

# Fetch Size Adaptation vs. Stream Buffer for Media Benchmarks

Weiyu Tang  
wtang@ics.uci.edu

Rajesh Gupta  
rgupta@ics.uci.edu

Alexandru Nicolau  
nicolau@ics.uci.edu

Alexander Veidenbaum  
alexv@ics.uci.edu

Department of Information and Computer Science  
University of California, Irvine  
Irvine, CA 92697-3425

## ABSTRACT

In current cache designs, cache size and line size are often fixed and determined by the spatial and temporal locality of benchmarks used to evaluate the targeted processors. Stream buffer and large fetch size are two techniques that exploit spatial locality to hide memory latency. In this paper, we compare the effectiveness of stream buffer and fetch size on media benchmarks. We have presented a dynamic prediction mechanism for fetch size adaptation. Our results show that fetch size adaptation is as effective as stream buffer and both can improve the performance of media benchmarks by up to 10%. Due to its simple hardware design, fetch size adaptation is a viable alternative to stream buffer in hiding memory latency.

## 1. INTRODUCTION

In current cache designs, a cache consists of multiple lines of equal size. For a cache with a fixed line size, the determination of the line size is based on the spatial and temporal locality of average benchmarks. Due to variations in locality in different applications and in different phases of an application, fixed line size limits cache's ability in locality utilization.

Stream Buffer (SB) and large fetch size are two techniques to exploit spatial locality. There are similarities between large fetch size and SB. Both prefetch data to hide memory latency. The SB prefetches data into dedicated buffers to avoid cache pollution. Large fetch size prefetches data into the cache directly. We conjecture that cache pollution can be kept to minimal by using large fetch size scrupulously.

While the SB can avoid cache pollution, it entails other prob-

lems. First, the data path needs modification because missing lines can be provided from either the SB or the next level memory. Second, to reduce the number of unnecessary prefetches, the cache is looked up before a prefetch request is sent to the next level memory. This increases cache port contention and may degrade performance. Third, there can be data inconsistency between the SB and the write buffer. Although the above problems can be dealt purely in hardware, they increase design complexity and require more time in design and verification, which is undesirable for media processors because time to market is critical.

An example of processor design, which increases processor functionality without adding extra design complexity, can be found in StrongARM [3]. StrongARM has a 4-entry SB that is under software control. Software decides when to use the SB and maintains the data consistency between the SB and the write buffer. The downside of this approach is that it shifts the burden to programmers.

We conjecture that if the performance by a cache with variable fetch sizes is comparable to that by a cache with a SB, then a cache with variable fetch sizes can be a simple alternative to the SB in hiding memory latency.

In this paper, we first brief related work in Section 2. Next, we evaluate the impacts of variable fetch sizes and SB on performance in Section 3. Then fetch size adaptation is presented in Section 4, followed by experimental results in Section 5. This paper is concluded in Section 6.

## 2. RELATED WORK

With widening speed disparity between the processor and the main memory, memory hierarchy is often the performance bottleneck of many applications. Efficient use of on-chip caches is crucial for performance. Prefetching is a technique commonly used to hide memory latency. The idea is to predict data access needs in advance so that a piece of data is loaded from the memory before it is actually needed. Different prefetching techniques mainly differ in three aspects:

- what miss patterns to follow
- when to prefetch
- where to store the prefetched data

In next line prefetch by Smith [9], the predictive assumption is that the line after current missing line will also miss and that line will be prefetched into a dedicated buffer.

In SB proposed by Jouppi [5], the prediction is that several lines after current missing line will also miss and they are prefetched into dedicated buffers. A SB consists of several cache line sized buffers with FIFO replacement policy. The SB is searched on a cache miss. On a hit in the SB, data can be provided from the SB and cache miss latency to the next level memory is eliminated. Then the SB is refilled to enable future hits in the SB. On a miss in the SB, several lines following the missing line are prefetched into the buffers. By using several buffers, long memory latency can be hidden. Jouppi also recognized the need for multi-way stream buffers to support multiple active streams.

Palacharla and Kessler [8] proposed several enhancements to the stream buffer. They proposed a filtering scheme to eliminate the number of prefetches and used a method to allow variable length strides. Zucker et al. [11] enhanced the SB with a stream cache to store the data evicted from the SB. It can further reduce cache pollution. Zucker et al. also investigated the effects of prefetching on media applications.

Fu and Patel [2] proposed a stride prediction table to keep track of the last address and the difference between the last address and the address before it. Data streams with non-unit strides can be prefetched.

Joseph and Grunwald [4] designed a Markov predictor with the assumption that the address streams seen in a program can be modeled by a Markov model and bases its prediction on the last values seen. A Markov model is a set of states and transition frequencies where each state has a probability of transition to another. Each transition from address A to B is assigned a weight.

Changing cache line size is another way to exploit spatial locality. While some processors such as MIPS 3000 [7] support multiple line sizes, the line size can only be changed at boot time to avoid high penalty in cache flush.

In our previous research [10], we have proposed an **Adaptive Line Size (ALS)** cache to take advantage of the changing locality in applications. Large line sizes are used when good spatial locality is detected and small line sizes are used when poor spatial locality is detected. Different cache line sizes can coexist at the same time.

There are several differences between the ALS cache and the **Adaptive Fetch Size (AFS)** cache proposed in this paper. The AFS cache can only exploit locality in the time domain. The cache has one fetch size at a time, but the size can change over time. In contrast, in the ALS cache, different cache lines can have different sizes at the same time. Hence

the ALS cache can exploit locality in both time domain and space domain. Although the ALS cache is more powerful, it has the following drawbacks: complicated prediction mechanisms, major modification of cache design, and high hardware implementation cost.

### 3. IMPACTS OF VARIABLE FETCH SIZES AND STREAM BUFFER

To evaluate the impacts of variable fetch sizes and SB on performance, we use the SimpleScalar toolset [1] to model a single in-order issue processor similar to StrongArm. The system configuration is shown in Table 1. A set of benchmarks from the MediaBench suite [6] is simulated and the benchmark description is shown in Table 2.

Parameter	Value
Processor	5-stage pipeline, single in-order issue
L1 I-cache	16KB, 4-way, 32B line, 1-cycle lat.
L1 D-cache	16KB, 4-way, 32B line, 1-cycle lat, fetch sizes-32B, 64B and 128B.
Memory	30-cycle lat.
L1-memory bus	8B wide, 2-cycle per transfer

Table 1: System configuration.

Figure 1 compares the miss rate by variable fetch sizes and by SB. The benchmarks are plotted in increasing order of miss rate. Half of the benchmarks-721\_dec, 721\_enc, adpcm\_c, adpcm\_d, gst, mpg\_dec and mpg\_enc, have small access footprints. For the rest 7 benchmarks, the miss rate by the optimal fetch size is equal to or slightly better than that by SB.

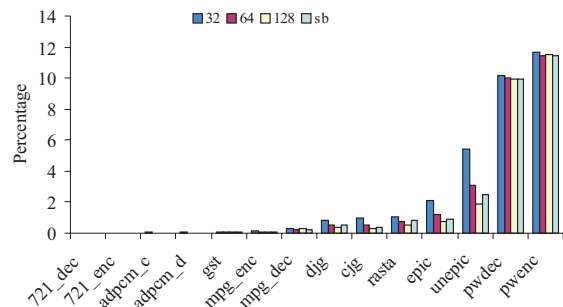


Figure 1: Miss rate: variable fetch sizes vs. SB.

Figure 2 compares the speedup by variable fetch sizes and by SB. The execution cycle with fetch size 32B is used as baseline. For benchmarks with small access footprints, there is virtually no change in speedup. As memory hierarchy is not performance bottleneck for these benchmarks, we will not further investigate these benchmarks for the rest of this paper.

Name	Description	Instr. (M)	Mem. (M)
721_dec	Voice decompression	409	75
721_enc	Voice compression	434	75
adpcm_c	Speech compression	11	0.8
adpcm_d	Speech decompression	9	0.8
gst	Postscript interpreter	1500	309
mpg_dec	MPEG decoding	1118	253
mpg_enc	MPEG encoding	1171	338
djg	Image decompression	37	10
cjg	Image compression	125	38.9
rasta	Speech recognition	39	11.8
epic	Data compression	54	7.7
unepic	Data decompression	7	1.9
pwdec	Public key decryption	18	5.2
pwenc	Public key encryption	32	8.5

Table 2: Benchmark description.

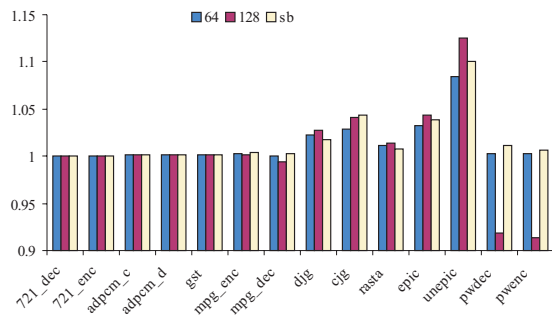


Figure 2: Speedup: variable fetch sizes vs. SB.

For the benchmarks with non-trivial miss rate, we can observe noticeable increase in speedup by both techniques. The speedup obtained with the optimal fetch size is comparable to that obtained with the SB. This indicates that varying fetch sizes is an alternative to the use of SB. However, there is no single fetch size that is the optimal for all the benchmarks. Some fetch sizes, such as 128B in *pwdec* and *pwenc*, show high performance degradation. Therefore, fetch sizes should be chosen carefully to avoid cache pollution.

Next, let us look at intra-application fetch size behavior. Each point in Figure 3 represents the miss rate during an interval of 200,000 memory accesses. In addition to 7 media benchmarks with non-trivial memory access footprints, *ijpeg*, an integer SPEC95 benchmark, is also shown for comparison.

In *ijpeg*, 64B and 128B are the optimal fetch sizes for different intervals, which means that the optimal fetch size for a benchmark may be suboptimal for some intervals. In contrast, media benchmarks show consistent fetch size behavior and one fetch size is the optimal for all the intervals in a benchmark. Therefore, the optimal fetch size for media benchmarks can be easily predictable at run-time.

## 4. FETCH SIZE ADAPTATION

The spatial locality in an application has a direct impact on cache performance. We can adapt fetch size with run-time locality prediction. Our prediction is interval based. Memory accesses are partitioned into intervals in time. The fetch size remains unchanged during an interval and the aggregate spatial locality during this interval is monitored. Then at the end of an interval, the fetch size for the next interval is predicted using the detected spatial locality.

In this section, we first describe the functionality of an AFS cache, whose fetch size can change over time. Then a mechanism for locality detection is presented, followed by fetch size prediction.

### 4.1 AFS Cache

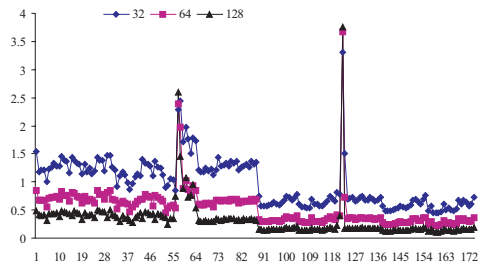
In an AFS cache, a fetch size is a multiple of cache line size and is a power of two. Changing a **Fixed Line Size** (FLS) cache to an AFS cache only needs small hardware support. Instead of hardwiring the fetch size, the fetch size is stored in a register, which can be set dynamically based on application locality.

Similar to a FLS cache, an AFS cache consists of several equal sized lines called **Physical Cache Line** (PCL). A PCL is an atom element in cache operations and the PCL size is used in tag generation and cache indexing. An address can be partitioned into three fields:

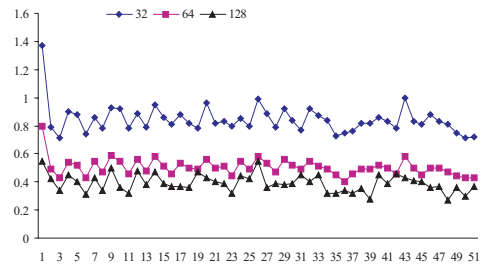
- tag
- index
- offset

Suppose the address width is  $A$  bits. For a  $2^W$ -way set-associative cache with cache size  $2^C$  bytes and line size  $2^L$  bytes, the size of field *tag*, *index* and *offset* is  $(A+W-C-3)$ ,  $(C-L-W)$  and  $(L+3)$  bits respectively. A PCL can be uniquely identified by a pair:

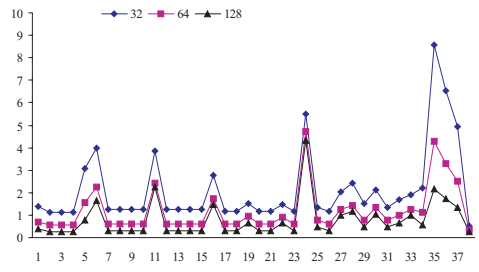
$$(way\_num, line\_num).$$



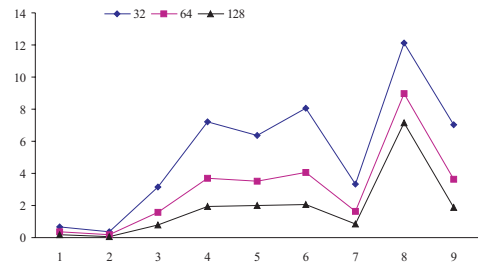
(a) Cjg



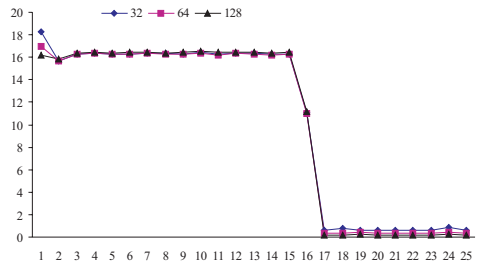
(b) Djg



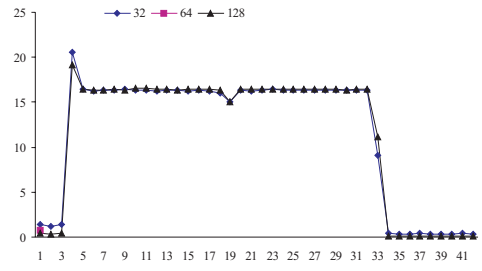
(c) Epic



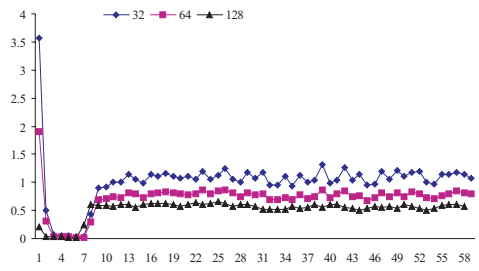
(d) Unepic



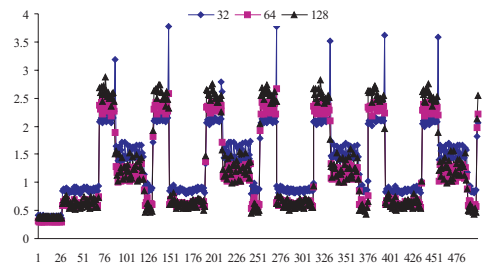
(e) Pwdec



(f) Pwenc



(g) Rasta



(h) Ijpeg

Figure 3: Miss rate in time with different fetch sizes.

On a miss-fetch, multiple continuous PCLs are filled. These PCLs form a Virtual Cache Line (VCL). The VCL size is equal to the fetch size and one miss-fetch will fill one VCL. The fetch size is  $2^V$  times the PCL size. A VCL can be uniquely identified by a triplet :

$$(way\_num, i * 2^V, (i + 1) * 2^V - 1)$$

where  $i * 2^V$  is the line number of the first PCL in this VCL and  $((i + 1) * 2^V - 1)$  is the line number of the last PCL in this VCL.

Two VCLs  $(way\_num1, i * 2^v, (i+1) * 2^v - 1)$  and  $(way\_num2, j * 2^v, (j + 1) * 2^v - 1)$  are called "neighboring" VCLs if they are part of a larger VCL with double line size. That is, they are "neighboring" if there exists an integer  $k$  such that one of the following conditions is satisfied:

- $i == 2 * k$  AND  $j == (2 * k + 1)$
- $i == (2 * k + 1)$  AND  $j == 2 * k$

Figure 4 shows the relationship between PCLs and VCLs in a direct-mapped cache. In this example, the fetch size is four times the cache line size. VCL (0, 0, 3) consists of PCLs (0, 0), (0, 1), (0, 2) and (0, 3). VCL (0, 4, 7) consists of PCLs (0, 4), (0, 5), (0, 6), and (0, 7). VCL (0, 0, 3) and VCL (0, 4, 7) are "neighboring" VCLs because they are part of a VCL (0, 0, 7), whose size is eight times the cache line size.

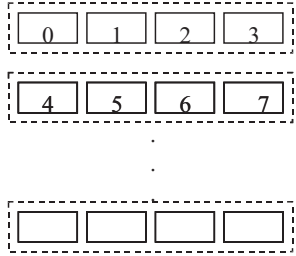


Figure 4: Relationship between PCL and VCL

Suppose there are two caches A and B. Cache A is a FLS cache and cache B is an AFS cache. Both caches have the same cache size and the same associativity. The line size of cache A is equal to the fetch size of cache B. There is a one-to-one mapping between one PCL in cache A and one VCL in cache B. It can be proven by induction that:

*If both caches are initially empty and the sequence of memory accesses to both caches are same, then both caches will always have the same data.*

As both caches always have the same data, a memory access will always hit or miss in both caches. Thus the performance of an AFS cache with fetch size  $lB$  is the same as the performance of a FLS cache with line size  $lB$ . By changing the fetch size dynamically, we are effectively having a cache with multiple physical line sizes.

## 4.2 Spatial Locality Detection

Spatial locality can be detected as follows:

- good spatial locality  
For two "neighboring" VCLs, if their tags are equal, then there is good spatial locality between them. A larger fetch size can fill all the PCLs in these VCLs and can potentially eliminate one cache miss in the future.

- poor spatial locality  
When a VCL is to be replaced, if either the first half or the second half of the PCLs in this VCL are unused, then this VCL doesn't have good spatial locality and unused PCLs are polluting the cache. A smaller fetch size can reduce cache pollution and increase cache utilization.

Spatial locality detection is done in parallel with miss fetch. Cache hit time, which is often on one of the critical paths in a processor design, is not affected. In addition, cache lookups for determining whether two VCLs have the same tag are done during cache misses and are unlikely to increase cache port contention.

## 4.3 Fetch Size Prediction

At the end of an interval, the fetch size for the next interval is predicted based on the aggregate spatial locality for all VCLs during this interval. The aggregate spatial locality can be measured using the following two parameters:

- $inc\%$  -the percentage of VCLs with good spatial locality.
- $dec\%$  -the percentage of VCLs with poor spatial locality.

Given:

$inc_{thresh}, dec_{thresh}, max_{fetch\_size}, min_{fetch\_size}$   
 $fetch\_size\_predict(fetch\_size, inc\%, dec\%)$

BEGIN

1. If  $inc\% > inc_{thresh}$  Then
  2.     If  $fetch\_size < max_{fetch\_size}$  Then
  3.          $fetch\_size = fetch\_size * 2$
  4.     Endif
  5. Else if  $dec\% > dec_{thresh}$  Then
  6.     If  $fetch\_size > min_{fetch\_size}$  Then
  7.          $fetch\_size = fetch\_size / 2$
  8.     Endif
  9. Endif
- END

Figure 5: Fetch size prediction.

Figure 5 shows a simple algorithm to predict the next fetch size. To simplify the prediction, the candidates for next fetch size are: current fetch size, the immediately larger fetch size and the immediately smaller fetch size. If  $inc\%$  is larger than  $inc_{thresh}$ , the fetch size will double for the next interval. If  $dec\%$  is larger than  $dec_{thresh}$ , the fetch size will decrease into half for the next interval.

## 4.4 Hardware Cost

The hardware needed to support fetch size adaptation is little. The additional hardware is listed below:

- a register to store interval length;
- two registers to store threshold values;
- two counters to store the number of VCLs with good/poor spatial locality;
- a comparator to determine whether two VCLs have equal tags.

## 5. RESULTS

The same simulation setup described in Section 3 is used to obtain the performance by fetch size adaptation. The access interval for fetch size prediction and change is 200,000. Both  $inc_{thresh}$  and  $dec_{thresh}$  are 0.7.

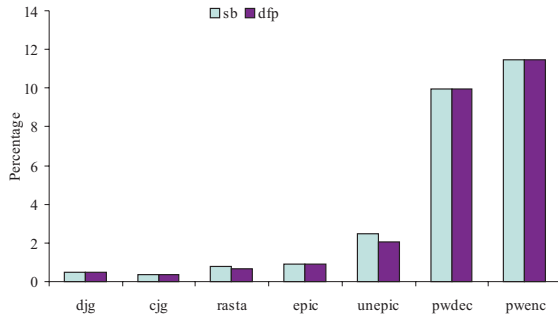


Figure 6: Miss rate: fetch size adaptation vs. SB.

Figure 6 compares the miss rate by fetch size adaptation and by SB. The miss rate by fetch size adaptation is equal to or slightly better than that by SB.

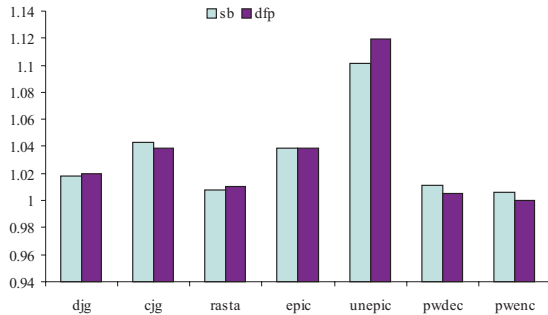


Figure 7: Speedup: fetch size adaptation vs. SB.

Figure 7 compares the speedup by fetch size adaptation and by SB. Similar to the trend of miss rate, the speedup by these techniques is also comparable. This demonstrates that fetch size adaptation is a viable alternative to SB in hiding memory latency.

In *pwdec* and *pwenc*, the difference in speedup between the SB and fetch size adaptation is much smaller than that between the SB and fetch size 128B shown in Figure 3(e) and

3(f). This proves that fetch size adaptation is effective in minimizing cache pollution.

## 6. CONCLUSION

In this paper, we have compared the effectiveness of the SB and fetch size adaptation for media benchmarks. Our experimental results show that an AFS cache with fetch size adaptation is comparable in performance to a cache with a SB. Fetch size adaptation often can choose the optimal fetch size. An AFS cache with fetch size adaptation has advantages over the SB due to the ease in implementation and verification, which makes it attractive for media processors.

## 7. REFERENCES

- [1] D. Burger and T. Austin. The simplescalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, 1997.
- [2] J. Fu, J. Patel, and B. Janssens. Stride directed prefetching in scalar processors. In *Int'l Symp. Microarchitecture*, pages 102–110, 1992.
- [3] Intel Corporation. *Intel StrongARM SA-1110 Microprocessor, Developer's Manual*, 2000.
- [4] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Int'l Symp. Computer Architecture*, pages 252–263, 1997.
- [5] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Int'l Symp. Computer Architecture*, pages 364–373, 1990.
- [6] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Int'l Symp. Microarchitecture*, pages 330–335, 1997.
- [7] MIPS Corporation. *MIPS R3000 hardware manual*, MIPS Corporation.
- [8] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Int'l Symp. Computer Architecture*, pages 24–33, 1994.
- [9] A. J. Smith. Cache memories. *Computing Surveys*, pages 473–530, 1982.
- [10] W. Tang, A. Veidenbaum, A. Nicolau, and R. Gupta. Adapting line size cache. Technical Report ICS-99-56, University of California, Irvine, 1999.
- [11] D. F. Zucker, M. J. Flynn, and R. B. Lee. A comparison of hardware prefetching techniques for media benchmarks. In *Int'l Conf. Multimedia Computing and Systems*, pages 236–44, 1996.