singh Jung Ho Ahn

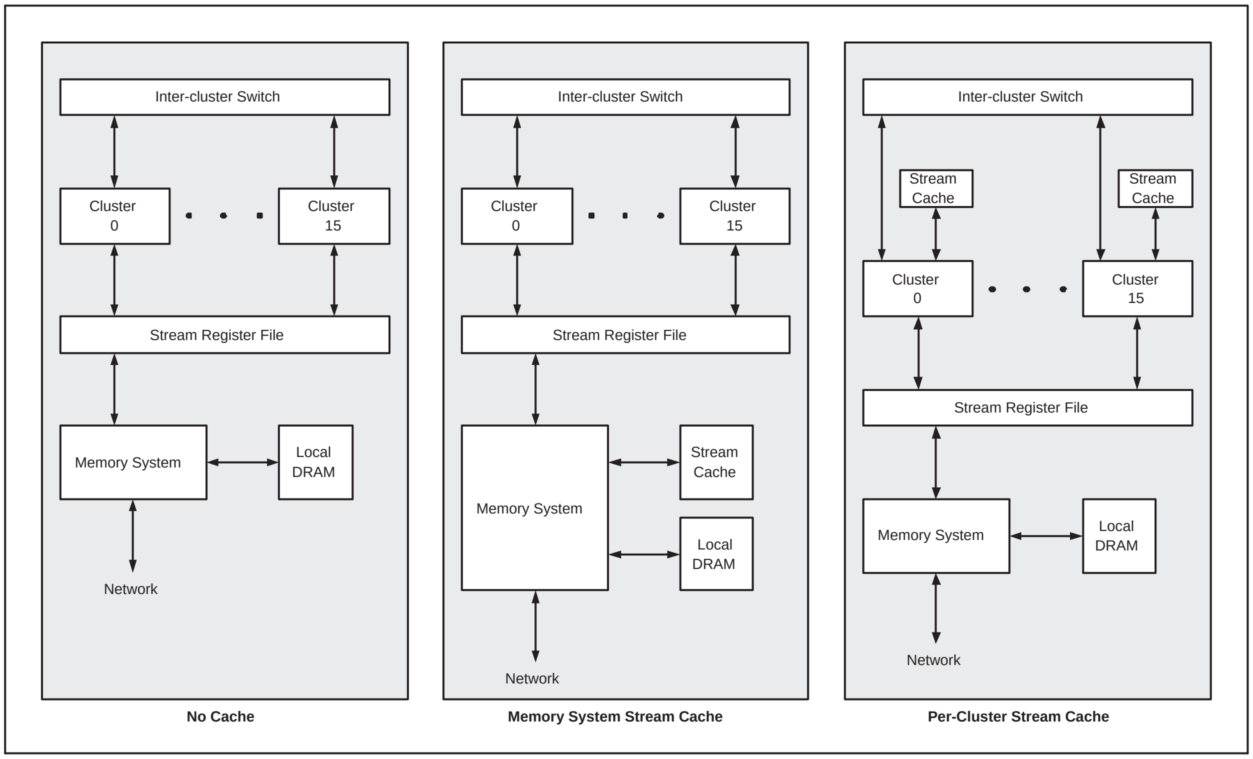
May 5, 2002

Contents

1 Introduction

This project examines various stream cache architectures for enabling the efficient implementation of irregular stream applications.

2 Architectures Evaluated



5 Architectures to be Evaluated:

1. Baseline - no stream cache.
2. Memory system cache (non-coherent).
3. Memory system cache (partially coherent).
4. Cluster cache (non-coherent, read-only, miss-fail).
5. Cluster cache (non-coherent, read-only, miss-stall).

Add-and-Store:

Each architecture is evaluated with and without hardware add-and-store support for application 2.

Parameters Varied in Each Architecture Evaluated:

- cache organization
- cache access latency
- cache access throughput
- cache size
- number of nodes

Parameters Kept Constant Across All Experiments:

- number of clusters
- DRAM size
- SRF size
- SRF to MS bandwidth
- SRF to cluster bandwidth
- inter-cluster switch bandwidth
- network bandwidth

Figure 1: Architectures evaluated

Figure 1 illustrates the 5 cache architectures evaluated. This section describes each of the architectural features employed.

2.1 Non-coherent Memory System Stream Cache

The memory system stream cache, when used, may be non-coherent. The following are salient points regarding this implementation of the stream cache.

- The scalar program has explicit control of the stream cache; prefetches, loads, invalidates, and flushes must all be initiated by stream instructions.
- The stream cache is able to cache both local and remote addresses.
- Stream cache elements can be read-only or write-back. Read-only elements, which are either *ValidA* or *ValidB*, can be gang-invalidated.
- The hardware doesn't provide any coherence support; stream cache entries which are not read-only are write-back, and if multiple nodes are writing to the same value, then they will simply have different copies of the same data in their stream caches.
- When a dirty stream cache entry is flushed, it is written back to memory. If multiple nodes are all updating the same memory location, then the final value after all the updates have completed is non-deterministic.
- The segment registers contain a bit which can override requests to cache data from that segment.
- An add-and-store operation always goes to memory, regardless of whether the destination address is cached.

2.2 Partially Coherent Memory System Stream Cache

The memory system stream cache may also be partially coherent, via the following memory locking mechanism. Instead of issuing READ and WRITE requests to the memory system, each node will issue either:

- READ-LOCK (RL)
- UNLOCK (U)
- READ-WITH-INTENT-TO-WRITE (RIW)
- WRITE-UNLOCK (WU)

Each memory address can be considered an instantiation of the state machine illustrated in figure 2.

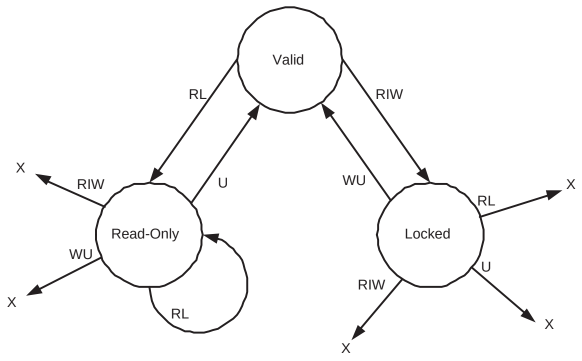


Figure 2: Memory element finite state machine

2.3 Non-coherent Read-Only Miss-Fail Cluster Stream Cache

Each cluster may have a private stream cache with the following features:

- There is no hardware support for coherency in these caches, either within a node or between nodes.
- The caches are read-only all the time; new entries can be loaded, but existing entries can't be updated.
- The caches must be explicitly loaded (from the SRF) and invalidated by the microcontroller.
- The assumption is made that the caches never miss; the cache read returns an undefined value in the event of a cache miss.

2.4 Non-coherent Read-Only Miss-Stall Cluster Stream Cache

This incarnation of the cluster cache differs from the miss-fail version in the handling of cache misses. In the event of a miss in the miss-stall cluster cache architecture, the following happens:

- The microcontroller stalls all the clusters.
- The microcontroller branches to a 'cache load' subroutine.
- Each cluster cache outputs signals describing whether or not the last access was a miss, and if so, what address was needed.
- The microcontroller checks all the caches which didn't miss in parallel to see if any have the address which was missed; if it is found, it is copied to the cache which needs it. The communication occurs over the inter-cluster switch. For there to not be a conflict with an inter-cluster communication which was being prepared when the miss occurred, there must be some local registers on the communication port which are reserved for cache miss handling.
- If the address wasn't found in another cache it is fetched from memory, via the SRF with an index stream of length one.
- The cache is updated with an LRU policy.

2.5 Add-and-Store

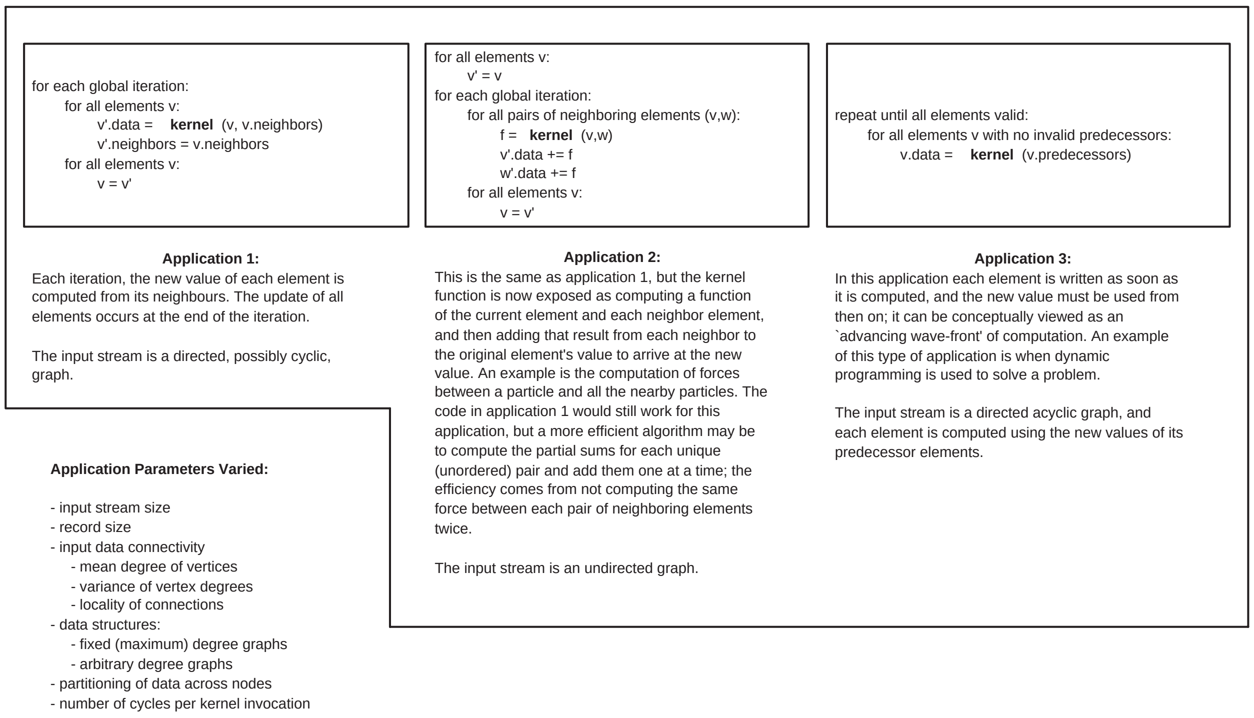
The hardware can support an add-and-store operation, in which a node can send a value to be added to a global memory location atomically. The implementation is as follows:

- The clusters output a series of *(VirtAddr, PartialSum)* tuples which are buffered in the SRF. Physically, these may be stored in the SRF as two different streams: one of addresses, and the other of the partial sums.
- The stream controller issues an instruction to read the buffer of partial sums from the SRF and send add-and-store operations for each value to the appropriate address.
- Upon reception of an add-and-store request, the memory-network interface passes the partial sum to the memory system, which adds it to the specified address.

Note that the software is responsible for the ordering of the partial sum updates across nodes. (i.e.) If multiple nodes are all computing partial sums to add to the same memory location, the software is responsible for ensuring that the partial sums from different sources are added together and to the destination memory address in a deterministic order, if that is a feature the application requires. The issue is that due to rounding errors, adding the partial sums in different orders results in different final values.

3 Applications Using Irregular Streams

Figure 3 lists 3 applications which use irregular streams. These applications will be implemented for each of the architectures modeled.



```
for each global iteration:
    for all elements v:
        v'.data = kernel (v, v.neighbors)
        v'.neighbors = v.neighbors
    for all elements v:
        v = v'
```

Application 1:
Each iteration, the new value of each element is computed from its neighbours. The update of all elements occurs at the end of the iteration.

The input stream is a directed, possibly cyclic, graph.

```
for all elements v:
    v' = v
for each global iteration:
    for all pairs of neighboring elements (v,w):
        f = kernel (v,w)
        v'.data += f
        w'.data += f
    for all elements v:
        v = v'
```

Application 2:
This is the same as application 1, but the kernel function is now exposed as computing a function of the current element and each neighbor element, and then adding that result from each neighbor to the original element's value to arrive at the new value. An example is the computation of forces between a particle and all the nearby particles. The code in application 1 would still work for this application, but a more efficient algorithm may be to compute the partial sums for each unique (unordered) pair and add them one at a time; the efficiency comes from not computing the same force between each pair of neighboring elements twice.

The input stream is an undirected graph.

```
repeat until all elements valid:
    for all elements v with no invalid predecessors:
        v.data = kernel (v.predecessors)
```

Application 3:
In this application each element is written as soon as it is computed, and the new value must be used from then on; it can be conceptually viewed as an `advancing wave-front' of computation. An example of this type of application is when dynamic programming is used to solve a problem.

The input stream is a directed acyclic graph, and each element is computed using the new values of its predecessor elements.

Application Parameters Varied:

- input stream size
- record size
- input data connectivity
    - mean degree of vertices
    - variance of vertex degrees
    - locality of connections
- data structures:
    - fixed (maximum) degree graphs
    - arbitrary degree graphs
- partitioning of data across nodes
- number of cycles per kernel invocation

Figure 3: Irregular stream applications evaluated