EE482C Project:

Stream Cache Architectures for Irregular Stream Applications

Timothy Knight Arjun Singh Jung Ho Ahn

June 6, 2002

1 Introduction

Irregular stream applications, such as programs which traverse an arbitrary graph data set, don't follow the 'traditional' stream model that Imagine was designed to exploit, in which records are gathered into the SRF in a predefined order (often sequentially or on a fixed stride) and then operated on by a kernel. Index streams can be used to traverse an irregular structure, but several inefficiencies may result:

- Irregular stream records may contain many pointers to other records, and a large percentage of these may be null, wasting SRF space if the entire record is read from memory.
- If a record is needed several times during the course of a kernel, for example if it is a neighbour of multiple other records, it may be read from memory repeatedly, wasting memory bandwidth.
- A record which is read multiple times in one or more index loads will also be replicated in the SRF.
- The overhead of doing pointer chasing using index streams will increase as the depth of the pointer chain increases, as for each level an indexed load from the memory to the SRF is needed, and if any pointer arithmetic is required a kernel must run between loads to generate the addresses.

This project examines various stream cache architectures for reducing the memory bandwidth wasted in repeated loads of words from memory. A secondary consequence of the work presented in this document is that a cluster cache system can also reduce the SRF space wasted due to both null pointer storage and data replication.

2 Irregular Stream Data Structure

In all of the experiments in this project, the following is the record definition used.

```
record {
  float data_0, ..., data_{D-1};
  float newdata_0, ..., newdata_{D-1};
  int numneigh;
  pointer neigh_0, ..., neigh_{M-1};
}
```

It is assumed that the records are laid out sequentially in memory, and that each node, in a multi-node simulation, has the same number of records.

3 Applications

The two applications described in this section were used to evaluate the architectures examined in this project.

3.1 Application 1: All updates committed after all computations are done

Application 1 is simply a graph traversal application in which the ‘new’ value for each vertex is computed from the ‘current’ values of itself and its neighbouring vertices. The pseudocode for application 1 is as follows.

```
// Computation phase
for each vertex v:
  v.newdata$_*$ = kernel (v.data$_*$, v.neigh$_*$.data$_*$)

// Update phase
for each vertex v:
  v.data$_*$ = v.newdata$_*$
```

3.2 Application 2: Updates committed as soon as they are computed

Application 2, which was chosen to create problems with the non-coherent cache architectures we evaluated, can be conceptually viewed as an advancing wavefront of computation in a DAG; the data value for each vertex is computed from the updated values of its predecessor vertices. The pseudocode for application 2 is as follows.

```
repeat until all vertices are updated:
  for each vertex v with valid predecessors:
    v.data$_*$ = kernel ( v.neigh$_*$.data$_*$)
```

Note that for application 2, the `newdata` fields in the record definition are not used and the `neigh` fields can be interpreted as `predecessor` pointers.

4 Stream Cache Architectures

The four stream cache architectures that were evaluated are illustrated in figure 1, along with the corresponding bandwidth and storage hierarchies. This section describes the specific choices that were made for each architecture.

4.1 Architecture 1: No stream cache (baseline)

Architecture 1 is essentially the currently proposed SSS architecture without the stream cache. Its relevant specifications are as follows.

- 16 clusters.
- A single 256 kword SRF divided into 16 banks.
- 2 AG/ROB pairs, each of which can independantly support up to 4 words per cycle between the memory system and the SRF for both strided and indexed memory addressing.
- 8 DRAM banks with a combined sustained throughput of 1 word per cycle (random access) and up to 2 words per cycle (sequential). In our simulator we have an ‘amortized’ memory model in which we assume a constant sustained bandwidth, regardless of

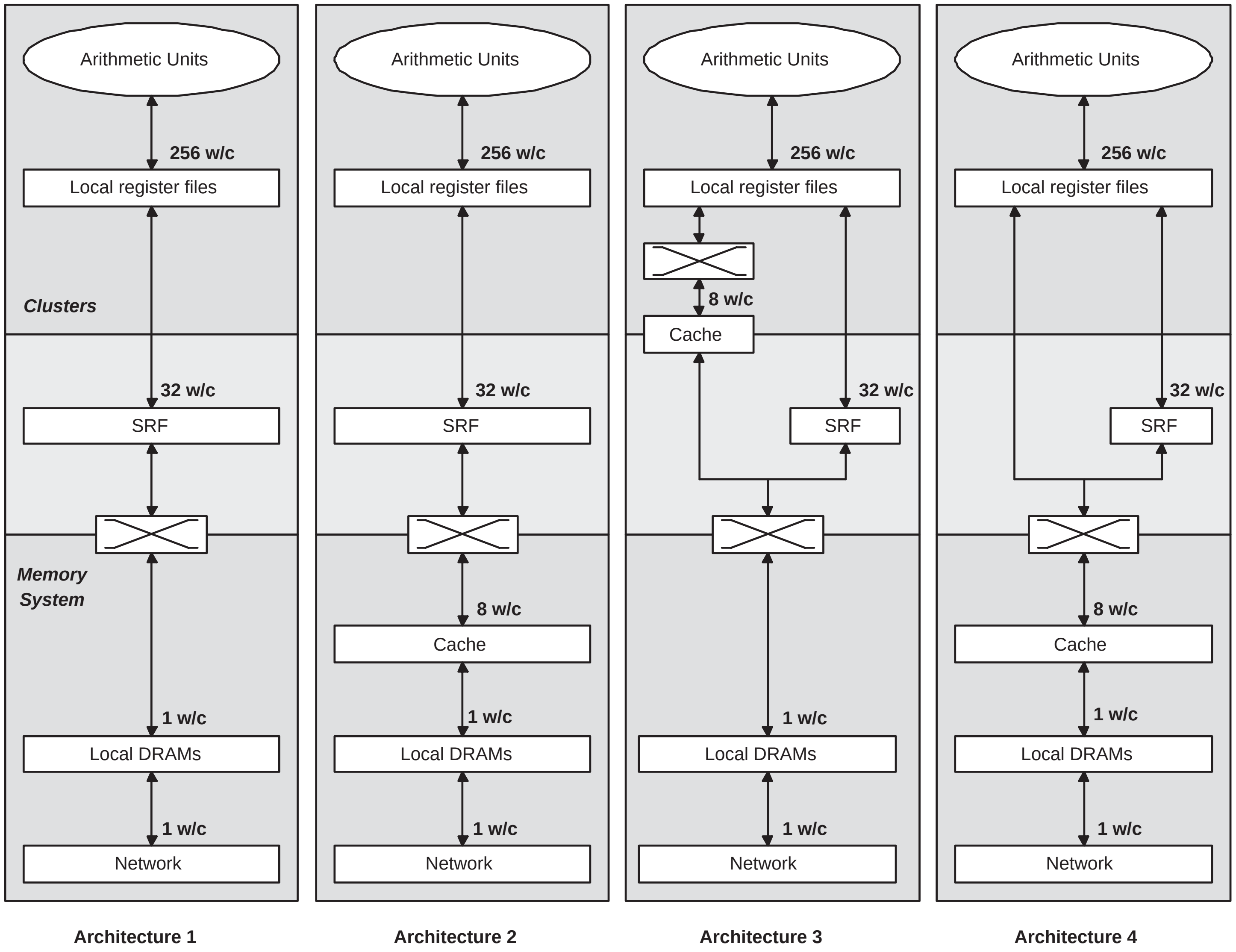


Figure 1: Bandwidth and Storage Hierarchies

whether the access is random or sequential. As the large majority of memory references in our applications are substantially random this approximation is adequate.

- A flat network between nodes with a sustained throughput of 1 word per cycle.

4.2 Architecture 2: Memory system stream cache

Architecture 2 extends architecture 1 by placing a cache in the memory system between the local DRAM and network interface and the SRF. The specifications of the memory-side cache are as follows.

- The cache is explicitly controlled by the software: prefetches, loads, invalidates, and flushes must all be initiated by stream instructions. Load and store instructions contain a parameter which specifies whether the data should go through the cache or directly to memory.

- The cache allows data to be marked as both read-only and read-write. Read-write data is handled with a write-back policy, and there is no hardware coherence support; the software must be aware of this limitation and act accordingly.

- The stream cache is able to cache both local and remote addresses. If multiple nodes are flushing their cache and try to update the same memory location, then the final value after all updates have completed is non-deterministic.

- The cache has 8 banks and supports a maximum throughput of 8 words per cycle. In the presence of bank conflicts, however, the throughput may decrease.

- The cache is S-way set associative, with S varying between 1, 2, 4, and 8 in our experiments, and has 1 word per line.

- The cache doesn't handle 'second misses' gracefully; if a request misses in the cache it is sent to memory to be satisfied, and if another request for the same address arrives while the first is still being handled, the second request is also sent to memory.

  We did simulate a slightly more advanced scheme, in which the second request is stored in the cache until the first miss completes, at which time both requests are satisfied, changing the problem to that of 'third miss'. The performance improvement using this, however, was quite small due to the fact that even though the second miss was being caught, the several subsequent requests for the same address would all still go to memory.

4.3 Architecture 3: Cluster stream cache

Architecture 3 extends architecture 1 by giving the 16 clusters access to a single memory cache, as illustrated in figure 2. The specifications of the cluster-side cache (size, policy, etc.) are substantially the same as for the memory-side cache used in architecture 2; the following is a description of how the cache is used by the clusters.

- The cache is explicitly controlled by the software, with loads originating from the kernel program and prefetches and invalidates originating from the stream program.

- Each time the SIMD kernel program executing on the clusters issues a load instruction a row of the request FIFO is written with each cluster's requested address. This FIFO is drained by the cluster cache at a rate of up to 8 words per cycle, enabling the clusters to have an issue rate of at most one load instruction every 2 cycles.

- Each cycle the cache drains some of the requests at the front of the FIFO and checks its tags for

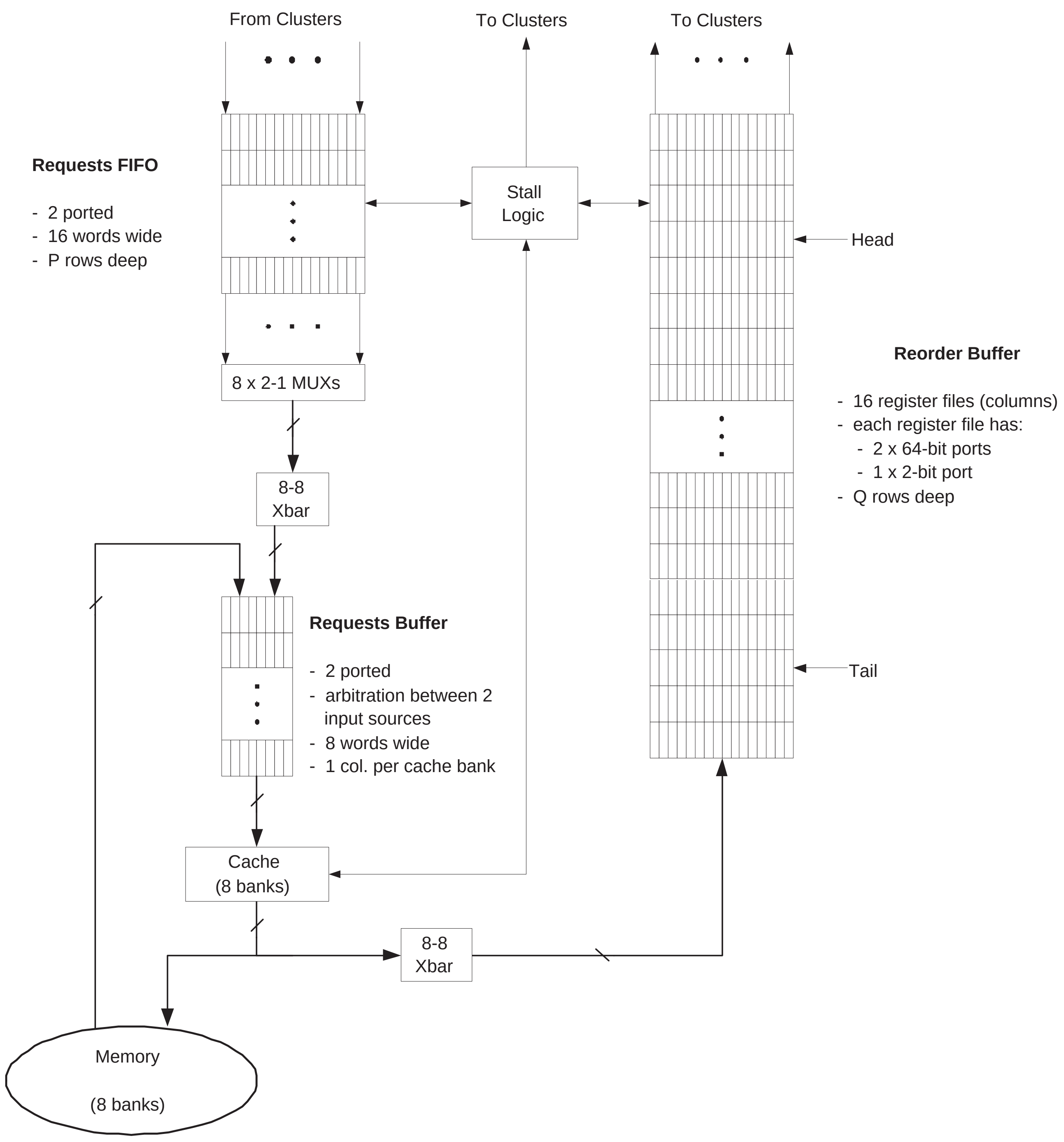


Figure 2: Cluster Cache Architecture

hits and misses; all misses are sent to memory. By maintaining a small number of pending requests it can reduce the number of bank conflicts. Each cycle, the cache writes up to 8 words to the reorder buffer. Each word written is either the data resulting from a cache hit, or a marker indicating that the cache missed and the memory is fetching the data.

- When a word of data is returned by the memory system it is inserted into the appropriate location in the reorder buffer.
- The front row of 16 words in the reorder buffer is popped at most every second cycle and copied back to the clusters.
- From the perspective of the clusters, which operate in a statically scheduled SIMD fashion, the cache must provide a constant latency, regardless of whether it hit or missed. This is done by ensuring that words stay in the reorder buffer for the appropriate amount of time before they get to the front and are popped; effectively, the number of rows of the reorder buffer is the fixed latency that the clusters expect. In the event that the front row of the reorder buffer contains an 'unready' marker, the clusters are stalled until the data returns from memory and is placed in the reorder buffer, providing the appearance to the kernel program of a constant latency.
- The stall logic handles the cases in which the constant latency appearence may be broken, such as lots of bank conflicts in the cache effectively causing the rate at which the reorder buffer is filled to decrease, and requests for memory addresses not having returned yet.
- The cache supports a predicated load feature to handle the SIMD nature of the architecture. Whenever the kernel program loads from the cache, all 16 clusters issue an address, even if they don't really want one. The address requests sent to the cache from the clusters can be flagged as 'invalid', in which case the requests effectively 'hit', and an undefined value is returned to that cluster when the result of the load is sent.

In our experiments, we varied the length of the reorder buffer between the two extremes, which are, at the minimum, a length corresponding to the actual cache hit latency, and, at the maximum, long enough to completely hide the expected worst-case memory latency (not including the infrequent higher latencies resulting from the unbounded and unpredictable network), ensuring that the clusters need (almost) never stall. The tradeoff is between having a small hit latency but high miss cost (due to stalling), in the case of a short ROB, and a high hit latency but smaller miss cost in the case of a long ROB. An additional advantage of the long ROB is that many memory requests resulting from cache misses can be outstanding at the same time, hiding the miss latency by amortizing it over many misses.

4.4 Architecture 4: Memory-side cache with direct cluster access

Architecture 4 is a hybrid of architectures 2 and 3; it has a memory-side cache, as in architecture 2, yet allows the clusters to issue loads and stores directly to memory through the cache, as in architecture 3.

5 Hardware Costs

5.1 Memory-Side Cache

The baseline architecture already contains a complete crossbar between the memory interface and the banked DRAMS; if the memory-side cache is placed behind that crossbar it essentially gets it for free. Given this, the cost of the memory-side cache is the data storage, the tag storage (at 1 tag per data word, since each line is just 1 word), and the necessary cache logic and internal buffering. In the layout, the banks of the cache can line up with the memory and SRF banks.

5.2 Cluster-Side Cache

The cluster-side cache is basically the same cache model as the memory-side cache, with 4 additional costs, which are as follows.

- 2 complete 8x8 crossbars, one at each side of the cache, to switch the requests and responses across the cache banks, and 16 2-to-1 multiplexers to switch from the 16 clusters issuing 16 requests every 2 cycles to the 8-banked cache processing up to 8 requests each cycle.

- A large reorder buffer to store the data to be sent back to the clusters. This is implemented as 16 register files, one per cluster. Each register file contains 2 64-bit wide ports, for reading and writing a word each cycle, and a third 1-bit wide port for reading the valid bit from each ROB entry to use in the stall logic. There are common 'head' and 'tail' indices across the 16 register files. It is known that the address issued to the 1-bit wide port will always be 1 less than the address issued to read from the 'head' index; this information could possibly be used to implelement the third 1-bit port more cheaply.

- A requests FIFO to feed the cache with the cluster requests. This is implemented as 16 2-ported register files.

- 8 new buses between the cluster cache's 8 banks and the memories 8 banks.

5.3 Cluster Memory Access Without the Cluster Cache

The cost of this is just the same reorder buffer needed for the cluster cache and an address generator of width 16, the 8 buses, and some stall logic.

6 Coding the Applications for Each Architecture

6.1 Application 1

To implement application 1 we chose the maximum strip size which wouldn't overflow the SRF and software pipelined the strips to overlap memory accesses with kernel computation. The implementations for architectures

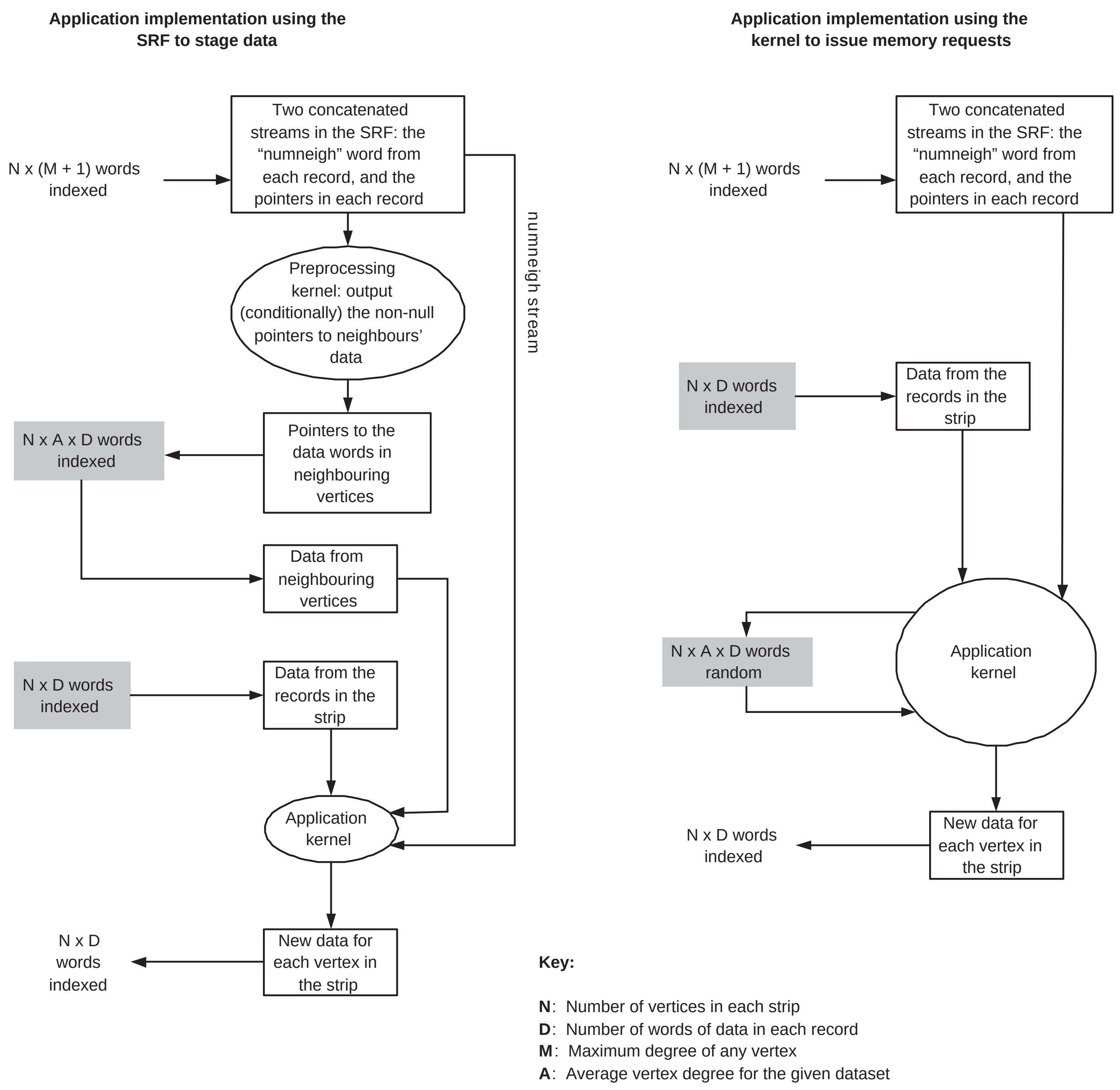

Application implementation using the SRF to stage data
Application implementation using the kernel to issue memory requests
N x (M + 1) words indexed
Two concatenated streams in the SRF: the "numneigh" word from each record, and the pointers in each record
numneigh stream
Preprocessing kernel: output (conditionally) the non-null pointers to neighbours' data
Pointers to the data words in neighbouring vertices
N x A x D words indexed
Data from neighbouring vertices
N x D words indexed
Data from the records in the strip
Application kernel
New data for each vertex in the strip
N x D words indexed
N x (M + 1) words indexed
Two concatenated streams in the SRF: the "numneigh" word from each record, and the pointers in each record
N x D words indexed
Data from the records in the strip
N x A x D words random
Application kernel
New data for each vertex in the strip
N x D words indexed
Key:
N: Number of vertices in each strip
D: Number of words of data in each record
M: Maximum degree of any vertex
A: Average vertex degree for the given dataset

Figure 3: Implementing a Strip of an Irregular Stream Application

1 and 2 (no cache and memory system cache, respectively) are the same, but the implementation on architectures 3 and 4 (clusters issue loads and stores) is quite different, as is illustrated in figure 3. In this diagram, the left side of each flowchart specifies the number of words loaded from and stored to memory, and the right side

illustrates the data stored in the SRF (rectangles) and kernels operating on that data (circles). The memory accesses in the grey box may exhibit temporal locality, but the non-coloured memory references do not; the use of a cache will not improve their performance any. There are no dependencies between strips in application 1, so the complete application simply becomes repeating the process in this diagram for all strips of input vertices in each node.

Note that the amount of preprocessing work required on architectures 3 and 4 is much less than on the memory system cache architecture, as the repeated indexed loads to traverse the input graph structure and read the data words from neighbouring vertices are not needed. One secondary benefit deriving from this is that SRF space is not wasted by storing either replicated data or excessive null pointers; each of these inefficiences is present in the implementation on the architectures without a cluster cache.

6.2 Application 2

Application 2 contains update-dependencies between vertices, and the vertices need to be processed in a certain order. In the most inefficent implementation, in which the vertices are stored in a random order in memory, the entire graph could be traversed over and over again, each time looking for vertices whose predecessors have become ready. To be able to traverse the DAG in a single pass it needs to be topologically sorted, ensuring that all predecessors of a vertex are processed before the vertex is. This can be done either by storing the data in a top-sorted manner in memory or by 'sorting by reference', in which the index streams basically contain a level of indirection through a table. In this project we've assumed, for simplicity, that the data is stored top-sorted in memory; this allows us to avoid simulating a streaming version of top-sort, which is beyond the scope of this project.

Consider first the single node situation without a cache, in which a strip of contiguous vertices is loaded into the SRF and processed by the clusters. For each vertex (record) processed, it is known that all of its predecessors are located before it in the ordering of vertices, but it is not known how far before. Specifically, the predecessors of a vertex could be located before it in the current strip in the SRF, and worse, could be processed by another cluster in parallel with the vertex. Handling this problem in a SIMD architecture is difficult and amounts to serialising the processing of the strip; thus, to support application 2 in an Imagine-like architecture, it is necessary to ensure that all predecessors of a vertex are located before it in the total ordering of the vertices, and also that all vertices in a strip have all their predecessors in earlier strips (i.e.) that there are no intra-strip neighbours. In the worst case, in which the DAG is a linked list, each strip will be of length 1, but that can't be helped, as there is no parallelism to exploit.

Extend the situation to now include multiple nodes without caches. The same requirement as in the single node case holds, in that all vertices in a single strip must depend only on vertices in previous strips, but an additional danger is that vertices could depend on vertices in other nodes. To guarantee that at the time a vertex is processed in one node all of its predecessors in other nodes are ready the nodes must synchronize with each other using barriers, conceptually traversing the DAG one 'level' at a time as illustrated in figure 4 for an example DAG and 2 nodes. All nodes process the vertices within the current level in parallel, using multiple strips if the number of vertices is too large for one strip, and then they barrier synchronize before beginning the next level. For optimal load balancing between nodes, each node should be assigned an equal size section of each level.

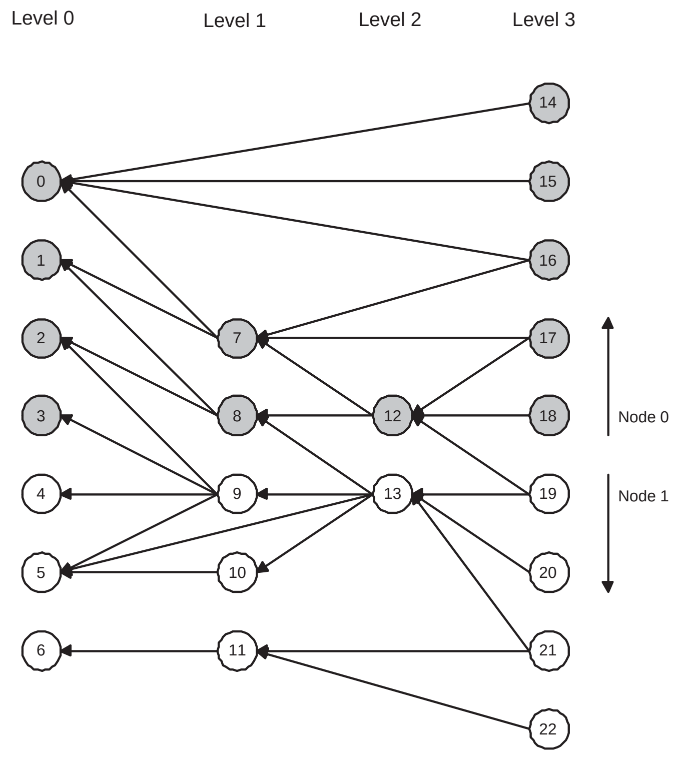


Figure 4: Application 2 Top-Sorted DAG

Lastly, consider adding a cache to the multi-node case. The order of traversing the DAG which is safe for the multi-node no-cache case is still safe, provided that at each barrier the elements just computed are flushed out to memory. For this application a write-through cache would be most useful, and to enable the write-back cache we are using to act somewhat like a write-through cache we simulated a 'flush-no-invalidate' operation in addition to the standard 'flush' operation, in which all dirty entries are copied to memory but not marked as invalid in the cache.

Once this ordering of strips and barriers is defined, implementing each strip essentially becomes the same problem as in application 1, for which the process of loading and operating on each strip is illustrated in figure 3.

7 The Simulator

We implemented a 'cycle-by-cycle performace simulator'; it is aware of the latencies, throughputs, and resource bottlenecks in the architecture, and simulates the performance of that architecture on a given application (which is coded in a 'macrocode' the simulator understands). It is not a functional simulator, however, as it doesn't actually simulate the computations, just the time they take and the resources they need.

The underlying mechanism in the simiulator is the use of infinite-length priority queues sorted by timestep value; each block, such as the SRF, the memory, etc., contains various queues which it receives messages on and services in a specific manner to simulate throughput and resource constraints. Requests from one block to another for memory reads and writes are simulated as messages pushed on the queue for the destination block. The latency part of the model comes from the timestep value contained within each message; a message may not be serviced until at least the timestep it contains.

The stream controller part of the simulator is able to track dependencies between instructions and issue them out of order, and has an issue window of instructions it can concurrently execute, provided that the resources they need to run are available.

By instantiating classes repeatedly we can perform multi-node simulations; each node contains a network interface class which handles the sending and recieving of messages (requests) over the network. The network model we implemented is completely flat, and models network congestion by allowing the queue of requests received over the network in each node to grow without bound if the rate of received messages is greater than the rate of serviced messages.

The kernel program and clusters are simulated by parameterizing the number of cycles of pipelined computation the kernel performs, and, in the case of the cluster cache architecture, allowing the kernel program to make requests for the needed addresses (which are real addresses, not fake parameters) to the cluster cache which will forward them to the memory system on a miss. The requests made by each cluster are determined from the datasets we generate, as they would be in the real hardware.

The simulator has an element of randomness, which tends to produce a small amount of noise in the results obtained. This can be observed as small fluctuations in the result graphs presented in this report.

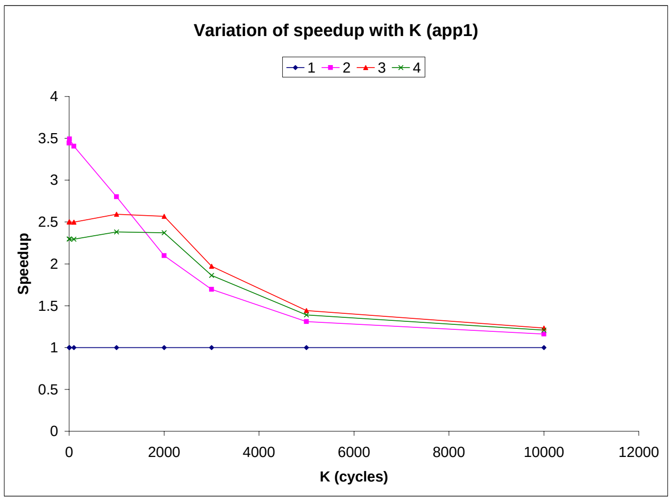
The cache models are the most detailed part of the simulator; they attempt to accurately model real caches, and are configurable with respect to the policies they support.

8 Results

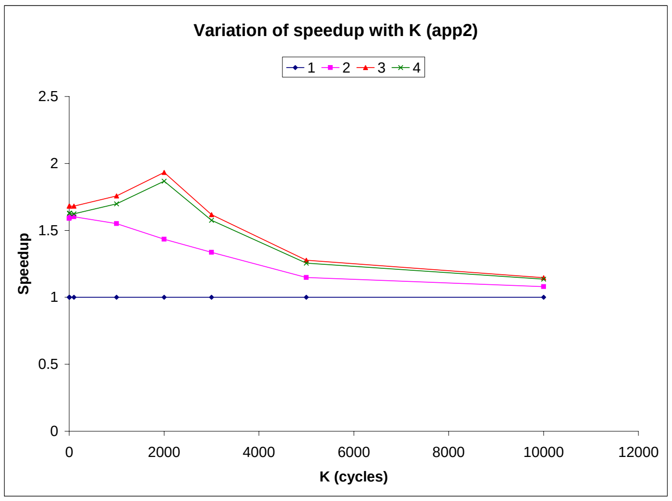
Figures 5 and 6 contain a series of 12 graphs, the same 6 for each of the 2 applications studied. For each graph, the following baseline parameters were used, and a single parameter was varied.

- SRF size = 256 kwords and cache size = 32 kwords (data).
- Input datasets: average degree (A) = 32, number of words of data per record (D) = 8, ROB length = 128 rows, number of nodes = 1, and number of vertices = 16,384.
- Peak cache bandwidth = 8 w/cycle, peak memory bandwidth = 1 w/cycle, and peak network bandwidth = 1 w/cycle.

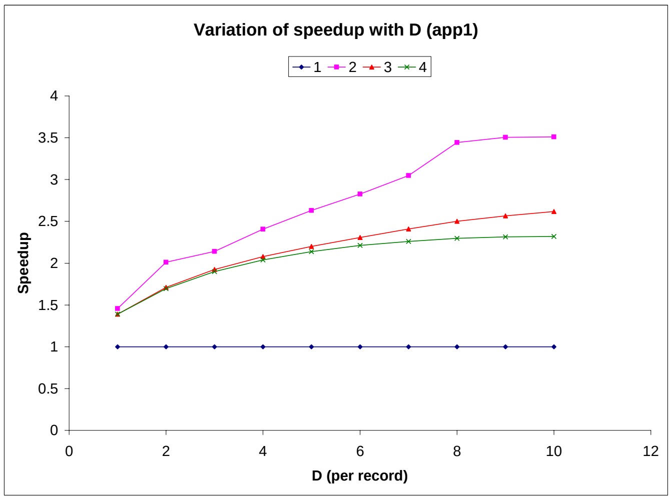
The one exception, which used 2,048 vertices per node as a baseline instead of 16,384, was the increasing number of nodes test, due to the fact that with both a



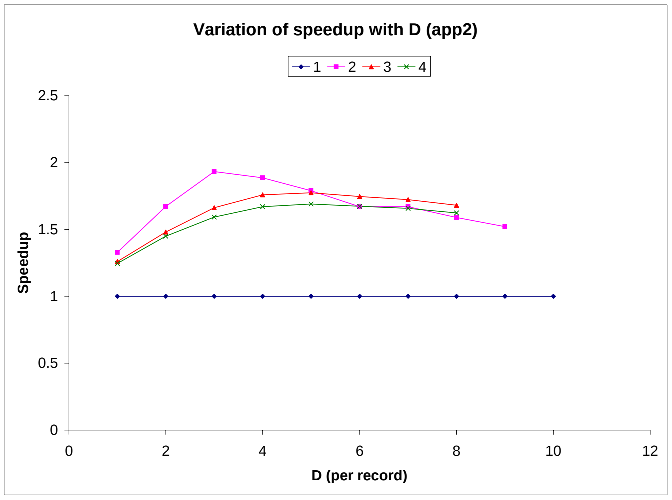
(a1)



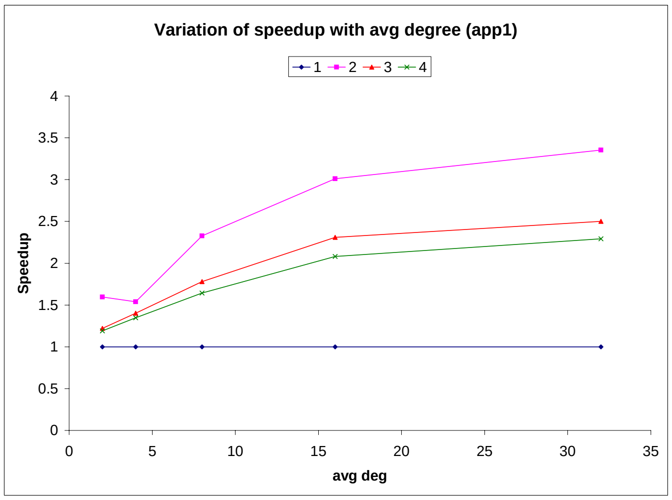
(a2)



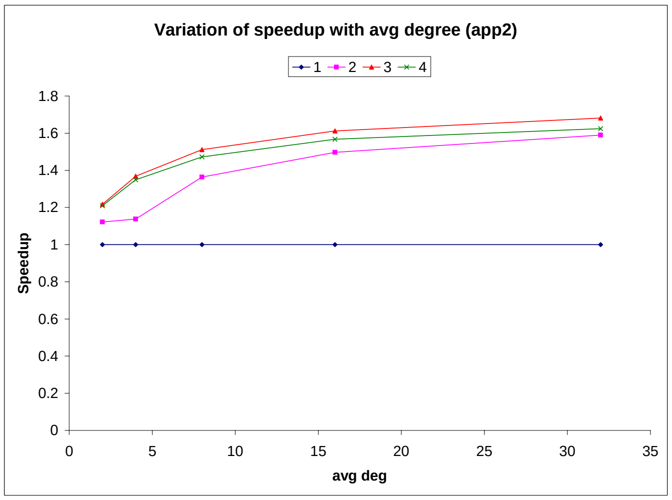
(b1)



(b2)

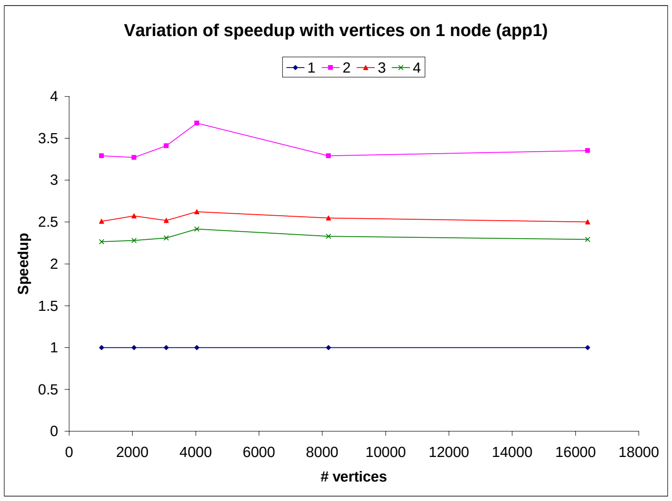


(c1)

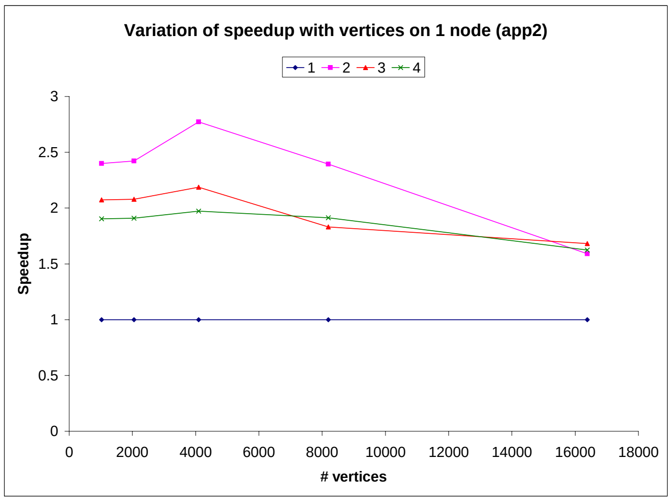


(c2)

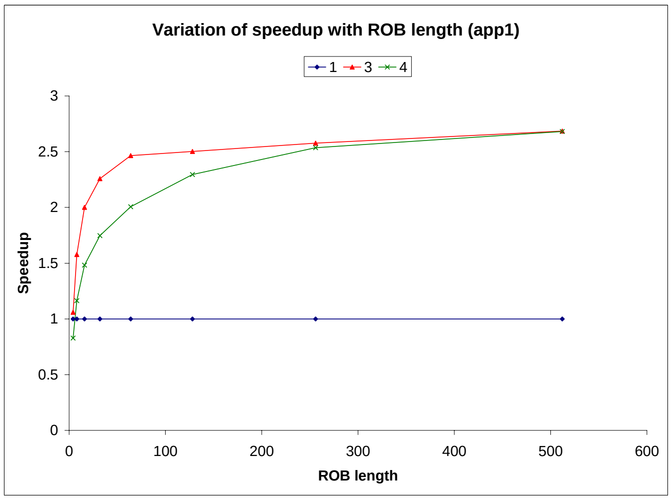
Figure 5: Results.



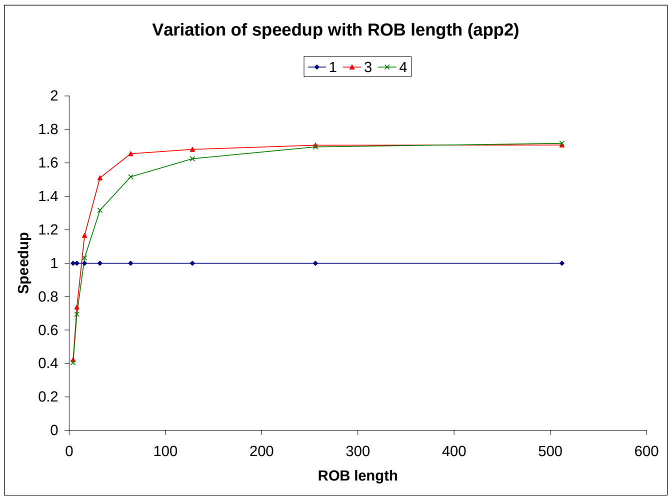
(a1)



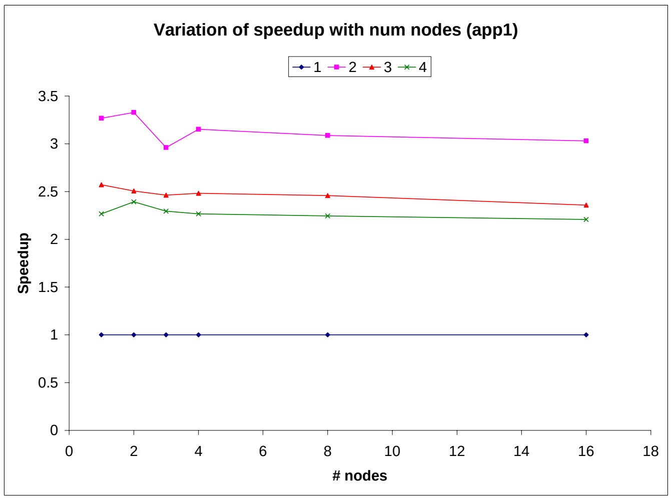
(a2)



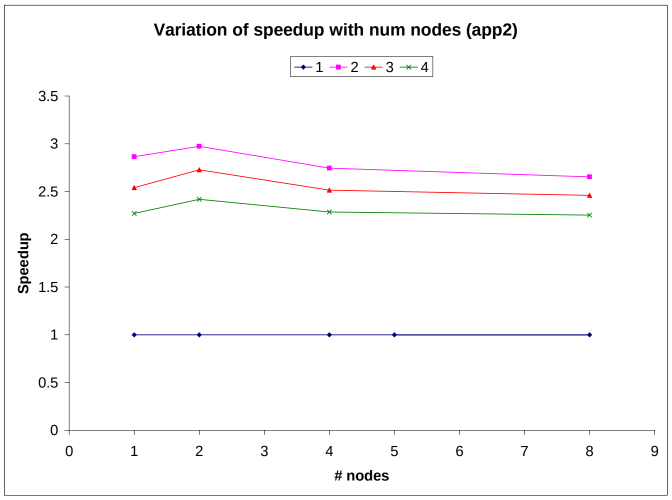
(b1)



(b2)



(c1)



(c2)

Figure 6: Results (continued).

large dataset and a large number of nodes, our simulator needed more memory than the available servers (*dim-sum* and *tree*) could provide, and would have taken many hours to complete a single simulation.

9 Analysis of Results

Figures 5(a1) and 5(a2) are graphs of the speedup vs. K, the number of cycles of pipelined arithemetic per record, for each application. Several points can be made about these graphs:

- As K gets larger, the speedup of all 3 cache architectures approaches 1, due to the application transitioning from being memory-bandwidth-limited to computation-limited. (i.e.) A cache is only useful when the number of ops per record performed by a kernel is 'small'. In architecture 2, in which indexed loads to the SRF are used for all memory reads, the speedup decreases monotonically.
- The speedup curves for the cluster-load architectures (3 and 4) initially rise as K increases. This is due to the 'spacing out' of requests from the clusters; when K is small, due to kernel software pipelining the clusters will be issuing requests at or close to the peak bandwidth of the cache, and the spacing out of requests helps performance by:
  - Reducing the effective rate at which the clusters issue requests to a sustained rate that the cache can handle; due to bank conflicts, the cache can usually not sustain its full peak bandwidth. When the FIFO containing requests from the clusters fills, the clusters are back-pressured by stalling them.
  - Improving the 'second miss' factor, as, by spacing out requests for the same address, there is a greater chance that later requests will be a hit.
- The relative performance of arch. 2 against the cluster load architectures was greater for app. 1 than app.2. This is due to the barrier synchronization required every small number of strips in app. 2 breaking software pipelining. Apps on arch. 2 software pipeline at the strip level, while apps on arch. 3 and 4 software pipeline at the record level, a much finer granularity, decreasing the pipeline prime and drain overhead they suffer relative to arch. 2.

Figures 6(b1) and 6(b2) are graphs of the speedup vs. the length of the cluster ROB, as a number of 16-wide rows. Note that:

- As the ROB length increases, the performance of architectures 3 and 4 for both apps become the same, indicating that the (more expensive) dedicated cluster cache architecture is only better than the (cheaper) shared memory cache architecture when the ROB is 'small'.

Other observations which can be made from the graphs include the following.

- Increasing the average degree of the graph increases speedup (figures 5(c1) and 5(c2)), as both the amount of temporal locality and also the fraction of the total number of memory references in the appliations which are susceptible to caching increase.
- Increasing the number of words of data per record increases speedup (figures 5(b1) and 5(b2)), once again by increasing the fraction of the total number of memory requests which exhibit temporal locality.
- Relatively constant speedup was seen with increasing number of vertices per node, even well after the total cacheable part of the dataset exceeded the cache size at 2000 vertices. This implies that the datasets used had a high degree of locality in their connections.

10 Conclusions

There are 2 conclusions which can be drawn fom the work prosented in this report.

- Using a stream cache to capture temporal locality can be effective if the dataset contains enough locality; speedups of over 3.5 were observed in our experiments.
- The 2 styles of architecture evaluated (SRF-load and cluster-load) showed relatively similar performance in the applications studied, and each have their different pros and cons. One factor which was neglected, however, was that running multiple application kernels on a single strip may have helped the performance of the SRF-load architecture more than the cluster-load architecture.

11 Discussion - Limitations on Speedup

Several limitations on the speedup attainable using a cache were observed in this project.

- Amdahl's Law: only some of the memory loads exhibit temporal locality, and the preprocessing overhead isn't helped by the cache.
- Bank conflicts can reduce the effective cache bandwidth.
- In the case of application 2, the speedup was hurt a lot by the synchronization barrier breaking the strip-level software pipelining.

12 Future Work

- Study cache performance on real apps in ISIM.
- Investigate using add-and-store with a cache.
- If an application can be found which would benefit from some level of partial coherence, such as memory locking, explore how much this would help and how expensive different partial coherence mechanisms would be.