Data Cache Performance When Vector-Like Accesses Bypass the Cache

Michael Nahas **W illiam Wulf**
University of Virginia
Charlottesville, VA 22903
nahas@virginia.edu

A Stream Memory Controller, when added to a conventional memory hierarchy, routes vector-like accesses around the data cache. A memory system was simulated under these conditions and the data cache performance increased dramatically. The gain in performance was a result of the increased temporal locality of the access pattern. The access pattern also showed a decrease in spatial locality, making smaller cache lines nearly as effective as long ones.

1.0 Introduction

This research attempts to quantify the effect on the data cache when vector-like accesses are handled outside the cache system. We hypothesized that vector-like accesses took up large amounts of space in the cache, had poor temporal locality, and, therefore, would cause cache pollution. In the primary experiments, we removed the vector-like accesses from the cache system and saw cache performance increase. Additional experiments, which examined the causes of the increase in cache performance, provided evidence to support our hypothesis

We wanted to examine a memory system where vector-like accesses bypass the cache because our research group has designed a piece of hardware called the Stream Memory Controller (SMC) [7,9] which can increase the speed of vector-like memory accesses which go directly to DRAM. Since previous research [8] had studied the effect of the SMC on the vector-like data, we needed to simulate the memory system where the

vector-like accesses bypass the data cache in order to understand the overall system performance.

To measure memory system performance, we simulated the data cache using memory address traces which were acquired from Brigham Young University's Performance Evaluation Laboratory [2,3]. Using two different methods, the accesses handled by the SMC were removed from the traces, resulting in new traces that represented the accesses to the data cache in a system with an SMC. Cache performance of such a memory system was measured by running these traces through a simulated memory system modeled after that of the DEC Alpha 21264.

After measuring the increase in cache performance, we analyzed the traces with a variety of methods to determine the causes of the change in performance. The SMC did remove a large percentage of the addresses with low temporal locality from the trace, increasing the overall temporal locality and producing a higher hit rate in the cache. The SMC also removed accesses with a high spatial locality, decreasing the overall spatial locality of the access pattern. This means that a smaller line size would not cause as large a drop in performance on a system with an SMC as on system without an SMC.

Section 2 of this paper describes the SMC and the architecture of a system incorporating an SMC. Section 3 describes the methodology and tools used in this investigation. Section 4 reviews how streams for the SMC were identified by recognizing patterns in the traces. Section 5 discusses an alternate method to identify streams. Section 6 contains the results of cache simulation and explains the causes of the increase in cache performance. Section 7 is a conclusion of all the results.

2.0 The Stream Memory Controller

The SMC is a hardware device that is placed in parallel with the first level data cache in a memory system. The SMC increases the performance of vector-like loads and stores by dynamically reordering when the accesses are issued to the memory. Because this special purpose hardware is in parallel with the first level data cache, the vector-like accesses never enter the data cache.

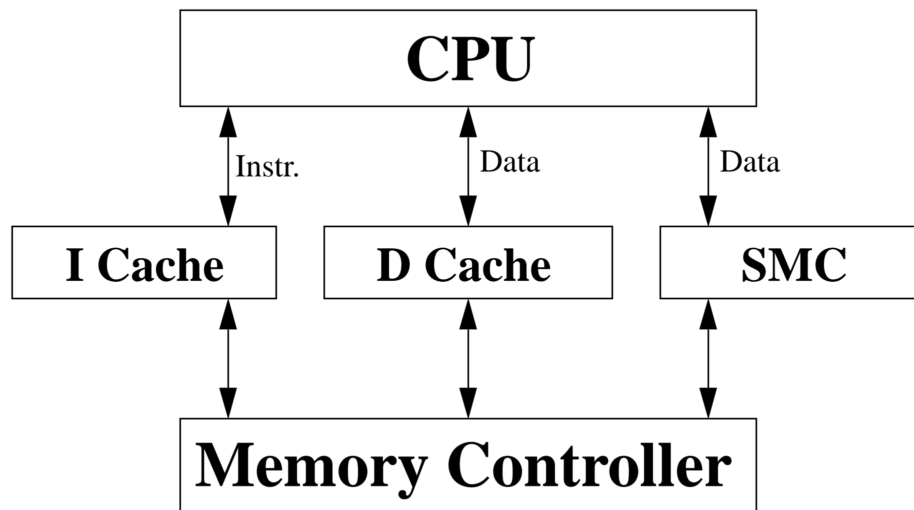

FIGURE 1. Layout of an SMC Memory System

Streams, the vector-like accesses handled by the SMC, are entire arrays that are accessed sequentially by a conventional scalar processor, e.g. a[0], a[1], a[2]... The CPU in an SMC system is a scalar processor, so it still executes an instruction for every load, store, and calculation. The loads and stores to elements in a stream are handled by the SMC, which reorders them and performs them at a higher bandwidth than a cache could.

SMC systems are often compared to systems with a vector processor. Vector processors work by loading fixed length sections of arrays into vector registers and performing operations on the vector registers. Each vector instruction performs multiple operations on the elements in a vector register. Each of these operations must be indepen-

dent of the results of every other operation; this allows the operations to be performed at a very high rate.

Because the processor in an SMC system is not a vector processor, operations on elements in a stream can be dependent on the results of previous items in the stream. This makes stream operations a superset of vector operations. Also, the processor in an SMC system can perform different operations on each element in a stream. This allows the SMC aid in operations such as sorting, where as a vector processor could not.

From the programmer's perspective, the SMC is a number of FIFO queues, each holding a stream. A queue is initialized by sending to the SMC the properties defining the sequential array access: the address at which to start (the base address), the distance between each item (the stride), whether the item is a byte, word, or double word (the data size), and, optionally, the number of items to be accessed (the length). Once the queue is initialized, a program can simply pop the next element of the stream from a read queue or push it into a write queue.

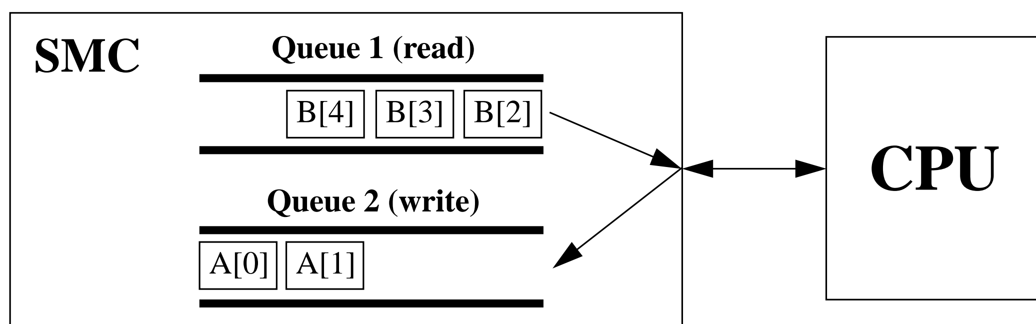

FIGURE 2. Programmer's Representation of the SMC as FIFO Queues

The SMC is able to handle sequential array accesses with more bandwidth than the cache system by dynamically reordering the loads and stores to the arrays. When the CPU is using a read FIFO, the SMC can read ahead in the array in a large block, using the memory system more efficiently. The SMC holds the data in internal memory until the CPU

pops the data off the queue. For write FIFOs, the SMC can use its internal memory to queue up the written values and write them out to main memory together. A small state machine in the SMC decides when it is most efficient to read ahead on a read queue or flush the buffer of a write queue.

The SMC gets higher bandwidth for the large block transfers by exploiting the internal structure of a DRAM, which favors some accesses over others. By grouping together reads and writes that go to the same DRAM page, the SMC lessens the number of reads or writes that cause the DRAM to switch pages. Reads and writes which do not cause page switches are 3 to 10 times faster than ones that cause page switches.

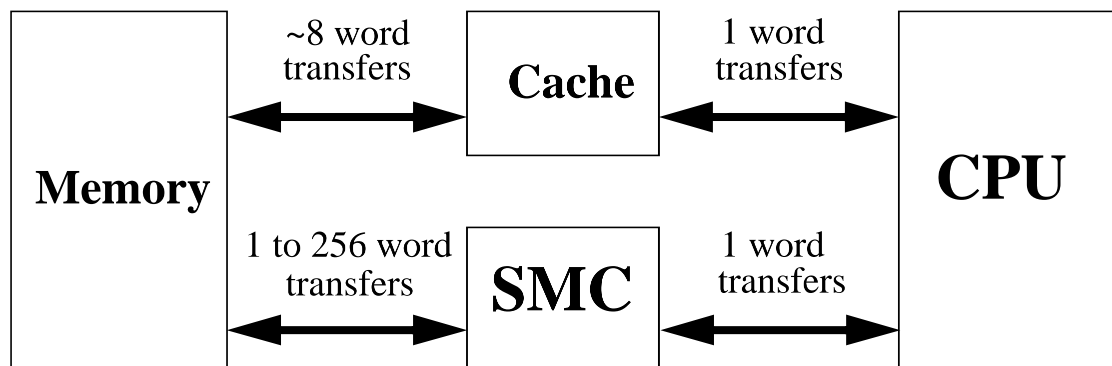

FIGURE 3. Relative Sizes of Memory Transfers

The SMC can queue writes and read ahead whenever it is efficient to do so, grouping together many single word transactions which go to the same DRAM page. These group transfers drastically reduce the number of page switches in the DRAM, and utilize nearly 100% of the available bandwidth from the DRAM. These results are documented in [7,8,9].

3.0 Tools and Methodology

We received a number of traces from BYU's Performance Evaluation Lab. The one we used for these experiments was SPEC's 061.Kenbus1 multi-user benchmark which had been run on a SPARC processor. The Kenbus benchmark is a collection of 80 UNIX command shells representing users. Commands are typed into the shells at 3 characters a second. The commands are common UNIX utilities including cc, grep, cp, ed, sed, awk, and a number of file manipulation tools such as cp, rm, and chmod. The Kenbus benchmark was chosen because it represents a common load on a multi-user system. The applications in the benchmark are not scientific computations, and they are not vector intensive.

BYU supplies its traces broken into blocks of 500,000 accesses. From the Kenbus trace, the first 10 blocks and 10 consecutive blocks from the middle of the trace were chosen to be processed. By examining the strings processed by the code, it was determined that the first 10 blocks of the Kenbus trace include a trace of the C preprocessor, and that the other 10 blocks tested include a trace of the linker.

Since we did not generate these traces ourselves, we needed to identify which accesses in the traces would be handled by the SMC and not run through the cache system. Identification of stream accesses would normally be done by the compiler [1], which must insert instructions to initialize the FIFOs into the object code.

We used two methods to identify stream accesses. The pattern recognition method identifies loops and stream accesses within the loops. These stream accesses are then removed from the trace, because the accesses would be handled by the SMC and not pass through the data cache. The second method of removing stream accesses matches data reads and writes to the load and store instructions that caused them. Then, individual load and store instructions are identified as sequentially accessing an array, and the data reads

or writes caused by those instructions are eliminated. This second method is called the disassembly method because it is necessary to disassemble instructions to identify which are loads and stores. The disassembly method also produces a second trace without stream or quasi-stream accesses. Quasi-streams are a type of access patterns that might be converted to streams by compiler optimizations. These methods of making traces without stream accesses will be discussed in more detail in sections 4 and 5.

Because neither of these methods used the source code, neither of these methods could be guaranteed to accurately identify only those streams that would be known at compile time. Both methods of stream identification were susceptible to false positives, incorrectly identifying accesses as parts of a stream, and to false negatives, not identifying accesses that are parts of a stream. Having two methods of removing streams allowed us to compare the results of both methods. At the end of sections 4 and 5 are summaries of the limitations of each method.

Once the stream and instruction accesses have been removed, the resultant traces are composed of all the accesses going to the data cache in an SMC system. A fourth trace, representing the accesses to the data cache of a conventional system, was also created. All of these traces were then run through a variety of cache simulators, including the one for the Alpha 21264 memory hierarchy. This produced four different categories of results: streams not removed, streams removed by the pattern recognition method, streams removed by the disassembly method, and streams and quasi-streams removed by the disassembly method (labeled as disassembly*.) Figure 4 illustrates this methodology. The results of all simulations are presented in section 6.

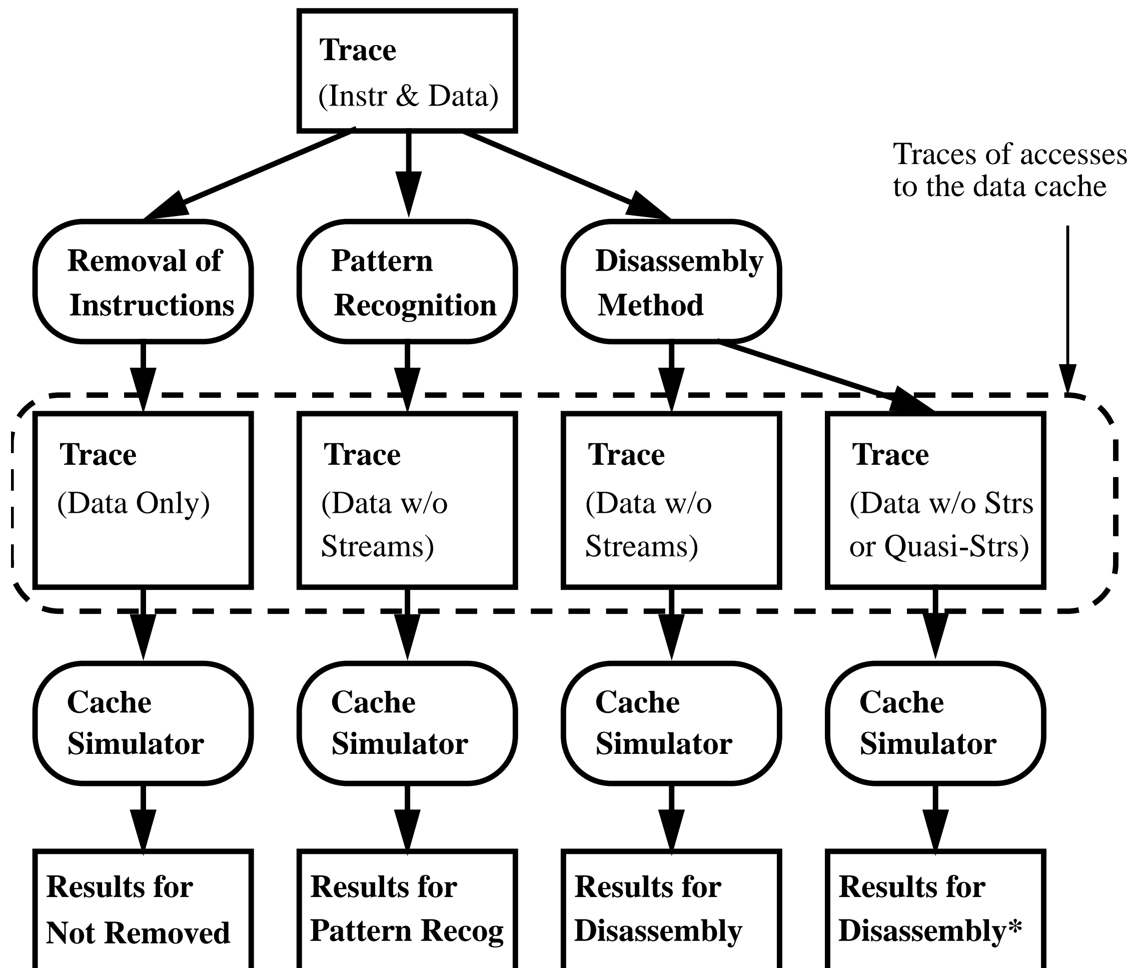

FIGURE 4. Flow of Original Trace into Results

4.0 Pattern Recognition

The pattern recognition method is one method of identifying streams in a trace. It works by finding repeated patterns of accesses that might represent the execution of a loop. The stream accesses within the loop are then identified and removed. The algorithm is limited by the fact that it only detects loops that have the same number of accesses in every iteration. This prevents loops containing intermittently taken branches from being identified.

The loop recognizing algorithm looks for the beginning of a loop within a sliding window. The beginning of a loop is found by searching for an access that is repeated 3 times with an equal number of accesses between each access. Once the beginning of the loop is recognized, a potential template for the loop is constructed. The template identifies which accesses in the loop are constants, which are streams, and which are neither. This last type are called wildcards. The template is then considered; if it is too small, has too few repetitions, or has too many wildcards, the loop can be rejected. If it is accepted, the stream accesses are removed from the loop.

```
// Pseudo Code for Pattern Recognition Algorithm

TraceFile file_in("filename");
window_end = 0;  // the lower end of the window

// i is the upper end of the window.
for (i = 0; ; i++) {
     // search window - j is the window length
     for (j = 0; j > i-window_end && j < WIN_MAX; j++)
          if ( file_in[i] == file_in[i-j] &&
               file_in[i] == file_in[i+j] ){
               template = create_template(i-j, j);
               if ( accept(j, template) ) {
                    // replace and advance window.
                    i = replace(i-j, j, template);
                    window_end = i+1;
               }
          }
}
```

The TraceFile class allows the accesses in a file to be indexed like an array and compared using ==. The sliding window starts at "i+WIN_MAX" and goes down to "window_end" or "i-WIN_MAX".

FIGURE 5. Pseudo Code for the Pattern Recognition Algorithm

A slight modification of the algorithm allowed the final version to find nested loops. When a loop is found, the iterations of the loop are removed from the trace and replaced with a marker. The pattern recognition algorithm could then make more passes

and identify nested loops, i.e. loops containing a marker. When every identifiable loop had been found, the markers were replaced and all streams eliminated.

The pattern recognition algorithm identifies possible loops, confirms that they are loops, and then removes streams from the loop. It is a simple algorithm that only identifies loops containing a fixed number of accesses and is susceptible to a few problems that are outlined in section 4.1. Despite its simplicity, it performs well and is able to find nested loops.

4.1 Limitations

Because the pattern recognition method uses only run-time information, it incorrectly identifies some accesses as compiler-recognizable streams. These accesses would not be identified by the compiler as streams, and therefore the SMC would not have been used to fetch them. These accesses are false positives; accesses that are identified as being handled by the SMC when they would not be in an actual implementation.

A linked list if a good example of a source of a run-time stream that is not compiler-recognizable. If all the nodes of a linked list were allocated at one time, it is likely they would be allocated sequentially in memory. When the list is traversed from the first node to the last, the accesses to the nodes would be sequential and evenly spaced. This would be identified by the pattern recognition program as a stream, producing false positives.

The pattern recognition method is also susceptible to false negatives, i.e. failing to identify streams that would have been recognized by the compiler. The current method of pattern recognition can only identify loops containing a constant number of accesses in each iteration. Therefore, it will not identify any loop containing an intermittently taken

branch. Streams in these loops would produce false negatives. However, streams in a loop that consistently took one direction of a branch, if even for a short time, can still be identified since the sequence of memory accesses looks like that of a fixed-length loop.

Another source of false negatives is misalignment of the template by the pattern recognition program. Occasionally, the template selected by the algorithm is not the ideal template; the template contains the second half of the loop body followed by the first half. This offset copy matches the second half of one iteration and the first half of the next. This works well, except at the first and last iteration of the loop. If part of either of those iterations matches enough of the template, it is considered another iteration of the loop, and data that is not actually part of the loop causes the template to be modified. This may change a stream in the template to a wildcard. The accesses that should have been part of a stream and marked for the SMC are left in the final trace.

The pattern recognition method has one more limitation; if separate parts of the same stream are identified in two different loops, the stream would appear to have two initializations, rather than just one. In this case, the pattern recognition algorithm would correctly identify most of the accesses as parts of a stream, but would overstate the number of FIFO initializations required by the SMC to service this stream.

Based on limited examinations, we believe that more accesses are identified as false positives than are overlooked as false negatives. This suggests the pattern recognition method provides an upper bound on the number of accesses that are parts of streams. Unfortunately, it cannot serve as an upper bound on cache performance since the number of false negatives is significant and cache performance is determined by *which* accesses are excluded, not the *number* that are excluded.

5.0 Disassembly

The disassembly method is an alternative way to identify and remove stream accesses in a trace. Each data read or write access in the trace is matched with the load or store instruction that generated it. If the list of reads or writes for an individual instruction appears to match the pattern of a stream, the reads or writes caused by it are removed from the trace. This identifies and removes streams in which the stream elements are accessed by a single instruction in a loop body, the most common case.

The load or store instruction for a given read or write is identified by searching backwards from the data access to find an appropriate memory instruction in an instruction access. A list of associated reads or writes is built up for each load or store instruction. Load and store instructions are uniquely identified by their address, which is part of the instruction load access. When all the reads and writes have been placed into the list for the appropriate address, each list is processed to determine if it is able to be handled by the SMC. If the list of loads or stores is judged to be able to be handled by the SMC, the loads or stores are eliminated from the trace. This method is illustrated in Figure 6.

A list is considered able to be handled by the SMC if it contains at least 10 reads or writes and over 90% of the list is composed of sequences of more than 3 accesses that fit the pattern of a stream. The 90% is to prevent ignoring some loads or stores that might access streams of length less than 3. Otherwise, a load in strcmp() would not be identified if it ever works on a string less than 3 characters long.

While studying the access patterns of individual instructions that were identified by the disassembly method, we found another type of pattern besides streams. Some loads and stores accessed very long sequences of increasing addresses, but the stride between the addresses was not uniform, so they could not be considered streams. Other authors [6]

Disassembly Method to Removing Stream Accesses

Original Kenbus Trace

Instr. ADD 0x1F00	Instr. LD Word 0x1F04	Instr. SUB 0x1F08	Instr. Jump LE 0x1F0C	RD Word 0x0001 0x0A0C	Instr. ADD 0x2B00

Addresses loaded by instr at 0x1F04

0x0A00
0x0A04
0x0A08
0x0A0C

1. The read of address 0x0A0C was generated by the load instruction at address 0x1F04.
2. The address 0x0A0C is added to the end of the sequence of addresses read by the load instruction at address 0x1F04.
3. At the end of the first pass through the trace, the sequence of addresses loaded by the instruction at address 0x1F04 is judged to be able to be handled by the SMC.

Kenbus Trace without Stream Accesses

Instr. ADD 0x1F00	Instr. LD Word 0x1F04	Instr. SUB 0x1F08	Instr. Jump LE 0x1F0C	~~RD Word 0x0001 0x0A0C~~	Instr. ADD 0x2B00

4. The second pass deletes all reads and writes that were caused by instructions that would be handled by the SMC.

FIGURE 6. Overview of Disassembly Method of Removing Stream Accesses

have called these “quasi-streams”. Quasi-streams might be made into streams by compiler optimizations. An example of this is quicksort, which contains store instructions which step sequentially through an array but skips over some elements. Quicksort can be transformed so that all its loads and stores are able to be handled by the SMC. Because quasi-stream accesses might be convertible into streams, the program which implemented the disassembly method was modified to produced another trace without streams or quasi-

streams. To differentiate it from the disassembly method when only streams are removed, this method is referred to as "Disassembly*".

5.1 Limitations

In the disassembly method, all loads and stores are matched up with an instruction, which in an SMC system could be changed to a FIFO pop or push. This makes the method resilient to false positives. A load or store would have to be 90% streams to generate a false positive, and that seems very unlikely.

The disassembly method is susceptible to false negatives, however. If a stream is loaded by two different instructions, the disassembly method may not identify either instruction as the source of a stream.

Since the disassembly method is unlikely to return a false positive, it makes a good conservative estimate of the amount of streams present in a trace. It also suggests that the disassembly method is a good lower bound on cache performance.

6.0 Results

This section is divided into 3 subsections. The first subsection contains all the results of stream identification. The second subsection contains a description of the simulated memory system and the results of the cache performance in the system. The third subsection explains the causes for the increased cache performance.

6.1 Results of Stream Identification

Table 1 presents the number of streams and the number of SMC accesses for the different methods of stream identification.

The disassembly method, which does not remove quasi-streams, represents a good lower bound and had 8.4% of its data routed through the SMC. The disassembly* method,

TABLE 1. Number of Streams Found and Number of Accesses to the SMC

Trace	Accesses to SMC Removed by	# of Cache Accesses	# of Streams	# of SMC accesses
Kenbus (1)	NOT Removed	715062	0	0
	Pattern Recog	516712	17639	198350
	Disassembly	664961	1124	50101
	Disassembly*	589864	3199	125198
Kenbus (2)	NOT Removed	661596	0	0
	Pattern Recog	437454	22294	224142
	Disassembly	596659	1582	64937
	Disassembly*	542952	2711	118644
Total	NOT Removed	1376658	0	0
	Pattern Recog	954166	39933	422492
	Disassembly	1261620	2706	115038
	Disassembly*	1132816	5910	243842

which does remove quasi-streams, had 17.7% of its data going through the SMC. The pattern recognition method, the upper bound on stream accesses, had 30.7% of its data going to the SMC, more than 3 times as much as the lower bound.

These numbers are very encouraging. The C preprocessor and the linker are not array intensive applications, but they do contain a large amount of string manipulation. For comparison, when the pattern recognition method was run on blocks from traces of 3 SPEC Int benchmarks (Compress, Eqntott, Espresso) it identified 24.4% of the accesses as parts of streams, which is close to the amount identified in the Kenbus benchmark. Hennessy and Patterson's survey of scientific benchmarks range from 41% to 1% vector operations, with the median around 16% [5]. However, streams are a superset of vector operations because operations on a stream can be dependent on preceding operations on the stream, unlike vector operations, and streams allow more compiler optimizations such as the one that changes quicksort to use streams.

The pattern recognition method is obviously identifying much shorter streams than the other methods when the number of accesses per stream is compared. The pattern rec-

ognition method averages 10 accesses per stream, while the other methods average over 40 accesses per stream. This is not surprising, since the pattern recognition method is identifying many short dynamically produced streams, and can identify a single stream in pieces, overstating the number of streams.

6.2 Results of Memory System Simulation

The simulated memory system had two levels of data cache. The first level cache is 64kB, with a 32B line size, and 2-way set associativity. The second level cache is 4MB, with a 64B line size and 2-way set associativity. For both caches, random replacement was used.

To make simulation easier, the second level cache was not a unified instruction-data cache. The largest data working set is 273kB and occupies only a small portion of the 4MB L2 cache. This, along with associativity of the cache, leads us to believe that adding the instructions to the second level cache would not cause a significant number of additional L2 misses.

The properties of the caches were selected to mimic the design of the memory hierarchy for the Alpha 21264 as described in Microprocessor Report **[ref]**. Although we are copying many of the properties of the 21264, we are not copying it exactly. The 21264 has many other features, such as data coalescing, that would be difficult for us to model.

To account for artifacts from cache initialization, a.k.a. “cold cache” problems, the cache miss data was not measured for the first half of the traces. About 0.3 million references were allowed to initialize the L1 and L2 caches before measurements began. This point was chosen because the rate of compulsory misses per access leveled out after about 0.2 million references.

TABLE 2. Pattern Recog vs. Disassembly Cache Miss Data for 2nd Half of Trace

Trace	Accesses to SMC Removed by	# of Cache Accesses	# of L1 Misses	# of L2 Misses	# of SMC Accesses
Kenbus (1)	NOT Removed	343730	6671	937	0
	Pattern Recog	271535	2656	521	72195
	Disassembly	335326	5974	848	8404
	Disassembly*	302274	3127	561	41456
Kenbus (2)	NOT Removed	320742	24938	3685	0
	Pattern Recog	213046	14276	3311	107696
	Disassembly	293179	17178	3429	27563
	Disassembly*	271299	15413	3442	49443
Total	NOT Removed	664472	31609	4622	0
	Pattern Recog	484581	16982	3832	179891
	Disassembly	628505	23152	4277	35967
	Disassembly*	573573	18539	4003	90899

The number of L1 cache misses was cut by 27% for the disassembly method. It was cut by 46% for the pattern recognition method and by 41% for the disassembly* method.

The results for L2 cache misses were less impressive. The disassembly method reduced L2 cache misses by 7%, the pattern recognition method reduced them by 17%, and the disassembly* method reduced them by 13%.

Using our lower bound, the disassembly method, 5.4% of memory accesses were sent through the SMC, resulting in a reduction of L1 cache misses by 27% and L2 misses by 7%. Also, the miss rate of the L1 cache dropped from 4.76% to 3.68% and the rate of L2 misses per L1 cache access dropped from 0.70% to 0.68%. Note, not only did the miss rates of the cache dropped, but also the caches were processing fewer accesses, resulting in a larger drop in the total number of misses.

6.3 Sources of Data Cache Performance Increases

This section examines the sources of the 7% to 46% reduction in misses seen in the preceding section. The first subsection considers the *number* of compulsory misses that

were removed when stream accesses were removed. The second subsection analyses the *kinds* of compulsory misses that were removed. The last subsection looks at the changes to the temporal and spatial locality of the trace when streams are removed.

6.3.1 Compulsory Misses

TABLE 3. Word Sized (4B) Compulsory Misses

Trace	Original	Pattern Recog	Disassembly	Disassembly*
Kenbus (1)	49522	14978	37546	19747
Kenbus (2)	68279	28903	47608	35503
Total	117801	43881	85154	55250

The reduction in compulsory misses is dramatic. The disassembly method removed an average of 28% of the compulsory misses. The disassembly* method eliminated an average of 53% of all compulsory misses in the Kenbus traces. The pattern recognition method removed an average of 63% of all compulsory misses.

Compulsory misses are important because even the largest cache must miss on them. The Stream Memory Controller is not a cache and therefore does not suffer compulsory misses. However, the SMC does take a penalty similar to that of a miss every time a FIFO is initialized.

Fewer compulsory misses indicates a more effective cache. If we assume an L1 cache has 1000 lines, each one word long, it must choose 1000 words out of the available 49552 words accessed by the Kenbus(1) trace. It can only hold 2% of the addresses used. The same L1 cache on the trace processed using the disassembly method chooses 1000 out of 37546, so it can hold about 3% of the addresses. Holding a larger percentage of the available addresses increases the chances of a cache hit.

The number of accesses per compulsory miss shown in table 4 is enlightening.

TABLE 4. Accesses per Word Sized Compulsory Miss

Accesses to SMC Removed by	Accesses to the SMC / CMs Removed from Data Cache	Accesses to Data Cache / Data Cache CMs
NOT Removed	n/a	11.69
Pattern Recog	5.72	21.74
Disassembly	3.52	14.81
Disassembly*	3.90	20.50

This result demonstrates that streams are bad for the cache; when that data is in a cache, there are very few accesses for each compulsory miss. The traces without streams are much better for caches because they average more accesses for each item that is cached.

6.3.2 Which Compulsory Misses Have Been Removed?

The previous section showed that a large number of compulsory misses are removed when streams are deleted from the trace. It also showed that the average number of accesses per compulsory miss is low for the accesses in the streams. To see which compulsory misses are removed, the trace was broken down by addresses, and a count was kept of how many accesses went to each address.

Each cell in Table 5 represents a count of the number of addresses, where each of those addresses was accessed the number of times in the first column. Many addresses were accessed only a few times, and a very small number of addresses were accessed many times. Since only 20 blocks of a 2000 block trace were processed, it is not unusual to see addresses only accessed one time.

All columns are the same for more than 4096 accesses. This means that removing streams did not remove a significant number of references from addresses that were accessed more than 4095 times. It is unlikely these addresses would be parts of streams; it is likely they are loop counters or other stack variables that are accessed frequently in a repetitive pattern.

TABLE 5. Addresses Broken Down by # of Accesses to the Address

# of Accesses per Address	Not Removed	Pattern Recog	Disassembly	Disassembly*
1	59480	14764	37118	16998
2-3	41838	13901	26088	16633
4-7	23527	9105	15984	11654
8-15	12530	7304	11472	9530
16-31	7111	3523	6693	5287
32-63	2589	1599	2513	2403
64-127	1004	742	1000	991
128-255	496	400	497	492
256-511	269	212	269	269
512-1023	85	80	85	85
1024-2047	31	31	31	31
2048-4095	12	11	12	12
4096-8191	3	3	3	3
8192-16383	2	2	2	2
16384-32767	1	1	1	1
32768-65535	0	0	0	0
65536-131071	3	3	3	3

Below 64 the data begins to really differentiate. In the row 4-7, the pattern recognition method and disassembly* method have removed half of the addresses, and the disassembly method has cut it by a third. In the row 1, the disassembly method cuts it by nearly a half, and the other methods cut it by nearly 3/4ths.

So, removing streams removed addresses that were being used only a few times. The more an item is accessed, the less likely it is to be removed. (The pattern recognition method is worse on this than the other two methods, but the statement still holds.)

The addresses that are accessed only a few times make up a large percentage of compulsory misses, but make up a small percentage of all references. Table 6 shows traces broken down by percentage of all references and the percent removed.

Overall, addresses accessed 7 or fewer times make up 19.66% of the trace and 35% of their access were identified as parts of streams by the disassembly method. For the dis-

TABLE 6. Percentage of All Accesses Broken Down by Number of Accesses to Each Address

# of References	Not Removed	Pattern Recog	% Removed	Disasse mbly	% Removed	Disasse mbly*	% Removed
1	4.32	1.08	75	2.70	38	1.23	71
2-3	7.04	2.35	67	4.49	36	2.76	61
4-7	8.30	3.31	60	5.68	32	4.24	49
8-15	9.67	5.31	45	8.93	8	7.10	27
16-31	11.20	5.39	51	10.63	5	8.24	26
32-63	7.96	5.03	39	7.73	3	7.41	7
64-127	6.25	4.72	25	6.22	0	6.17	1
128-255	6.24	5.02	20	6.23	0	6.17	1
256-511	6.73	5.39	20	6.73	0	6.72	0
512-1023	4.22	3.97	6	4.21	0	4.21	0
1024-2047	3.20	3.22	0	3.20	0	3.20	0
2048-4095	2.81	2.51	11	2.81	0	2.81	0
4096-8191	1.34	1.34	0	1.34	0	1.34	0
8192-16383	1.58	1.58	0	1.58	0	1.58	0
16384-32767	1.25	1.25	0	1.25	0	1.25	0
32768-65535	0.00	0.00	0	0.00	0	0.00	0
65536-131071	17.87	17.87	0	17.87	0	17.87	0
Stream Accesses	0.00	30.69		8.36		17.71	

assembly method when quasi-streams were removed, the factor was 58%, and for the pattern recognition method, the factor was 66%

The pattern recognition method was much more aggressive than either of the other methods, removing 20% of the accesses from addresses that were accessed 64 to 511 times, and 11% of the accesses from addresses accessed 2048 to 4905 times. This indicates it is removing data that might be much better kept in the cache.

By removing a small number of accesses that cause a large number of the compulsory misses, the SMC removes pollution from the data cache. This should increase the temporal locality of the data cache. In the next section, we'll see that it not only increased the temporal locality, it also decreased the spatial locality of the trace.

6.3.3 Temporal and Spatial Locality

Temporal and spatial locality are concepts rather than measurable quantities. Temporal locality is the concept that values that are used in the near future are likely to be items that were used in the near past. Spatial locality is the concept that values that are used in near future are likely to lie near items that were used in the near past. Although we could not directly measure these concepts, we attempted to measure their effect on caches.

To measure the effect of temporal locality, we chose a cache with a cache line of one word. This meant cache hits only occurred when a specific word was reused.

To measure the effect of spatial locality, we compared a cache with K lines of size N to a cache of K lines of size 2N. The increase in hit rate could only occur when two smaller sized lines fit into one of the larger lines, freeing up a cache line for something else; this indicates the data was near each other.

Table 7 shows the hit rate for the caches varying by line size. Table 8 shows the increase in hit rate for each doubling of line size. The cache used for these results was a 2-way set associative cache with 2048 lines. The line size was then varied from 4 bytes (1 word) to 256 bytes (64 words). The first level cache in the Alpha system is 2-way set associate with 2048 lines and a line size of 32 bytes (8 words).

TABLE 7. Hit Rate for Varying Cache Line Sizes

Cache Line Size (In Words)	Original	Pattern Recog	Disassembly	Disassembly*
1	78.00	90.25	81.94	88.07
2	86.72	92.82	89.65	92.49
4	91.65	95.22	94.16	95.41
8 (Size for Alpha 21264)	94.92	96.72	96.53	97.04
16	97.43	97.84	97.97	98.13
32	98.55	98.50	98.77	98.76
64	99.10	98.97	99.20	99.15

Note how much higher the hit rate for a cache with a 1 word line size is for all the traces with stream accesses removed. The disassembly method increases the hit rate by 4 percent, and the other methods by 10% and 12%! This is a huge increase in temporal locality. In other terms, the disassembly method changed the miss rate from 22% to 18.06%, or a cut of 17.9%! The other methods cut the miss rate by 45.8% and by 55.7%.

TABLE 8. Percentage Increase in Hit Rate per Doubling of Cache Line Size

Percentage Increase in Hit Rate	Original	Pattern Recog	Disassembly	Disassembly*
from 1 to 2 words	8.72	2.57	7.71	4.42
from 2 to 4 words	4.93	2.40	4.51	2.92
from 4 to 8 words	3.27	1.50	2.37	1.63
from 8 to 16 words	2.51	1.12	1.44	1.09
from 16 to 32 words	1.12	0.70	0.80	0.63
from 32 to 64 words	0.55	0.47	0.43	0.39

Also notice in Table 8 that whenever the line size is doubled, the original trace always gets the largest increase in hit rate. This indicates that the original trace has more spatial locality than the other traces. It appears that the pattern recognition method produced the trace with the least spatial locality, followed by disassembly* and finally by disassembly.

The L1 cache in the Alpha 21264 has a 32 byte line size which achieves a hit rate of 94.92% on this test. The disassembly method, our lower bound, achieves a hit rate of 94.165 with a 16 byte line size. The other two methods have hit rates above 95% for the 16 byte lines size. So by decreasing spatial locality, the SMC allows the data cache to achieve similar performance with a smaller cache line.

7.0 Conclusions

The goal of this research was to quantify the change in data cache performance when an SMC was added to the system. We chose to measure that performance by simu-

lating caches on memory address traces and to approximate the SMC by removing stream references from the traces.

Two methods were used to remove stream accesses: by recognizing loops in the traces and removing stream accesses from those loops, and by identifying the instruction causing each of the data accesses and removing accesses that belonged to loads or stores that showed stream characteristics. Additionally, traces which had quasi-streams removed were also generated because they might be able to be handled by the SMC system by either added hardware or by compiler optimizations.

These traces were then simulated on a cache layout modeled after the Alpha 21264, and the number of misses was recorded with the caches already initialized. The number of misses in the L1 cache was cut in the range of 27% to 46%, and the L2 cache showed a cut of 7% to 17%.

The increase in cache performance was due to an increased temporal locality in the trace. This increase in temporal locality resulted from the removal of a large number of polluting accesses which would be serviced by the SMC.

Spatial locality was drastically cut in the traces with streams removed. This indicates that streams and quasi-streams are the cause of a high percentage of the spatial locality in memory accesses patterns. It also indicates that a cache with a smaller line size in an SMC system would not be penalized as much as the same cache in a conventional system.

In conclusion, because streams make up a large percentage of all compulsory misses and have poor temporal locality, they cause cache pollution. By adding an SMC, those accesses are handled in a more efficient manner. The SMC also keeps the polluting accesses out of the cache, which dramatically improved cache performance. Additionally, the decrease in spatial locality allows the smaller data cache line, and, therefore, smaller

data caches, without a significant impact on performance. The increased data cache performance, along with the SMC's proven increase in bandwidth for streams, leads us to believe that the SMC will speed up memory accesses, even for non-scientific code such as the Kenbus benchmark.

References

[1] M. E. Benitez and J. W. Davidson, "Code Generation for Streaming: An Access/Execute Mechanism", Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991, pages 132-141.

[2] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson, "On the Accuracy of Memory Reference Models", International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Vienna, Austria, May 1994.

[3] Performance Evaluation Laboratory web page, http://pel.cs.byu.edu/

[4] L. Gwennap, "Digital 21264 Sets New Standard', Microprocessor Report; MicroDesign Resources, October 28, 1996, Available at http://www.digital.com/semiconductor/microrep/digital2.htm

[5] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach (Second Edition)", San Francisco: Morgan Kaufmann Publishers, 1996, page B-22.

[6] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", Digital Equipment Corporation: WRL Technical Note TN-14, Available by email to WRL-Techreports@decwrl.pa.dec.com.

[7] S. A. McKee, "Maximizing Memory Bandwidth for Streamed Computations", Ph.D. thesis, University of Virginia, May 1995. Available through http://www.cs.virginia.edu/techrep

[8] S. A. McKee, C. W. Oliver, Wm. A. Wulf, K. L. Wright, J. H. Aylor, "Evaluation of Dynamic Access Ordering Hardware", International Conference on Supercomputing, Philadelphia, PA, April 1996.

[9] The SMC web page, http://www.cs.virginia.edu/~wm/smc.html

[10] S. Mehrotra and L. Harrison, "Examination of a memory access classification scheme for pointer-intensive and numeric programs", International Conference on Supercomputing, Philadelphia, PA, April 1996.