Stream Processing: Hardware and Software Discussion

Lecture #2:	Thursday, 9 April 2002
Lecturer:	Prof. Bill Dally
Scribe:	John Davis and Andrew Lin
Reviewer:	Mattan Erez

Logistics:

- Reading is required.
- Discussion is required.
- Please have nameplates.
- Everyone must scribe. Due to the size of the class, multiple people will scribe per lecture/discussion session.

2 handouts:

- Beginner's Guide to Imagine Application Programming
- Imagine Programming System User's Guide

Imagine programming assignment on 4/16/2002

Class Format:

- Stream Processor Architecture lecture postponed until later.
- Discussion of two papers listed below.

Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Jin Namkoong, John D. Owens, Brian Towles, and Andrew Chang. "*Imagine: Media Processing with Streams.*" IEEE Micro, Mar/April 2001.

This paper provides a high-level overview of the Imagine processor using stereo depth extraction as a motivating software example. This paper discusses the fundamentals of the Imagine Stream Architecture and programming model along with some projected VLSI implementation details.

John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. "*Polygon Rendering on a Stream Architecture.*" 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware, August 2000. pp. 23-32. ACM SIGGRAPH / Eurographics / ACM Press.

This paper discusses polygon rendering implemented on the Imagine processor. The authors compared the simulated performance of the Imagine processor to specialized dedicated hardware, the Nvidia Quadro graphics card and an extrapolated Nvidia graphics card, and a software implementation, opengl32.dll. This paper suggests hardware extensions to the Imagine processor that would boost performance for particular phases of the polygon rendering pipeline.

1 Discussion: Imagine: Media Processing with Streams

1.1 How are stream instructions and their dependencies handled?

There are 32 stream instructions, which can be written to the stream controller by the host processor. The stream controller acts as a score board, dynamically tracking dependencies using a bit field associated with the instructions. These dependencies are identified at run-time by the host processor and communicated along with the stream instructions to the stream controller. Figure 1 illustrates the stream controller.

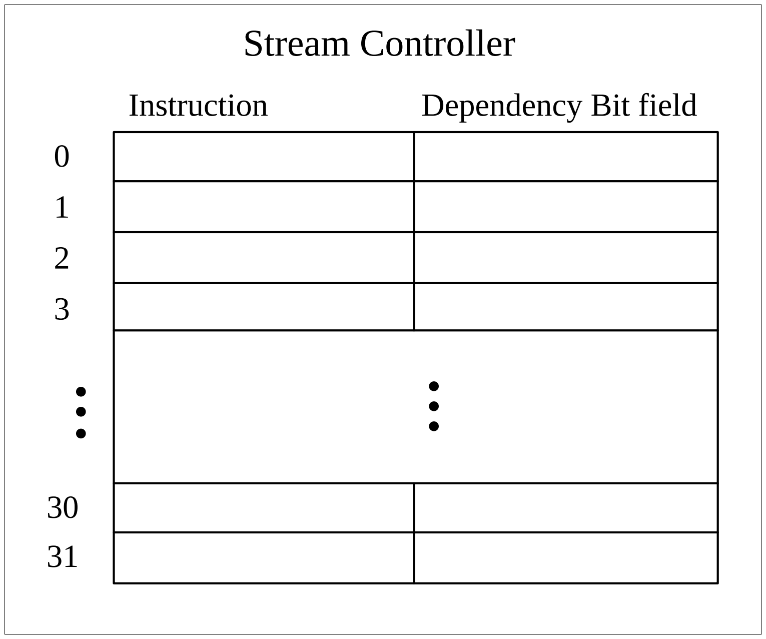

Figure 1: Stream controller contains the instruction and an associated bit field that tracks instruction dependencies at run-time, much like score boarding or other related resource management techniques.

1.2 Can you have different data types in a stream?

The streams can be composed of heterogeneous data types, which lead to conditional streams. Imagine's clusters operate in a SIMD fashion so all the cluster must be working on the same data type at the same time.

1.2.1 Conditionals:

- **Prediction**: This is the normal flavor of prediction. A branch condition using history or other methods determines whether or not a branch is taken. Misprediction, if it was possible, is very expensive in the Imagine processor. Kernels only operate on loops so there is very limited control flow in the clusters. The pipeline is 6 instructions deep and there are 64 execution units, which means there are about 300 operations in flight at any time. If a branch misprediction were to occur, then all the execution units must be flushed. There are a lot of situations in programs that really are not predicted, loop bounds, exceptions like divide by 0, etc.

- **Predication**: This scenario has a test associated with an instruction. It converts control dependencies to data dependencies. Instructions executed based on a predicate (guard or condition) and if the condition is false then there is no state change, (data write, etc.). The form of predication actually used in Imagine is a select operation. This is basically predication without hardware support, where both sides of the branch are executed into different registers and then the appropriate results are selected in software (copied into the result registers).

There are 2 approaches to dealing with streams composed of multiple data types: sorting and predication. Figure 2 illustrates these two approaches with a stream containing circles and squares. The upper diagram illustrates the use of a sorting kernel combined with additional kernels to process the circles and squares. While the lower diagram in Figure 2 uses a predicated kernel to operate on the different components of the stream.

When considering the appropriate method for dealing with streams, the overhead associated with the different operations can dramatically affect the performance. Adding a conditional sorting kernel may minimize overhead while using a predicated stream may only operate on a portion of the original stream. The opposite may be true as well.

1.3 In the convolve example, how is the shared data of the partials managed?

The kernel processes the new row and old partials and outputs a filtered row and new partials. The SRF, LRF, and scratchpad are used to manage the different data needed for the convolve. Figure 3 shows the inputs and outputs of the kernel as well as the transition of new partials to old partials. The ”old partials” stream is only used in the next invocation of the kernel. Temporary values that are used within the same invocation are stored in the LRF or scratchpad.

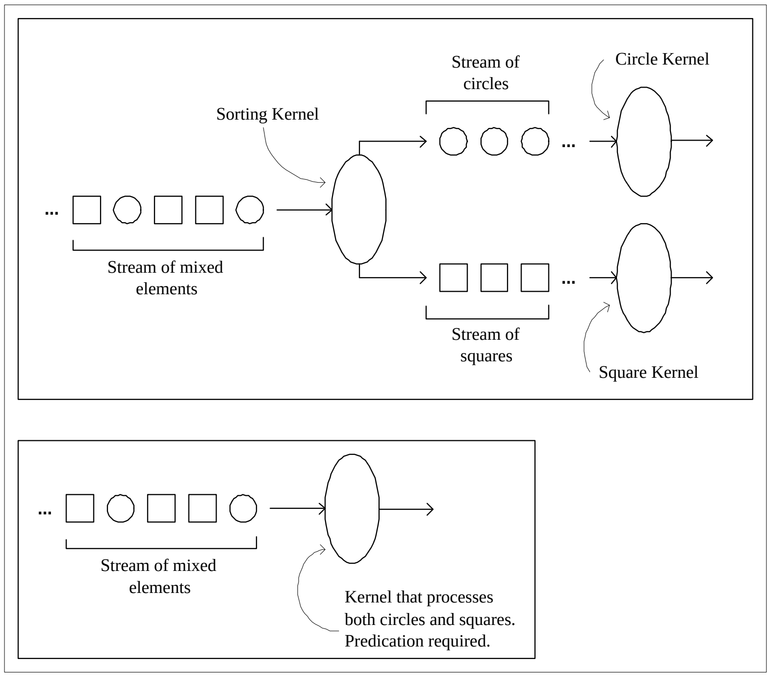

Figure 2: Sorting vs. Predicated stream processing.

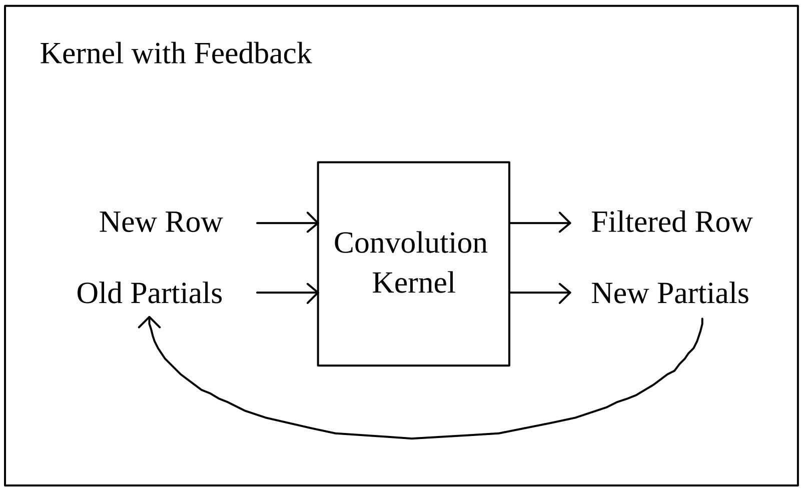

Figure 3: Convolve kernel with input and output streams.

1.4 How do instructions migrate from DRAM to the microcontroller?

1.5 How is memory access scheduling done?

It enables the reordering of DRAM operations, which improves memory bandwidth by facilitating ”good” DRAM access patterns. Streams are not necessarily contiguous in DRAM, but must be in the SRF. Because access time is dependent on the DRAM address, memory access

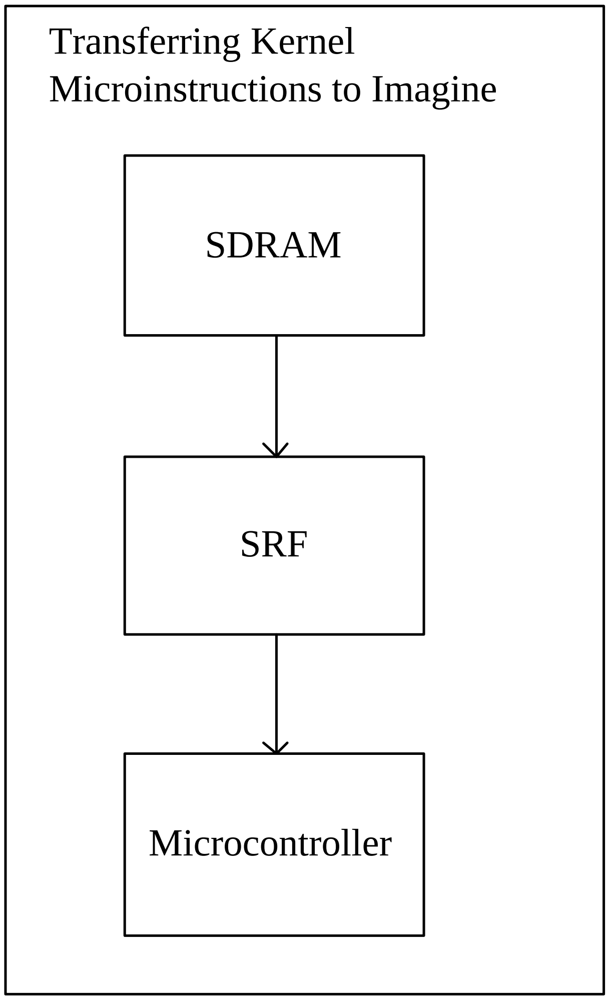

Figure 4: Moving instructions from DRAM to the microcontroller.

Instructions are stored in DRAM and read into the SRF. There are two types of instructions, micro instructions, 576 bits wide that control the ALU clusters, and the stream ops: load, store, send, receive, cluster op and load microcode (from SRF to microcontroller store). Figure 4 shows the migration of microcode instructions from DRAM to the microcontroller store. Stream operations are handled by the host procssor and come out of its memory. The host than issues the appropriate instruction (with its dependencies) to the stream controller.

scheduling can optimize the DRAM by enabling out-of-order DRAM requests to build streams. Once the requests are filled the data is gathered into a contiguous stream and placed in the SRF. You basically increase the latency of some words in order to achieve higher throughput (and you only care about throughput since latency is hidden).

More information on memory access scheduling can be found in this paper: Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. ”*Memory Access Scheduling*.” In Proceedings of the 27th Internation Symposium on Computer Architecture (June 2000). ftp://cva.stanford.edu/pub/publications/mas.pdf

1.6 Is there a high-speed interconnect between the clusters?

There is a high-speed switch that enables intercluster communication, which is part of the ALU Cluster, see slide 32 of the Stream Processor Architecture lecture. Any cluster can read or write a single word from or to any other cluster. Thus, 8 words can be interchanged between the clusters. There is also the instruction broadcast to all the clusters of the 576-bit long instruction word. For algorithms that require more than one word to be communicated between the clusters in a single cycle, the streams and related communication must be staged in the SRF. In order to increase the bandwidth of the high-speed intercluster switch, a faster clock cycle could be used to communicate many more words in one Imagine clock cycle or more memory with multiple ports can be used. The latter incurring a major chip area penalty, while both increase the complexity.

1.7 Could adding another level of memory on the Imagine processor be beneficial?

DRAM density is 5-6 times greater than SRAM. However, you must add a refresh mechanism. The design approach for the SRF was to keep it simple. The goal of adding more memory to the chip would be to increase its effective memory bandwidth. Currently, the SRF is clocked at half the speed of the system clock with no degradation in performance. In this case, bandwidth not latency, is important! It is not clear how to utilize the additional memory. Some suggestions were made: larger SRF, L2. Streams don't use the SRF like a cache because there is no temporal locality, thus hit rate is not a good measure of performance. If the SRF was thought of as a cache, then it would have a 100% hit rate.

1.8 Why does the SRF have high bandwidth?

It reads wide and full lines at a time. Furthermore, there are 8 lanes of SRF and each one has a one-to-one mapping with the ALU Clusters. Thus there is no switch between the SRF and ALU cluster making the communication very fast.

Additional levels of memory hierarchy may extend the memory bandwidth, but the costs and benefits of this extension should be carefully examined. This was suggested as a good project idea. Figure 5 illustrates the expanded memory hierarchy.

1.9 What about adding a stream engine on a commodity processor?

Intel was headed in this direction for a while, but dropped the idea. That seems like a good combination as a co-processor or more like a vector machine with a vector processor and a scalar process.

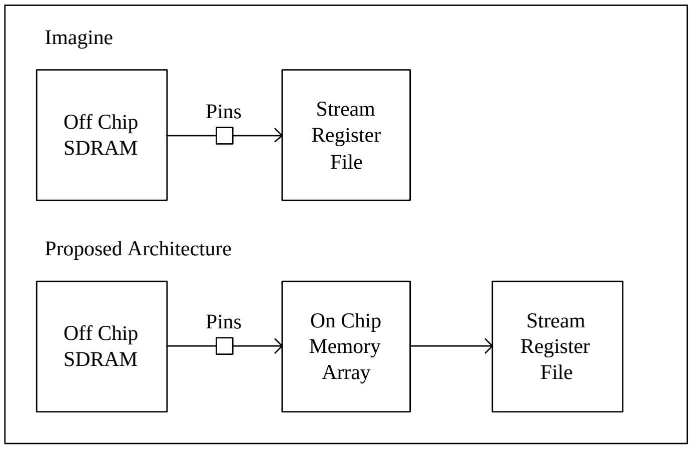

Figure 5: Possible expanded memory hierarchy of the Imagine processor.

1.10 Why mesh (tightly couple) scalar and stream processors?

This reduces the cycle difference in communication between the processors which could increase bandwidth by reducing the setup and tear down overhead of small streams. The stream length dramatically affects the performance of the processor. As the Figure 6, below, shows, tightly coupling the scalar and vector processors can reduce the critical stream length thereby improving the performance on smaller streams and enabling more diversity in the set of kernels. Prof. Dally defined the critical stream length as 3dB down from the high performance plateau.

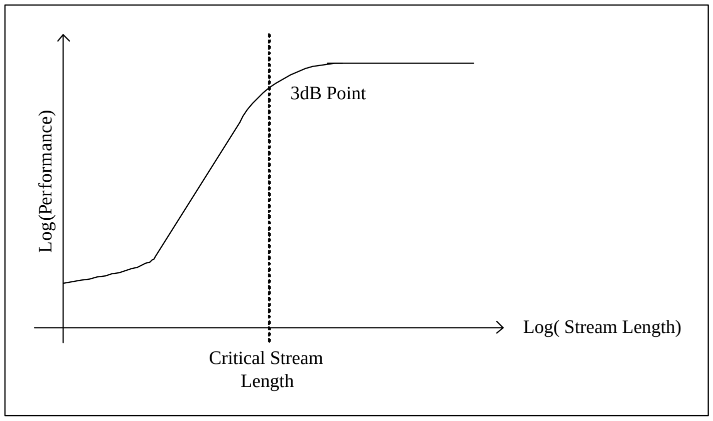

Figure 6: Performance vs. stream length

1.11 How does Imagine accomplish instruction bandwidth reduction?

Imagine's host processor fetches instructions from the instruction cache and then distributes them according to the diagram shown in Figure 7. The host processor which executes all scalar instructions and is responsible for sending all stream instructions (memory and kernel) to the Imagine processor's stream controller unit. The increased instruction bandwidth comes from the two sources: first, stream instructions encapsulate the executions of entire kernels or stream-memory ops (instead of having a single load per data item or ALU operation); second, within the Imagine processor a single instruction is used for all eight clusters in SIMD (instead of one instruction per cluster).

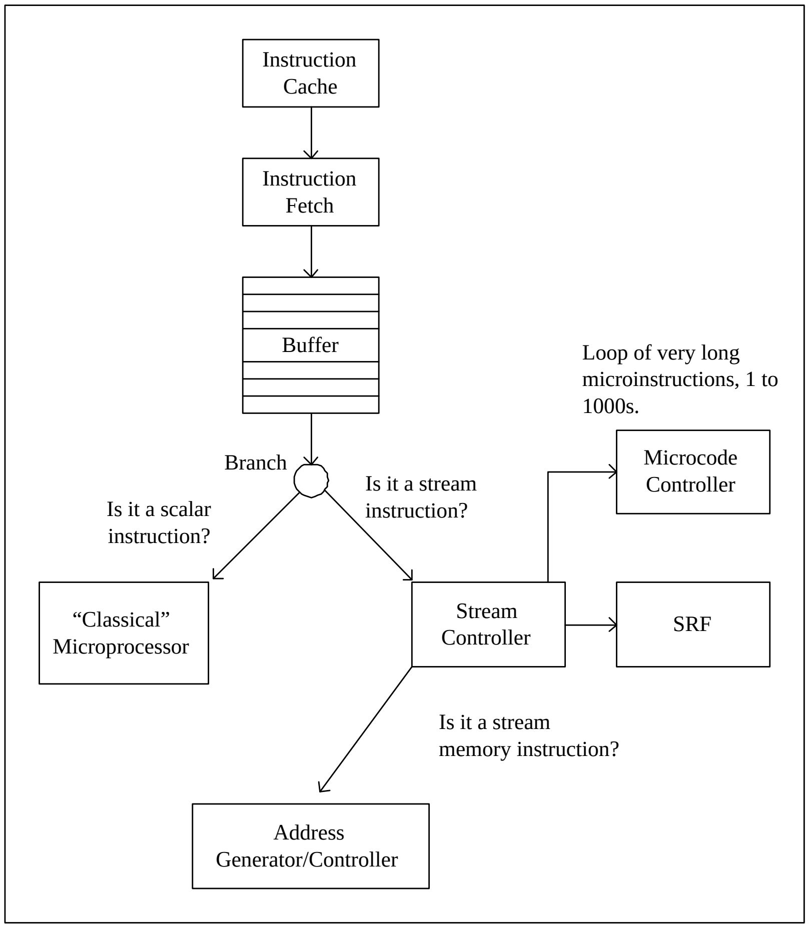
Instruction Cache
Instruction Fetch
Buffer
Loop of very long microinstructions, 1 to 1000s.
Branch
Is it a stream instruction?
Is it a scalar instruction?
Microcode Controller
"Classical" Microprocessor
Stream Controller
SRF
Is it a stream memory instruction?
Address Generator/Controller

Figure 7: Instruction Processing flow diagram.

1.12 SIMD vs. Streaming comparison

- Explicit memory hierarchy vs streaming model $\rightarrow$ dataflow (saves memory bandwidth)
- temporal/spatial vs. producer/consumer locality
- Cache (Data fetched reactive) vs. SRF (Data fetched proactively, explicit)
- Record order (vector specific, not SIMD) vs. operation order (stream)

Pointers always can cause a problem: pointer chasing → scatter/gather issues, pointer deferencing.

1.13 Is life better with a cache?

There is lower overhead if there is reuse; it is better to bypass the cache if there is no reuse.

1.14 Is memory management an issue for special purpose processors?

Parallelism is needed to cover up the latency. However storage is always needed for the working set. A FIFO or other simple memory structure may suffice.

2 Discussion: Polygon Rendering on a Stream Architecture

This paper is used as a case study of a streaming application. The topics covered in this discussion include batching, performance bottlenecks, and out-of-order completion.

2.1 The Batching Problem

Large data sets will not fit in the SRF. To handle this, the data sets (in this case, the input primitives) are partitioned into batches, and an entire batch is processed at one time by the stream processor. In this case study, the size of the batch is chosen so that all active streams fit in the SRF. That way, consumer-producer locality is maintained, since kernels pass streams to each other using the SRF.

The problem lies in the size of intermediate batches. For most programs, the sizes of the intermediate batches can be predicted. However, the intermediates are difficult to predict for polygon rendering because triangles don't generate predictable intermediates. Thus, the intermediate streams may not fit in the SRF, resulting in an overflow. This is a very difficult static scheduling problem.

2.1.1 How do you handle this?

One thing we can do is to reduce the size of the input batch so that all intermediate data can be stored in the SRF. However, as discussed before, there is a lot of overhead involved in using small streams. We want to maximize stream length to reduce startup and teardown costs, but don't want the stream size to result in an overflow of the SRF.

Another, more direct, way to handle the overflow problem is to, well, overflow. Simply stop the kernel, and save state. However, this involves a lot of overhead — all intermediate variables must be saved, and the data in the LRFs may have to be saved as well. This can be quite painful.

Another solution is to switch kernels. When an overflow is detected, switch kernels to consume data out of the SRF. Then, switch back and use the space now available in the SRF.

The problem with doing this is that, when we switch kernels, the ”new” kernel can also generate intermediate data and streams, resulting in overflow. In addition, when a kernel switch occurs, the state of the ”old” kernel must be saved, resulting in operational overhead.

An escape buffer can be used to potentially handle overflow. A portion of the SRF is reserved for overflow. When a kernel overflows, the data is stored in this portion of the SRF. This is good, since producer-consumer locality is maintained. The question then becomes, how much space in the SRF should be sacrificed for this escape buffer?

Double Buffering is what’s usually done in Imagine when we don’t want to decrease the strip-size (batch size) too much. This is similar to an escape buffer. It simply means that each stream is broken into two buffers in the SRF and while one is being processed the second half is being read/written to memory.

2.2 Performance Bottlenecks

The Imagine processor performed worse than special purpose hardware in fill-intensive datasets. The paper states several reasons for this. It is important to note that the memory is not the primary bottleneck. Rather, it is the amount of computation that slows down the processor, particularly in rasterization. The NVIDIA system, with its dedicated rasterization hardware, is expected to excel in this application. The other performance bottleneck is the overhead expended from the use of short streams. Shorter streams are less efficient because the amount of time consumed in processing the stream can be comparable to the amount of time required to setup and teardown the required support data structures. A discussion on the implications of stream size on performance was presented previously.

2.2.1 What can be done to improve the performance of Imagine with respect to specialized hardware systems, such as the NVIDIA system?

Building a dedicated graphics engine can be used. In this case, if the Imagine processor has a separate, specialized cluster that performs rasterization operations, then the performance gap would, theoretically, be closed. However, this would really hurt the programmability and flexibility of the processor (How many applications would require a rasterization engine?)

Another suggested solution would be to include a reconfigurable hardware cluster (a reconfigurable datapath). This would allow the implementation of specialized datapath hardware without sacrificing programmability and flexibility. The question then becomes, how granular should the reconfigurable hardware be? If the hardware was reconfigurable at the gate level (like an FPGA), the performance and area costs may be too high. Better performance may be achieved if the hardware consisted of functional blocks that could be pushed around to create a custom datapath.

2.3 Out-of-Order Completion

Some applications require that the sequential order of data must be preserved. In this case study, input primitives must be completed in order. Here, some triangles are small, and some of them are big, requiring different computation times. Since eight clusters process the triangles in parallel, the fragments from the rasterization stage complete out of order. How is the reordering problem handled?

A unique ID is assigned to each triangle in the batch. This ID is passed to the fragments generated by the rasterization stage. A hash is performed on the coordinates of the fragments to identify the conflicting fragments. Once the conflicting fragments are identified, they can be sorted in ID order (the fragment with the lower ID gets handled first). In essence, elements without hash collisions are handled in parallel. Elements that collide are handled in order.

Two streams are generated — a stream of unique fragments, which have no conflicts with other fragments, and a stream of conflicting fragments, which is sorted and appended to the stream of unique fragments. The final single stream is thus ordered, as shown in Figure 8 below. The complexity of the operation should also be considered. Hashing is O(n) whereas sorting is O(nlogn) so you want to sort as little as possible. Also, you can hash as you go while sorting is more complex.

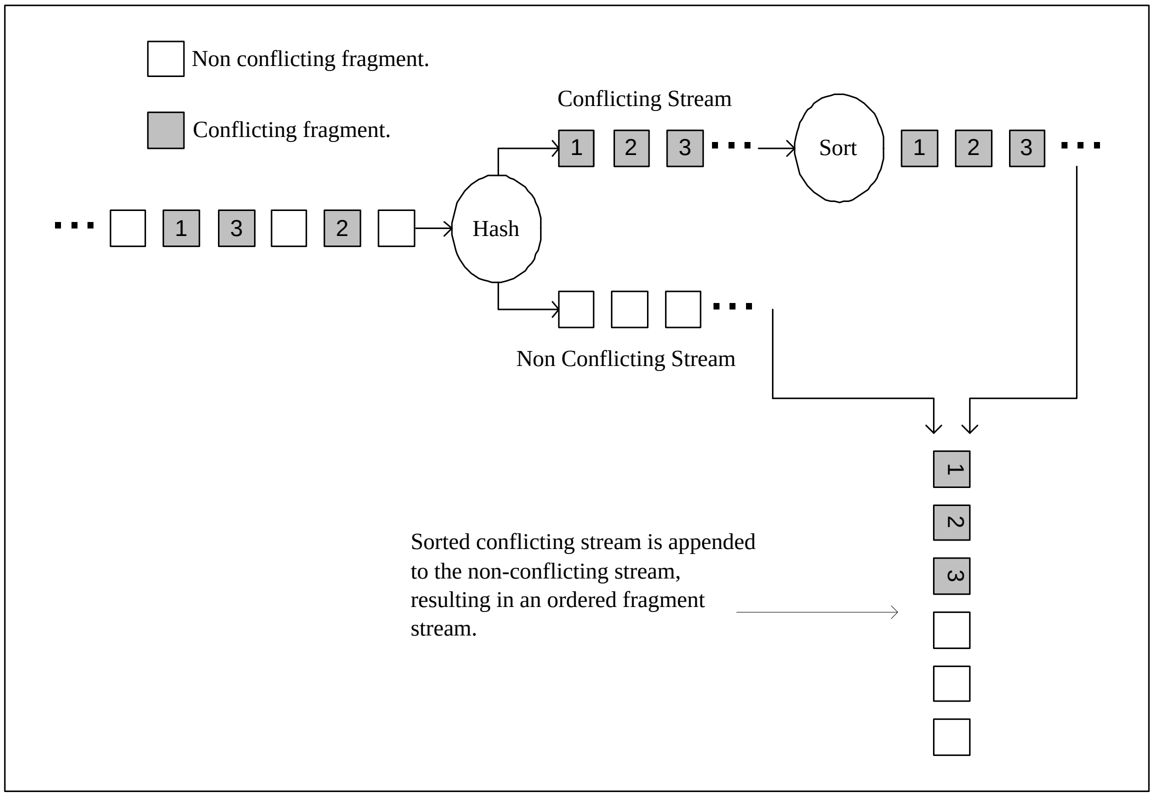
Non conflicting fragment.
Conflicting fragment.
Conflicting Stream
Sort
Hash
1
2
3
Non Conflicting Stream
Sorted conflicting stream is appended to the non-conflicting stream, resulting in an ordered fragment stream.

Figure 8: Sorting the fragments.

3 Project Ideas:

- **Extend bandwidth hierarchy**
- **Solutions for SRF overflow**