Appears in the proceedings of the
21st Annual International Symposium on Computer Architecture, April 1994.

Evaluating Stream Buffers as a Secondary Cache Replacement [*]

Subbarao Palacharla
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706
subbarao@cs.wisc.edu

R. E. Kessler
Cray Research, Inc.
900 Lowater Rd.,
Chippewa Falls, WI 54729
richard.kessler@cray.com

Abstract

Today's commodity microprocessors require a low latency memory system to achieve high sustained performance. The conventional high-performance memory system provides fast data access via a large secondary cache. But large secondary caches can be expensive, particularly in large-scale parallel systems with many processors (and thus many caches).

We evaluate a memory system design that can be both cost-effective as well as provide better performance, particularly for scientific workloads: a single level of (on-chip) cache backed up only by Jouppi's stream buffers [10] and a main memory. This memory system requires very little hardware compared to a large secondary cache and doesn't require modifications to commodity processors. We use trace-driven simulation of fifteen scientific applications from the NAS and PERFECT suites in our evaluation. We present two techniques to enhance the effectiveness of Jouppi's original stream buffers: *filtering* schemes to reduce their memory bandwidth requirement and a scheme that enables stream buffers to prefetch data being accessed in large strides. Our results show that, for the majority of our benchmarks, stream buffers can attain hit rates that are comparable to typical hit rates of secondary caches. Also, we find that as the data-set size of the scientific workload increases the performance of streams typically improves relative to secondary cache performance, showing that streams are more scalable to large data-set sizes.

1 Introduction

A key design question for any computer system is: what kind of memory hierarchy should be provided? Conventional high-performance workstations (circa 1993) contain a processor with an on-chip cache augmented by an off-chip (secondary SRAM) cache of a megabyte or more. We consider replacing the secondary cache with Jouppi's stream buffers [10]. Stream buffers require much less hardware to implement, yet we find that they can provide performance similar to a large secondary cache for scientific codes. Some of the cost saved by replacing the expensive secondary cache with cheaper stream buffers can be applied towards implementing more plentiful main memory bandwidth, and the resulting system will likely have both significantly higher overall system cost-efficiency and performance, particularly for typical scientific codes that have regular access patterns. Memory system efficiency is particularly critical within the context of large-scale parallel machines (1K processors or more) because the costs of any inefficiencies are magnified by the scale of the system. Gigabytes of SRAM are required to implement the conventional workstation memory system design for each processor in these systems; this is an exorbitant cost if the caches are not being effectively used.

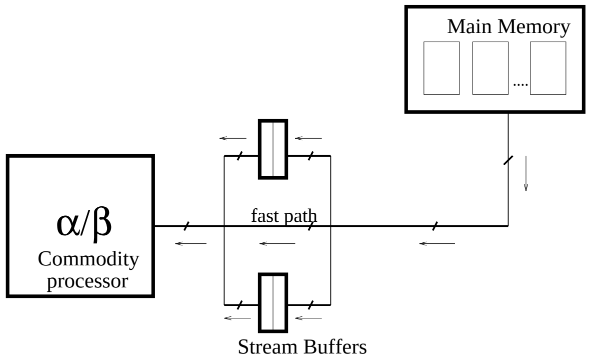


Figure 1: Logical organization of a typical uniprocessor
This figure displays our default system assumptions. Stream buffers prefetch data from main memory and make it available to the processor when a on-chip cache miss occurs. Note the existence of a *fast* path to memory bypassing the stream buffers; this is used when the data is not found in the stream buffers.

Stream buffers are FIFO prefetch buffers that prefetch cache blocks. Figure 1 illustrates the logical organization of a typical uniprocessor (or one of the processors in a large parallel system). For our simulations, we assume a commodity microprocessor that is backed up only by streams and a main memory. Streams prefetch cache blocks from the

[*]This work was supported in part by National Science Foundation grants MIPS-8957278 and MIPS-9225097, a summer internship at Cray Research, Inc., and a Wisconsin Alumni Research Foundation Fellowship.

main memory resulting in faster service of on-chip misses than in a system with only on-chip caches and main memory. Stream buffers will be most effective in systems with “sufficient” main memory bandwidth since some extra memory bandwidth is inevitably wasted by unnecessary prefetching, though streams can still be effective in a wide range of systems when the system limits memory bandwidth wastage via the *filtering* technique we introduce in this paper. Compared to secondary caches, stream buffers require very little logic, and we find that they scale better with larger scientific data sets.

This study is particularly timely since stream buffers have recently become commercially available. MacroTek [7] announced a memory controller chip for PowerPC based systems (a commodity microprocessor-based system) that incorporates stream buffers. The Macrotek implementation is similar to the original streams implementation we consider, except that it allows partitioned instruction and data streams.

We evaluate stream buffers for a large number of scientific application codes (fifteen applications), and determine the types of these programs that benefit most from streams. We show that for the majority of our programs stream buffers can reach good performance levels (hit ratio $\geq$ 50%). We also show how stream buffers could result in considerable inefficient use of memory bandwidth and how this can be improved by adding a *filter*. We present an implementation to extend the original streams to handle the case of non-unit stride memory accesses. We also compare stream buffer performance to that of secondary caches, indicating the relatively better scalability of streams to larger data set sizes.

The remainder of this paper is divided into eight sections. The next section describes related work. In section 3, we describe stream buffers. Section 4 describes the simulation methodology and framework. Section 5 gives simulation results for the performance of the original stream buffers, Section 6 describes the technique to reduce the memory bandwidth requirement of streams, Section 7 presents the scheme for detecting non-unit strides, and finally Section 8 compares streams to secondary caches for varying data set size. We draw conclusions in Section 9.

2 Related work

Many interesting prefetching studies appear in the literature. Prefetching strategies can be broadly classified into two groups: hardware based and software based.

Baer and Chen [1] proposed an on-chip scheme that detects strides in program references using history buffers. A hardware table (maintained as a cache), called the *reference prediction table*, keeps currently active load/store instructions and predicts future references. Fu and Patel [6] use the stride information encoded in vector instructions to prefetch in vector processors. They also suggest a scheme [5] for scalar processors that is similar to the Baer and Chen scheme. Another similar scheme is suggested by Sklenar [13]. Note that all of these hardware schemes make use of the program counter (PC) of the load/store instruction to implement prefetching. This is a significant disadvantage since it requires that commodity processors be modified to insert prefetch logic. Rambus Inc. has developed a memory system [8] that consists of a small (1 KB) prefetching secondary cache backed by high bandwidth Rambus DRAMS. They find that for typical corporate applications their cache achieves hit rates that are comparable to that shown by conventional Pentium system implementations with a 256 KB secondary cache and a 64-bit interleaved DRAM memory. Smith [14] evaluated schemes based on the one-block-lookahead (OBL) policy of prefetching block *i + 1* whenever block *i* is referenced. So and Rechtschaffen [16] suggest using a reference to a non-MRU (most recently used) block to trigger prefetches. As an extension to OBL, Jouppi suggested stream buffers [10]. Jouppi suggested using stream buffers on-chip to prefetch data at the maximum bandwidth of the second level cache. Smith and Hsu studied instruction cache prefetching in supercomputers (e.g. [15]).

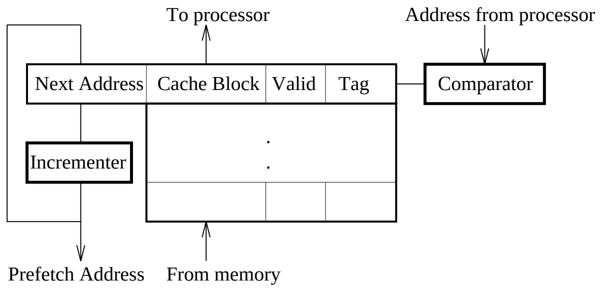


Figure 2: Stream buffer

A stream buffer has one or more entries, where each entry consists of a cache block of data, tag for the cache block and a valid bit. In addition, an incrementer is used to generate prefetch addresses and a comparator is used to match the miss address with the tag of the cache block at the head of the buffer.

Several schemes for compiler prefetching of data have been suggested. Porterfield et. al. [4] looked at prefetching array references within inner loops and used a simple heuristic of prefetching cache blocks a single loop iteration in advance. Mowry, Lam and Gupta [12] present a compiler algorithm to perform prefetch insertion. Their compiler takes into account data reuse to eliminate unnecessary prefetches. They show that selective prefetching is better than indiscriminate prefetching. While more flexible than hardware prefetching, software prefetching has a few disadvantages. Prefetch instructions require extra cycles for their execution. Perhaps even more importantly, they consume external or pin bandwidth of the commodity processor chip. Also, software may not be able to predict conflict or capacity cache misses, so unnecessary prefetches may be executed while the data is already in the cache.

3 Stream buffers

Jouppi first proposed the notion of stream buffers or streams [10]. They are FIFO prefetch buffers that prefetch consecutive cache blocks starting at a given address. Each entry of a stream buffer consists of a tag, an available bit, and a cache block as shown in Figure 2. When a reference misses in the on-chip cache, it allocates a stream and prefetches cache blocks starting at the miss target. The adder generates the address of the next cache block to be prefetched. When a cache block returns from main memory, the stream buffer hardware fills the tag and data fields of the entry and sets the available bit.

While Jouppi considered stream buffer prefetching from a large secondary cache into a primary cache, we instead consider prefetching directly from the main memory into buffers close to the processor chip.

Subsequent primary cache misses compare their address against the head of the stream buffer. If the reference hits in the stream buffer, the processor pulls the cache block into the primary cache. Write-backs bypass the streams and on their way to memory invalidate any stale copies that might be present in the streams. Compared to second level caches, stream buffers require little hardware. Each buffer needs a comparator and an adder in addition to a small amount of SRAM for the cache blocks. Also, the access time for stream buffers can be smaller than that of second level caches as there is no RAM lookup involved.

Since most programs access more than one array inside a loop, one could potentially benefit by using more than one stream in parallel. (as Jouppi also recognized [10]). *Multi-way* streams help in prefetching multiple data streams concurrently. The primary cache miss address is compared with the head of each stream in parallel. If the reference hits in one of the streams, the cache block is transferred to the primary cache; otherwise, the oldest stream is flushed and reset to prefetch from the miss address. We assume that a least recently used (LRU) replacement policy selects the stream to be reallocated. We have found the required number of streams to be sufficiently small (eight or less) that the parallel search mentioned above should not cause any significant access time increase.

Two important design parameters for stream buffers are the number of streams and the *depth* of each stream. The number of prefetched entries in each stream is called the depth of the stream. The optimal depth depends largely on the characteristics of the memory system that backs up the processor. A stream should be deep enough so that it can cover the main memory latency and supply data to the processor at its maximum rate. Since we wish to make as few assumptions about the underlying memory system as possible, we will assume a constant stream buffer depth of two. Henceforth, we will use the words stream, stream buffer and buffer interchangeably.

4 Methodology

4.1 Benchmarks and simulation environment

We used trace driven simulation as our evaluation methodology. We used *Shade* [17] to generate address traces of primary cache misses. We fed these traces to a stream buffer simulator which generates hit rate and other relevant statistics for the program. We used time sampling [11] to reduce the size of the trace files. We switched tracing on and off for 10,000 and 90,000 references, respectively, so that we sampled 10% of the trace. We used fifteen scientific applications, listed in Table 1 from the PERFECT [3] and NAS [2] suites, as our benchmarks. These Fortran programs were first converted to C using “f2c” and then compiled using “gcc” (version 2.4.3) with the -O2 option. We traced complete program runs. The number of instructions executed by each application varied from a few hundred million to a few billion.

Simulations were done assuming 64K I + 64K D 4-way set associative caches. The write policy of the data cache is write-back and write-allocate. The caches use a random replacement policy. We think this cache configuration is representative of what future processors will have. Also, the associativity minimized the effect of cache conflicts, so that we could focus on stream buffers. (In a direct-mapped cache, Jouppi’s victim buffers may also be needed.)

Table 1 shows the base performance of the benchmarks used. The table shows that in general, for the input sizes we used, the PERFECT codes show much lower primary cache miss rates than the NAS codes. The low miss rates may be partially explained by the small data set sizes selected for the simulations to complete within a reasonable period of time. At the same time, for four of the benchmarks we found larger data set sizes improved stream buffer performance (as we show in Table 4). It should be mentioned that we used the benchmark codes “as is” and did not modify them to make efficient use of stream buffers.

4.2 Performance metric

We use stream hit rate as our primary performance indicator. There are a number of reasons for using stream hit rate rather than metrics such as total execution time or effective CPI. First, hit rates indicate the maximum benefit that streams can provide. Second, there were no previous results (other than Jouppi’s [10] original results) to indicate what kind of stream buffer performance to expect for scientific workloads. Consequently we thought it was important to study a wide variety of benchmarks. Third, we did not want to make this paper too specific to any particular memory system design details. Also, we think that hit rate is an accurate metric for the kind of target systems we have in mind; systems for which memory bandwidth is “sufficiently” greater than the load data requirements of the processor. (An example target

Name	Description	Input Data Set	Data Set Size(Mb)	Data Miss Rate(%)	MPI
NAS					
embar	Embarassingly parallel	$n = 2^{20}$	1.0	0.28	0.10
mgrid	Multigrid kernel	32 X 32 X 32 grid	1.0	0.84	0.08
cgm	Smallest eigen-value of a sparse matrix	1400 X 1400 matrix, 78148 non-zero elements	2.9	3.33	1.43
fftpde	3-D pde solver using FFT	64 X 64 X 64 complex array	14.7	3.08	0.50
buk	Integer sort	64K integers, maxkey = 2048	0.80	0.53	0.20
appsp	Fluid dynamics	24 X 24 X 24 grid, 50 iterations	2.2	2.24	0.38
appbt	Fluid dynamics	18 X 18 X 18 grid, 30 iterations	4.2	1.88	0.45
applu	Fluid dynamics	18 X 18 X 18 grid, 50 iterations	5.4	1.26	0.18
PERFECT					
spec77	Weather Simulation	-	1.3	0.50	0.15
adm	Air pollution	64 X 1 X 16 grid, 720 time steps	0.6	0.04	0.00
bdna	Nucleic acid simulation	500 molecules, 20 counter ions	2.1	1.39	0.42
dyfesm	Structural dynamics	4 elements, 1000 time steps	0.1	0.01	0.00
mdg	Liquid water simulation	343 molecules, 100 time steps	0.2	0.03	0.01
qcd	Quantum chromodynamics	12 X 12 X 12 X 12 lattice	9.2	0.16	0.06
trfd	Quantum Mechanics	-	8.0	0.05	0.00

Table 1: Benchmark Characteristics

This table describes the benchmarks used in this paper. The first eight programs are from the NAS suite. The rest were selected from the PERFECT suite. The fourth column gives the data set size of the benchmark as returned by the Unix utility *size*. The fifth column gives the primary cache miss rate assuming 64KB, 4-way on-chip instruction and data caches. The final column shows the number of misses per instruction in percentage for the same cache configuration.

system is the Cray T3D, for which the available raw main memory bandwidth is 600 MB/sec while the maximum off-chip processor load bandwidth is 320 MB/sec.)

5 Performance of unit stride-only streams

While it is simple enough to understand the usefulness of streams for small kernels, it is an entirely different question as to how well stream buffers will perform on larger examples that include real code. Figure 3 shows how hit rates vary with the number of streams for our benchmarks. Hit rates here are the fractions of on-chip misses that hit in the streams. The stream buffers are unified (i.e. they prefetch both instructions and data). Partitioning the streams into separate instruction and data streams was not beneficial since the relatively large on-chip instruction cache resulted in very few instruction misses.

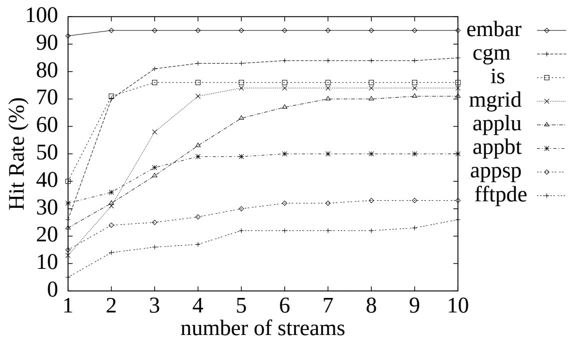
From Figure 3 we can see that majority of the benchmarks show hit rates in the 50-80% range. Also, hit rates plateau as the number of streams is increased. The number of streams at which the hit rate saturates is related to the number of unique array references in the program loops of the benchmark. For our benchmarks, seven to eight streams suffice. *fftpde* and *appsp* from the NAS suite perform poorly as they have a large number of non-unit stride references. Similarly, *adm* and *dyfesm* show low hit rates since a high percentage of the references made by these programs reference data via array indirections (scatter/gather). Surprisingly *cgm* exhibits good stream performance even though it is a sparse matrix program that has a significant number of array indirections.

How good are hit rates in the 50% - 80% range? Values of *local* hit rates for second level caches are in the 70% - 85% range [9] for "typical" applications. Also, for scientific codes this number may often be lower due to the lack of temporal locality in these codes. Hence, the fact that streams achieve comparable, though perhaps slightly lower, hit rates suggests their use as a viable and cost-effective alternative to huge second level caches. (We do more comparison with caches in section 8.)

Compared to secondary caches, streams require more memory bandwidth. This is because the unnecessary prefetches made by streams consume memory bandwidth. If NUP represents the number of useless prefetches, NC the number of cache misses, and NS the number of stream misses then the extra bandwidth (EB) can be quantified as follows:

$$\begin{aligned} EB &= NUP/NC \\ &= (NS * depth)/NC \\ &= stream\, miss\, ratio * depth \end{aligned}$$

By a *stream miss* we mean a cache miss that also misses in the streams. Whenever a stream is re-allocated, it could have up to $depth$ prefetches that have to be flushed. Hence, the total number of useless prefetches will be the product of the number of stream allocations (this is equal to the number of misses since a stream is allocated on every miss) and $depth$.



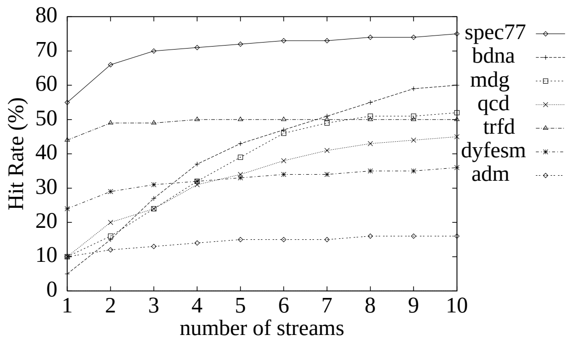


Figure 3: Stream buffer performance of the benchmarks

This figure shows how the hit rate varies with the number of streams. Here hit rate is the fraction of on-chip misses that hit in the streams. Using multiple streams helps in *locking* on to interleaved streams of data.

Benchmark	**Extra Bandwidth Required (%)**
NAS	
embar	8
cgm	30
mgrid	36
fftpde	158
is	48
appsp	134
appbt	62
applu	38
PERFECT	
spec77	44
adm	150
bdna	68
dyfesm	108
mdg	76
qcd	74
trfd	96

Table 2: Extra Bandwidth consumed by ordinary streams

This figure shows the amount of memory bandwidth wasted by ordinary streams as a percentage of the actual memory bandwidth required by the program in the absence of streams. This wastage is due to the speculative nature of the prefetching scheme.

Table 2 shows the extra bandwidth required by streams. From the table it is clear that ordinary streams, depending on the program, could waste a lot of memory bandwidth. This is especially true for programs for which streams do not perform well (low hit rates). For example, for *trfd* the extra bandwidth required is as high as 96%. Since memory bandwidth is not free it is desirable to reduce the amount of extra bandwidth required by streams. Also, it would be nice if we could do this with at most a slight reduction in hit rate. The next section describes a technique for doing this.

6 Reducing the Memory Bandwidth Requirements of Stream Buffers

To reduce wasted bandwidth we have to avoid useless prefetches (i.e. we have to prefetch with greater accuracy [16]). One way to avoid unnecessary prefetches is to allocate a stream only when a particular reference shows promise of belonging to a stream. The scheme we use to reduce memory bandwidth wastage filters away isolated references and does not present them to the stream buffers. This can be done using the following allocation policy for streams - a stream is allocated when there are misses (note that a miss here means the reference missed both in the primary cache and the streams) to consecutive cache blocks. For example, if there is a miss on a reference to cache block *i* and then there is a miss on reference to cache block *i + 1*, only then will a stream be allocated for prefetching cache blocks *i + 2*, *i + 3*, and so on. A reference is considered to be isolated if there is no reference to the preceding cache block in the "recent" past.

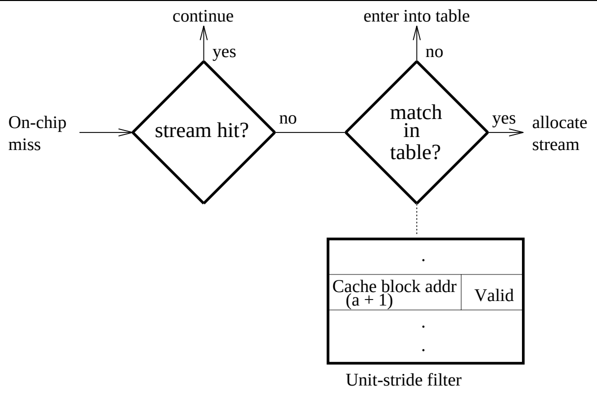


Figure 4: Filter mechanism

This policy can be implemented as follows: maintain a list of the N most recent miss addresses in a history buffer, but store *a + 1* for miss address *a*. For every primary cache miss that also misses in the stream buffers, the miss address is compared with the addresses stored in the history buffer. If there is a hit, this means that there were two references *a* and *a + 1* and there is a good possibility that there will be a reference to *a + 2* and so on. In this case a stream is allocated. However, if the miss address doesn't match in the history buffer, then *a + 1* is stored in the history buffer. (Since the history buffer is not infinite, the new entry might cause an old entry to be replaced.)

We call this history buffer a *filter*. N is the number of entries in the filter. It helps in filtering away isolated references. Our experimental results suggest that a filter of eight to ten entries is sufficient. Also, an entry in the filter need not be allocated for the entire duration of a stream; it is freed as soon as the stream is detected. Figure 4 illustrates the scheme.

The above scheme helps in two ways. It reduces the number of unnecessary prefetches and it prevents active streams from being disturbed. However, the total number of hits could be reduced, since now we allocate a stream only after observing the second reference of a stream of accesses.

We can calculate the extra memory bandwidth required with a filter as we did when the filter was not present. For a filter-based stream buffer, we allocate a stream only when the miss address matches in the filter. Hence, in this case the extra bandwidth (EB) required is

$$\begin{aligned} EB &= NUP/NC \\ &= (NS * filter\,hit\,ratio * depth)/NC \\ &= stream\,miss\,ratio * filter\,hit\,ratio * depth \end{aligned}$$

In this case a stream is allocated only when a reference misses both in the primary cache and the streams and hits in the filter. This explains the factor $filter\,hit\,ratio$ in the number of useless prefetches. The above expressions show that there is a trade-off between filter hit rate (but perhaps not stream buffer hit rate!) and the extra memory bandwidth required by streams.

6.1 Hit rates for filter-based unit stride streams

We studied how a filter affects the performance of stream buffers. We used ten streams for the experiments reported in the rest of this paper. Figure 5 shows how hit rate and EB, the extra bandwidth required, vary with the addition of a filter. For most of the benchmarks the filter was very effective in reducing EB; often the reduction is more than 50%. For example, the *trfd* hit rate remains almost the same while EB falls from 96% to 11%. In this case the filter is very successful at eliminating isolated references. Similarly, for *is*, *appsp* and *cgm* EB falls from 48% to 7%, 134% to 45%, and 30% to 13% respectively with almost no reduction in hit rate. In the case of *fftpde* the filter actually increased hit rate by preventing active streams from being disturbed, and EB also fell from 158% to 37%. On the other hand, for *appbt*, hit rate drops from 65% to 45% and EB falls only from 62% to 48%. This indicates that the filter may not be optimal for all applications, depending on the available memory bandwidth in relation to the processor demands.

Benchmark	**Length distribution (% hits)**				
	1-5	**6-10**	**11-15**	**16-20**	**>20**
NAS					
embar	1	0	0	0	99
mgrid	13	1	0	0	86
cgm	3	0	0	0	97
fftpde	41	0	0	0	59
is	4	2	1	0	93
appsp	5	0	11	0	84
appbt	63	0	0	0	37
applu	22	3	4	7	64
PERFECT					
spec77	14	1	1	0	84
adm	73	12	5	1	9
bdna	36	17	8	5	33
dyfesm	50	17	7	1	25
mdg	32	9	7	6	46
qcd	50	6	1	0	43
trfd	7	2	1	0	90

Table 3: Distribution of stream lengths

This figure shows how the stream lengths are distributed. Note that this distribution depends on the number of streams being used. We used ten streams for these experiments.

These variations in hit rates can be explained by looking at how the stream lengths are distributed. By stream length, we mean the number of references after which the regular pattern of accesses is broken. Stream length distributions are shown in Table 3. For most benchmarks stream lengths of less than 5 and greater than 20 constitute a major fraction of the hits. The programs that have a large concentration of small stream lengths show a greater reduction in hit rate when the filter is used. This is obvious since the filter requires two references for verifying a unit-stride pattern of accesses. For example, in the case of *appbt* the fact that 63% of the hits are from stream lengths of less than 5 explains why the filter reduces the hit rate from 65% to 45%.

From the above results we conclude that a filter may often be a good idea, since in most cases it reduces the memory bandwidth requirement of streams for a small or negligible performance hit. At the same time if the program's memory bandwidth requirement is not high and the memory system is capable of supplying the extra bandwidth, the filter should be deactivated, since the stream buffer hit rate typically falls slightly with the filter.

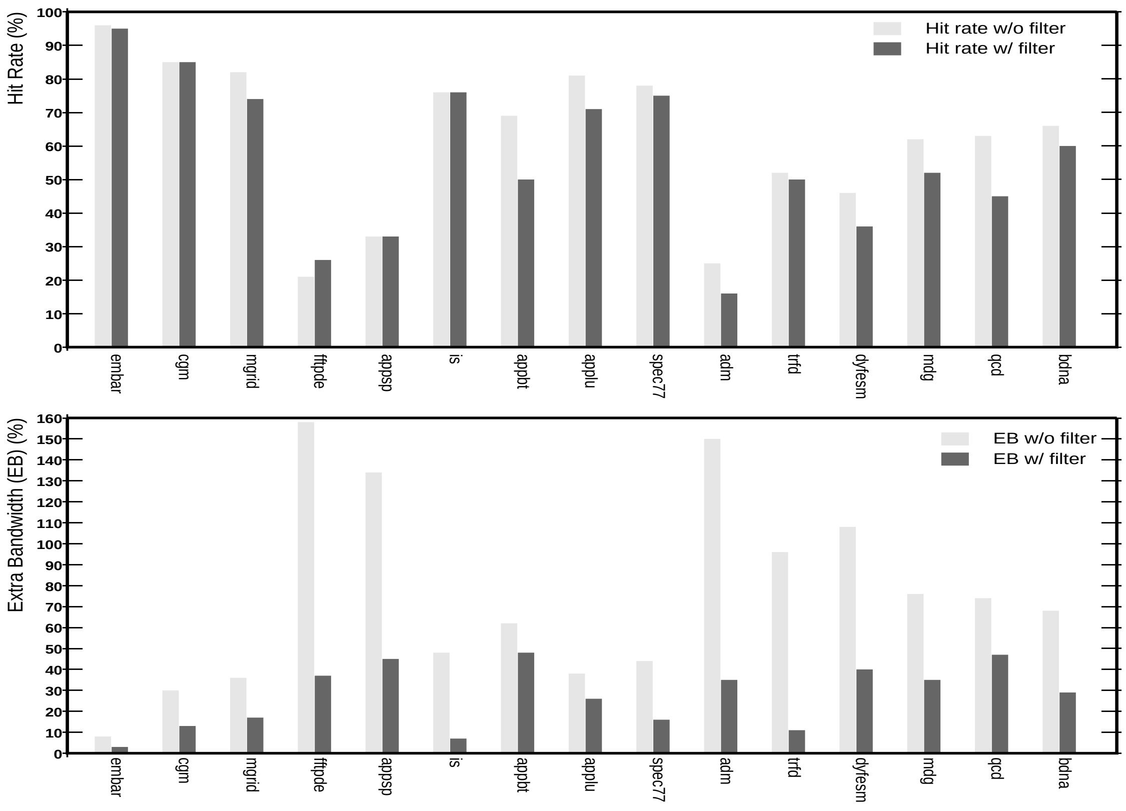


Figure 5: Performance of filter

This figure displays how the filter affects stream performance. The top graph shows the stream hit rate with and without the filter. The bottom graph represents the extra bandwidth required in each case. For this data we used ten streams and a filter of sixteen entries.

7 Detecting non-unit strides

A closer look at the benchmarks revealed that some of them, *appsp*, *fftpde*, and *trfd* contain significant percentages of large non-unit stride memory accesses. Streams, as proposed by Jouppi, are of little use in prefetching cache blocks being accessed in large non-unit strides. In this section we show ways to extend the original streams to detect non-unit strides.

Detecting non-unit strides off-chip is harder than detecting them on-chip. Once off-chip the only information one has are the physical addresses of the data references. For instance, Baer and Chen [1] use the *reference prediction table* with an entry for each load/store instruction for calculating strides. But since off-chip logic almost always does not know the the PC of the instruction that issued the reference, it is difficult to maintain a similar table off-chip.

We instead extend the basic stream buffer structure for prefetching cache blocks being accessed in non-unit stride. A stride field is added to maintain the prefetch stride. Also, the incrementer is replaced by a general adder (see Figure 2).

The basic idea behind our non-unit stride detection scheme is to dynamically partition the physical address space and detect strided references within each partition. Two references are within the same partition if their addresses have the same tag (higher order) bits. The processor (i.e. program) sets the size of the tag by storing a mask in a memory-mapped location. A history buffer, shown in Figure 6, is used to store the tags of the currently active partitions. We call this history buffer a non-unit stride filter. Also, we use a finite state machine (FSM) to detect the stride for references that fall within the same partition. The FSM we use is depicted in Figure 7. It verifies that the difference between the third and the second address is the same as the difference between the second and the first address. If so, the off-chip logic allocates a stream and sets its stride. Partitioning helps in grouping references to an array and analyzing them in isolation to detect strides.

The details of the non-unit stride detection scheme follow. We partition each word address into two parts: *czone* or the concentration zone, the size of which is set at run-time, and the tag. Each entry of the non-unit stride filter, in addition to the tag of the partition, has a few state bits, *last address* and *stride* fields which are required to implement the stride detecting FSM. At the end of three consecutive strided references a stream is allocated and the entry in the filter is freed. To minimize the effects this scheme has on the scheme for detecting the common case of unit-strides we use the non-unit

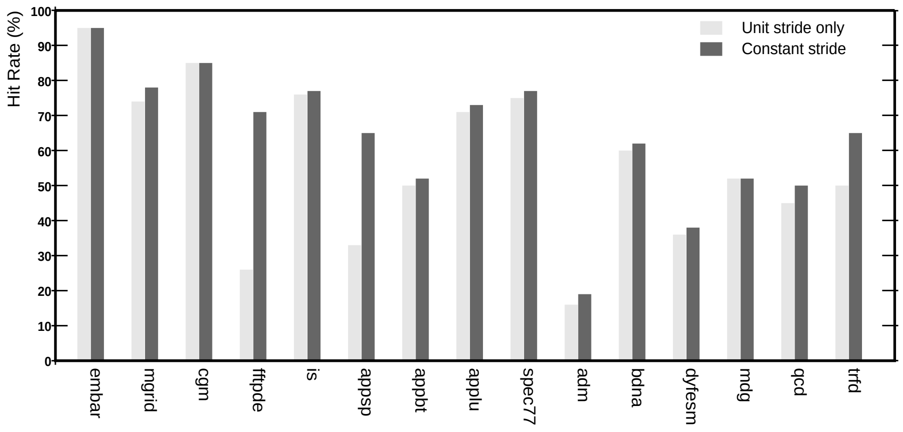


Figure 8: Performance of non-unit stride detecting scheme
This figure shows how the non-unit stride detection scheme performs. This data assumes ten streams and a non-unit stride filter, containing sixteen entries, backing a similar sized unit-stride filter.

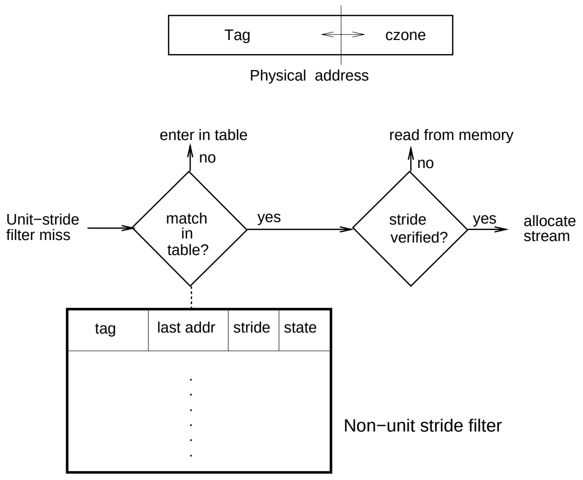


Figure 6: Scheme for detecting non-unit strides

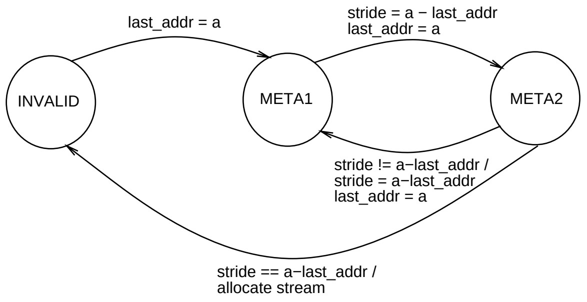


Figure 7: State Machine for detecting strides
This figure displays the finite state machine required to verify a non-unit stride. The FSM is triggered by a on-chip miss. The state of the FSM (*stride, last_addr*) is stored in a filter entry. *last_addr* stores the previous miss address and *stride* stores the current guess for the stride. Note that some of the transitions are conditional.

stride filter behind the unit-stride filter (i.e the non-unit stride filter processes references that miss in the unit-stride filter).

We considered other schemes to detect non-unit strides. One that showed similar performance is what we call the *minimum delta* scheme. Here, we cache the last N miss addresses and maintain them in a history buffer. When an on-chip miss occurs and it misses in the streams, we find the minimum distance (or delta) between the address and any of the entries in the history buffer. The delta is then used as a stride for the stream. The hardware requirements of this scheme make it less attractive than the partition scheme.

7.1 Performance of non-unit stride detecting scheme

Figure 8 shows how the partition scheme for detecting non-unit stride streams performs. From the figure we can see that for *fftpde*, *appsp* and *trfd*, programs which have a significant number of non-unit stride references, our scheme does well. For example, for *fftpde* the hit rate increases from 26% to 71%. Similarly for *appsp* and *trfd* the hit rate improves from 33% to 65% and 50% to 65%, respectively. Gains in other benchmarks are minor.

Figure 9 shows how hit rate varies with the size of the czone. It indicates that for *fftpde* the size of the czone should lie between 16 and 23 bits for the scheme to be effective. However, for the other two benchmarks *appsp* and *trfd*, a large value for the czone is sufficient to predict most of the non-unit stride references. This shows that one has to be careful in selecting the czone size; if the czone size is too small then three consecutive strided references will not fall in the same partition. On the other hand, if the czone is too large then references from more than one stream may fall into the same partition, and hence prevent stride detection. The optimal size for the *czone* is (a little more than) twice the stride of the references. Since the size of the czone depends on the stride

Benchmark	Input size	Stream hit-rate (%)	Minm. L2 cache size for same hit-rate
appsp	12 X 12 X 12	43	128 KB
	24 X 24 X 24	65	1 MB
appbt	12 X 12 X 12	50	512 KB
	24 X 24 X 24	52	2 MB
applu	12 X 12 X 12	62	1 MB
	24 X 24 X 24	73	2 MB
cgm	1400 X 1400, 78148	85	1 MB
	5600 X 5600, 98148	51	64 KB
mgrid	32 X 32 X 32	76	2 MB
	64 X 64 X 64	88	4 MB

Table 4: Stream buffers versus secondary cache

This table shows how the performance of streams and secondary caches vary with the input size. We used ten streams, a unit-stride filter of sixteen entries backed up by a non-unit stride filter of sixteen entries. For the secondary cache we considered associativities from one (direct-mapped) to four as well as block sizes of 64 and 128 bytes. Set Sampling [11] was used to determine the hit rate of secondary caches.

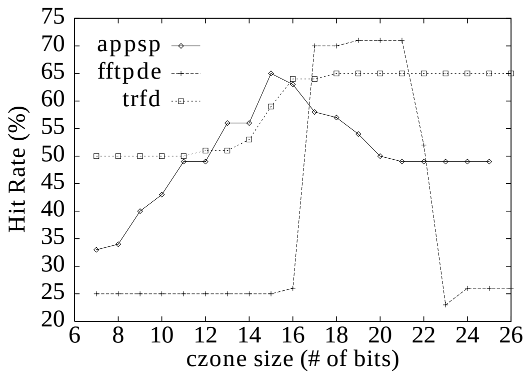


Figure 9: Hit-rate sensitivity to czone size

This graph shows how stream hit rate varies with the size of the czone. Only programs that contain significant percentage of non-unit stride references are shown here. This data assumes ten streams.

and the array dimensions (in the case of multi-dimensional array references), it is possible for the programmer or the compiler to set it to a suitable value.

8 Comparison with second level caches

For five benchmarks, *appsp*, *appbt*, *applu*, *cgm*, and *mgrid*, we compare how secondary cache performance and stream buffer performance scale with the input size. In particular, we determine the minimum size of the secondary cache required to obtain the same (local) hit rate as stream buffers. For the secondary cache we considered associativities from one (direct-mapped) to four as well as block sizes of 64 and 128 bytes. Our results, shown in Table 4, indicate that stream buffers typically scale better than secondary caches. For example, for *applu*, when the input size was increased, stream hit rate improved from 62% to 73% while the minimum secondary cache size for achieving the same hit rate doubled from 1MB to 2MB. For all the benchmarks except *cgm* there was very little temporal reuse and the cache size that had approximately the same miss ratio as streams is proportional to the data set size. This emphasizes that as the data set size for scientific programs increases, it may be more cost-effective to exploit the regular pattern in memory references rather than to fit a large data set in a huge second level cache. The reason for the anamolous behavior of *cgm* is that for the larger data set the sparse matrix had a very irregular distribution of elements. This benchmark also shows where streams might not perform well - programs that involve widely-scattered array in-directions.

A caveat to the comparison of this section is that it isn't entirely fair to directly compare streams and caches via their hit ratios since a stream buffer entry may have been prefetched but the data hasn't returned from memory yet. In our stream results, we would call this a hit since the prefetch was correct, but the performance of this case could possibly be more similar to a cache miss since the processor's request for data must wait until the streaming data returns from main memory. The probability of this situation depends highly on the particular memory system design. We feel that in many realistic system designs the depth of the streams will be sufficient that most of the time the stream data will immediately available, so the direct comparison between hit rates is fair. We particularly feel this is a balanced comparison since, depending on the system design, stream buffer access time on hits may be lower than the access time of a cache on hits because stream buffers do not require a large RAM lookup.

9 Conclusions

In this paper we evaluated stream buffers for efficient memory system design with scientific codes. We showed that stream buffers can achieve hit rates that are comparable to the (local) hit rates of very large caches. We also presented schemes for reducing the memory bandwidth requirement of stream buffers. For the majority of the benchmarks we studied, a hit rate of greater than 60% using only 30% extra main memory bandwidth was achieved using ten streams. However, they did not perform as well for benchmarks that had a large number of irregular accesses (e.g. array indirections). We also extended streams to prefetch cache blocks being referenced in non-unit strides. For programs that have significant percentage of non-unit stride references our scheme is successful in detecting them. We found that as the data set size of the scientific codes increase, streams typically performed relatively better than large secondary caches. Hence, we conclude that stream buffers are a viable implementation option for regular scientific workloads and systems with "sufficient" memory bandwidth. We also conclude that stream buffers can be more economical than large secondary caches for scientific codes: the cost savings of stream buffers over large caches can be applied to increase the main memory bandwidth, resulting in a system with better overall performance.

10 Acknowledgements

We thank Jim Smith for his contributions, inspiration and review of an earlier draft. Thanks to Jim Goodman, Mark Hill, Tony Laundrie and Steve Scott for many interesting discussions. Greg Faanes, Tony Laundrie and Jim Schwarzmeier helped us with the benchmarks. Doug Burger, Satish Chandra, Shamik Das Sharma and T.N. Vijaykumar read an earlier draft of the paper and helped improve the presentation. Thanks also to the Cray employees who put up with all the simulations we required.

References

[1] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, November 1991. Also available as U. Washington CS TR 91-03-07.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[3] M. Berry, D. Chen, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The Perfect Club: Effective performance evaluation of supercomputers. *The International Journal of Supercomputing applications*, 3(3), December 1989.

[4] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[5] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proceedings of the 25th MICRO*, pages 102–110, 1992.

[6] John W. C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. In *The 18th Annual International Symposium on Computer Architecture*, pages 54–63, May 1991.

[7] Linley Gwennap. Macrotek to sell VME chip set for PowerPC. *Microprocessor Report*, pages 8–9, August 1993.

[8] Craig Hampel. Using Rambus technology in Pentium-based systems, 1993.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[10] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[11] R. E. Kessler, Mark D. Hill, and David A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. Technical Report CS 1048, University of Wisconsin-Madison, September 1991.

[12] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[13] Ivan Sklenar. Prefetch unit for vector operations on scalar computers. *ACM Computer Architecture News*, 20(4):31–37, September 1992.

[14] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[15] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing'92*, pages 588–597, 1992.

[16] Kimming So and Rudolph N. Rechtschaffen. Cache operations by MRU change. *IEEE Transactions on Computers*, C-37(6), June 1988.

[17] Sun Microsystems Laboratories, Inc. *Introduction to Shade*, April 1993.