Prefetching in a Texture Cache Architecture

Homan Igehy
Computer Science Department

Matthew Eldridge
Department of Electrical Engineering

Kekoa Proudfoot
Department of Electrical Engineering

Stanford University

Abstract

Texture mapping has become so ubiquitous in real-time graphics hardware that many systems are able to perform filtered texturing without any penalty in fill rate. The computation rates available in hardware have been outpacing the memory access rates, and texture systems are becoming constrained by memory bandwidth and latency. Caching in conjunction with prefetching can be used to alleviate this problem.

In this paper, we introduce a prefetching texture cache architecture designed to take advantage of the access characteristics of texture mapping. The structures needed are relatively simple and are amenable to high clock rates. To quantify the robustness of our architecture, we identify a set of six scenes whose texture locality varies over nearly two orders of magnitude and a set of four memory systems with varying bandwidths and latencies. Through the use of a cycle-accurate simulation, we demonstrate that even in the presence of a high-latency memory system, our architecture can attain at least 97% of the performance of a zero-latency memory system.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture.

1 INTRODUCTION

Texture mapping has become ubiquitous in real-time graphics hardware over the past few years. The accelerators found in every segment of the market, from the consumer level to the graphics supercomputer level, have on-chip support for performing the costly operations associated with texture mapping. The use of texture mapping is so pervasive that many systems are built to perform the necessary operations without any penalty in fill rate.

Texture mapping is expensive both in computation and memory accesses. Continual improvement in semiconductor technology has made the computation relatively affordable, but memory accesses have remained troublesome. Several researchers have proposed and demonstrated texture cache architectures which can reduce texture memory bandwidth. Hakura and Gupta examine different organizations for on-chip cache architectures which are useful for exploiting locality of reference in texture filtering, texture magnification, and to a limited extent, repeated textures [5]. Cox, Bhandari, and Shantz extend this work to multi-level caching [3]. They demonstrate that on-chip caches in conjunction with large off-chip caches can be used to exploit all of the aforementioned forms of texture locality as well as inter-frame texture locality. Thus, memory bandwidth requirements can be dramatically reduced for scenes in which the working set of a frame fits into the off-chip cache.

A second troublesome point about texture memory access (which is not addressed by Hakura or Cox) is the high latencies of modern memory systems. In order to address this problem, several systems have been described that make use of large pipelines which prefetch the texel data [1, 7, 11]. Two of the systems [1, 7] do not use any explicit caching, although their memory systems are organized for the reference patterns of texture filtering, but one system [11] does employ prefetching as well as two levels of caching, one of which holds compressed textures. However, the algorithm which combines the prefetching with the caching is not described. Several other consumer-level architectures exist which undoubtedly utilize some form of prefetching, possibly with caching. Unfortunately, none of these algorithms are described in the literature.

In this paper, we introduce a texture architecture which combines prefetching and caching. Our architecture is designed to take advantage of the peculiar access characteristics of texture mapping. The structures needed for implementing the prefetching algorithm are relatively simple, thus making them amenable to the high clock rates associated with texture mapping. To quantify the robustness of our prefetching texture cache architecture, we identify a set of six scenes whose texture locality varies over nearly two orders of magnitude and a set of four memory systems with varying bandwidths and latencies. Through the use of a cycle-accurate simulation, we demonstrate that the texture prefetching architecture can successfully hide nearly all of the latency of the memory system over a wide range of configurations. The space overhead of this architecture is reasonable, and the resulting texture system is always able to attain at least 97% of the performance of a zero-latency memory system.

2 MIP MAPPING

Texture mapping, in its most basic form, is a process by which a 2D image is mapped onto a projected screen-space triangle under perspective. This operation amounts to a linear transformation in 2D homogeneous coordinates. The transformation is typically done as a backward mapping—for each pixel on the screen, the corresponding coordinate in the texture map is calculated. The backward mapped coordinate typically does not fall exactly onto a sample in the texture map, and the texture may be minified or magnified on the screen. Filtering is applied to minimize the effects of aliasing, and ideally, the filtering should be efficient and amenable to hardware acceleration.

{homan,eldridge,kekoa}@graphics.stanford.edu

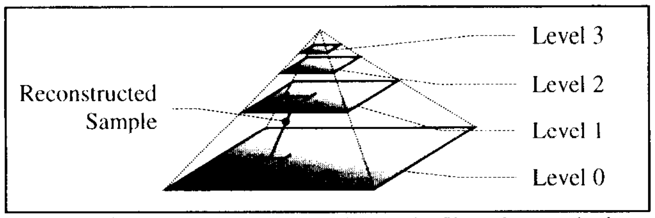


Figure 1: Mip Mapping. An image is filtered recursively into quarter-sized images. Trilinear interpolation reconstructs a sample by linearly interpolating between two adjacent levels of the mip map, each of which is sampled with bilinear filtering on the four closest texels in that level of the mip map.

Mip mapping [12] is the filtering technique most commonly implemented in graphics hardware. In mip mapping, an image pyramid is constructed from the base image which serves as the bottom of the pyramid. Each successive level of the pyramid is constructed by resampling the previous level of the pyramid by half in each dimension, as illustrated in Figure 1. For each screen-space fragment, the rasterization process computes a texture coordinate and an approximate texel-to-pixel ratio (also known as the level-of-detail value). This ratio is used to compute the two closest corresponding mip map levels, and a bilinear interpolation is performed on the four nearest texels in each of the two adjacent levels. These two values are then combined with linear interpolation based on the level-of-detail value, and the resulting trilinearly interpolated sample is passed to the rest of the graphics pipeline. If a fragment falls beyond either end of the mip map pyramid, the algorithm performs bilinear filtering on the one closest level of the mip map.

The popularity of mip mapping can be attributed to three characteristics. First, mip mapping reduces many aliasing artifacts. Although it is by no means an ideal filter, especially since it often blurs excessively, the results are quite acceptable for interactive applications. Second, the computational costs of mip mapping, though by no means cheap, are reasonable and fixed for each fragment. Finally, mip mapping is efficient with respect to memory. The additional space required for the pyramid representation is only one-third the space occupied by the original image. Furthermore, because the level-of-detail computation is designed to make one step in screen space correspond to approximately one step in the appropriate mip map level, the memory access pattern of mip mapping is very coherent.

3 CACHING AND PREFETCHING

For the past few decades, many aspects of silicon have been experiencing exponential growth. However, not all aspects have grown at the same rate. While memory density and logic density have seen tremendous growth, logic speed has experienced more moderate growth, and memory speed has experienced slight growth. These factors have made the cost of computation on a chip very cheap, but memory latency and bandwidth sometimes limit performance. Even with the advent of memory devices with high-speed interfaces [4], it is easy to build a texturing system that outpaces the memory it accesses. The problem of directly accessing DRAM in a texture system is aggravated by the fact that memory devices work best with transfers that do not match the access patterns of texture mapping: DRAM provides high bandwidth when moving large contiguous blocks of memory, but a fragment's texture accesses typically consist of several small non-contiguous memory references.

An obvious solution to this problem is caching. Many issues are resolved by integrating a small amount of high-speed, on-chip memory organized to match the access patterns of the texture system. According to our measurements (detailed in Section 5.1) as well as data found in other literature [3, 5], it is quite reasonable to expect miss rates on the order of 1.5% per access. Many texture systems are capable of providing the computation for a trilinearly mip mapped fragment on every clock cycle. Thus, because there are eight texture accesses per cycle, the per-fragment texel miss rate is 12%. Even if these misses could be serviced in a mere 8 cycles each, a calculation of the average memory access time shows that overall performance is cut in half. Clearly, this is not acceptable.

While caching can alleviate the memory bandwidth problem, it does not solve the memory latency problem. The latency problem with relation to texture caching is a special one. In current interactive graphics interfaces, texture accesses are read-only for large amounts of time, and address calculation for one texture access is never dependent on the result of another texture access. Thus, there are no inherent dependencies to limit the amount of latency that can be covered. This means that a prefetching architecture should be capable of handling arbitrary amounts of latency.

3.1 Traditional Prefetching

In the absence of caching, prefetching is very easy. When a fragment is ready to be textured, the memory requests for the eight texel accesses are sent to the memory system, and the fragment is queued onto a fragment FIFO. When the replies to the memory requests arrive, the fragment is taken off the FIFO, and the fragment is textured. The time a fragment spends in the FIFO is equal to the latency of the memory system, and if the FIFO is sized appropriately, fragments may be processed without ever stalling. For greater efficiency, part of the fragment FIFO can actually be a fragment processing pipeline [1, 7]. Note that this non-caching prefetching architecture assumes that memory replies arrive in the same order that memory requests are made, and that the memory system can provide the required bandwidth with small memory requests.

One straightforward way to combine caching with prefetching is to use the architecture found in traditional microprocessors that use explicit prefetch instructions. Such an architecture consists of a cache, a fully associative prefetch address buffer, and a memory request buffer. A fragment in such a system is processed as follows: first, the fragment's texel addresses are looked up in the cache tags, and the fragment is stored in the fragment FIFO. Misses are forwarded to a prefetch buffer that is made fully associative so that multiple misses to the same memory block can be combined. New misses are queued in the memory request buffer before being sent to the memory system. As data returns from the memory system, it is merged into the cache. When a fragment reaches the head of the fragment FIFO, the cache tags are checked again, and if all of the texels are found in the cache, the fragment can be filtered and textured. Otherwise, additional misses are generated, and the system stalls until the missing data returns from memory. Fortunately, the architecture works even in conjunction with an out-of-order memory system.

There are three problems with using the traditional microprocessor prefetch architecture for texture mapping. First, if the product of the memory request rate and the memory latency being covered is large compared to the size of the caches utilized, a prefetched block that is merged into the cache too early can cause conflict misses. Second, in order to support both reading and prefetching of texels at the full fragment rate, tag checks must be performed at twice the fragment rate, increasing the cost of the tag logic. Finally, as the product of the memory request rate and the memory latency increases, the size (and therefore the associativity) of the prefetch buffer must be increased proportionally.

3.2 A Texture Prefetching Architecture

While some of the problems with the traditional microprocessor prefetching architecture can be alleviated, we have designed a custom prefetching architecture that takes advantage of the special access characteristics of texture mapping. This architecture is illustrated in Figure 2. Three key features differentiate this architecture from the one described in Section 3.1. First, tag checks are separated in time from cache accesses, and tag checks are performed only once per texel access. Second, because the cache tags are only checked once and always describe the future contents of the cache, a fully associative prefetch buffer is not needed. And third, a reorder buffer is used to buffer memory requests that come back earlier than needed.

The architecture processes fragments as follows. As each fragment is generated, each of its texel addresses is looked up in the cache tags. If a tag check reveals a miss, the cache tags are updated with the fragment's texel address immediately and the

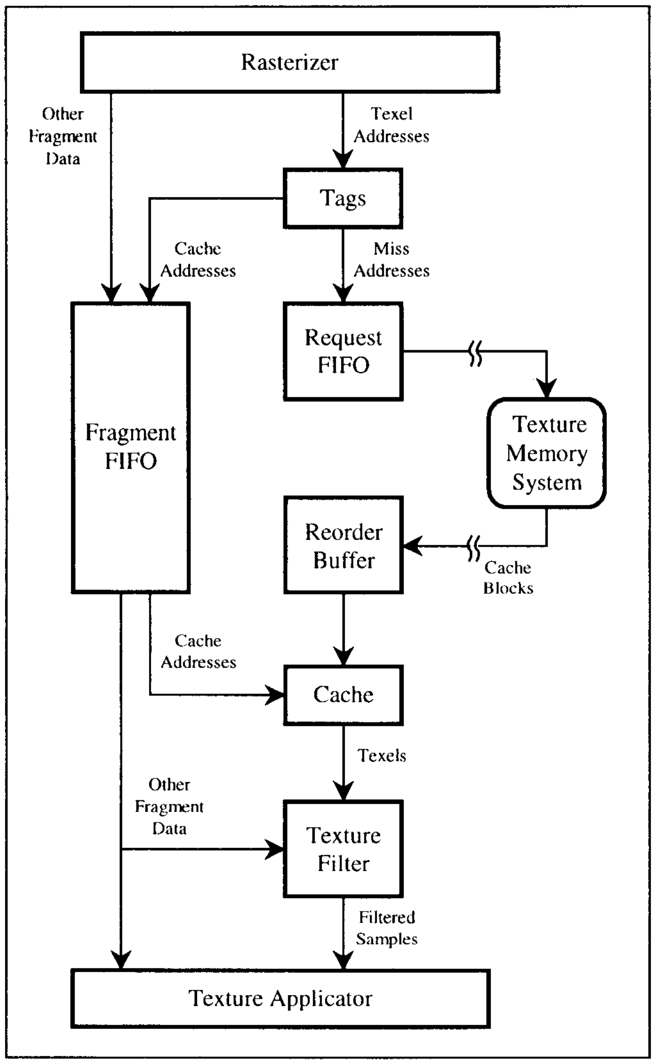


Figure 2: A Texture Prefetching Architecture.

address is forwarded to the memory request FIFO. The cache addresses associated with the fragment are forwarded to the fragment FIFO and are stored along with all the other data needed to process the fragment, including color, depth, and filtering information. As the request FIFO sends requests for missing cache blocks to the texture memory system, space is reserved in the reorder buffer to hold the returning memory blocks. This guarantee of space makes the architecture robust and deadlock-free in the presence of an out-of-order memory system. A FIFO can be used instead of the reorder buffer if responses from memory always return in the same order as requests sent to memory.

When a fragment reaches the head of the fragment FIFO, it can proceed only if all of its texels are present in the cache. Fragments that generated no misses can proceed immediately, but fragments that generated one or more misses must first wait for their corresponding cache blocks to return from memory into the reorder buffer. In order to guarantee that new cache blocks do not prematurely overwrite older cache blocks, new cache blocks are committed to the cache only when their corresponding fragment reaches the head of the fragment FIFO. Fragments that are removed from the head of the FIFO have their corresponding texels read from the cache and proceed onward to the rest of the texture pipeline.

Our simulated implementation can handle eight texel reads in parallel, consisting of two bilinear accesses to two adjacent mip map levels. To support these concurrent texel reads, we organize our cache tags and our cache memory as a pair of caches with four banks each. Adjacent levels of a mip map are stored in alternating caches to allow both mip map levels to be accessed simultaneously. Data is interleaved so that the four accesses of a bilinear interpolation occur in parallel across the four banks of the respective cache. Cache tags are also interleaved across four banks in a fashion that allows the tag checks for a bilinear access to occur without conflict. The details of this layout can be found in Figure 3 of Section 5.

In order to make our architecture amenable to hardware implementation, we impose two limitations. First, the number of misses that can be added to the request FIFO is limited to one miss per cache per cycle. Second, the number of cache blocks that can be committed to the cache from the reorder buffer is similarly limited to one block per cache per cycle. These commits match up to the requests—groups of misses that are added to the request FIFO together are committed to the cache together. This means that each fragment may generate up to four groups of misses. Because our implementation can only commit one of these groups per cycle, a fragment that has more than one group of misses will cause the system to stall one cycle for every group of misses beyond the first.

4 ROBUST SCENE ANALYSIS

When validating an architecture, it is important to use benchmarks that properly characterize the expected workload. Furthermore, when validating interactive graphics architectures, an architect should look beyond averages due to various characteristics of the human perceptual system. For example, if a graphics system provides 60 Hz rendering for the majority of the frames, but every once in a while drops to 15 Hz for a frame, the discontinuity is distracting, if not nauseating. In designing a system, the graphics architect must evaluate whether or not sub-optimal performance is acceptable under bad-case conditions. Accordingly, a robust set of scenes that cover a broad range of workloads, from good-case to bad-case, should be utilized to validate a graphics architecture.

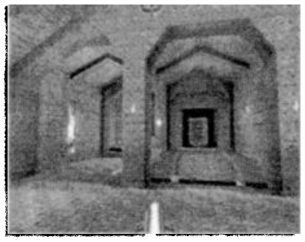
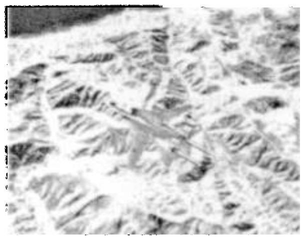
						
workload name	***quake***	***quake2x***	***flight***	***flight2x***	***qtvr***	***qtvr2x***
screen resolution	1280 x 1024	1280 x 1024	1280 x 1024	1280 x 1024	1280 x 1024	1280 x 1024
depth complexity	3.29	3.29	1.06	1.06	1.00	1.00
percent trilinear	30%	47%	62%	87%	0%	100%
unique texels/frag	0.033	0.092	0.706	1.55	0.569	2.83

Table 1: The Benchmark Scenes.

4.1 Texture Locality

The effectiveness of texture caching is strongly scene-dependent. For example, the size and distribution of primitives affect texture locality. Texture locality is also affected by what we call the scene's *unique texel to fragment ratio*. Every scene has a number of texels that are accessed at least once; these texels are called *unique texels*. Unless caches are big enough to exploit inter-frame locality (this requires several megabytes [3]), every unique texel must be fetched at least once by the cache, imposing a lower limit on the required memory bandwidth. If we divide this number by the number of fragments rendered for a scene, we can calculate the unique texel to fragment ratio. Note that this value is dependent on the screen resolution. A good-case scene will have a low ratio, and a bad-case scene will have a high ratio. Ideally, the number of texels fetched by the caching architecture per fragment will be close to the scene's unique texel to fragment ratio.

Three factors affect the unique texel to fragment ratio of a scene. First, when a texture is viewed under magnification, each texel gets mapped to multiple screen pixels, and the ratio decreases. Second, when a texture is repeated across a surface, the ratio also decreases. This temporal coherence can be exploited by a cache large enough to hold the repeated texture. Third, when a mip map texture is viewed under minification, the ratio becomes dependent on the relationship between texel area and pixel area. This relationship is characterized by the level-of-detail value of the mip mapping computation that aims to keep the footprint of a backward-mapped pixel equal to the size of a texel in a mip map level. Although this value is normally calculated automatically, the application programmer may bias it in either direction, thus modifying the scene's unique texel to fragment ratio.

A more surprising effect that occurs even without biasing is characterized by the fractional portion of the level-of-detail value. The level-of-detail value determines the two levels of the mip map from which samples are taken; the fractional portion is proportional to the distance from the lower, more detailed level. Given a texture mapped polygon that is parallel to the screen, a fractional portion close to zero implies a texel area to pixel area ratio of nearly one in the lower mip map level and a quarter in the upper mip map level, yielding a texel to fragment ratio near 1.25. Likewise, a fractional portion close to one implies a texel area to pixel area ratio of four in the lower mip map level and one in the upper mip map level, yielding a texel to fragment ratio near 5. The ratios are lower for polygons that are not parallel to the screen. Normally, we expect a wide variation in the texel to fragment ratio due to the fractional portion of the level-of-detail value However, most scenes exhibit worst-case behavior for short amounts of time, and a few scenes exhibit worst-case behavior for large amounts of time.

4.2 The Benchmark Scenes

In order to validate our texture caching architecture, we chose six real-world scenes that span a wide range of texture locality. These six scenes originated from three traces of OpenGL [10] applications captured by *glstrace*, a tool implemented on top of the OpenGL Stream Codec. In the future, we expect to see more texture for a given screen resolution; this will increase the unique texel to fragment ratio. To simulate this effect, each of the traces was captured twice, once with the textures at original size, and once with the textures at double resolution. Table 1 summarizes our six scenes, and high resolution images can be found in the Color Plate. Our workloads span nearly two orders of magnitude in the unique texel to fragment ratio (0.033 to 2.83). This is in contrast to the ratios in the scenes used by Hakura (0.2 to 1.1) [5] and the animations used by Cox (0.1 to 0.3) [3]. These workloads result from the fact that applications programmers choose the way they use texture according to the needs of the application and the constraints of the target systems. We now give a brief summary of each scene and highlight the points relevant to texture caching:

▪ *quake*	This is a frame from the OpenGL port of the video game Quake. This application is essentially an architectural walkthrough with visibility culling. Color mapping is performed on all surfaces which are, for the most part, large polygons that make use of repeated texture. A second texturing pass blends low-resolution light maps with the base textures to provide realistic lighting effects. Approximately 40% of the base textures are magnified, and 100% of the light maps are magnified.
▪ *quake2x*	In order to account for increasing texture resolutions needed for larger screen resolutions (Quake's content was geared towards smaller screens), the texture maps in *quake* were zoomed by a factor of two to create *quake2x*. This results in a scene which magnifies only the light maps.
▪ *flight*	This scene from an SGI flight simulator demo shows a jet flying above a textured terrain map. The triangle size distribution centers around moderately sized triangles, and most textures are used only once. A significant portion of the texture (38%) is magnified.

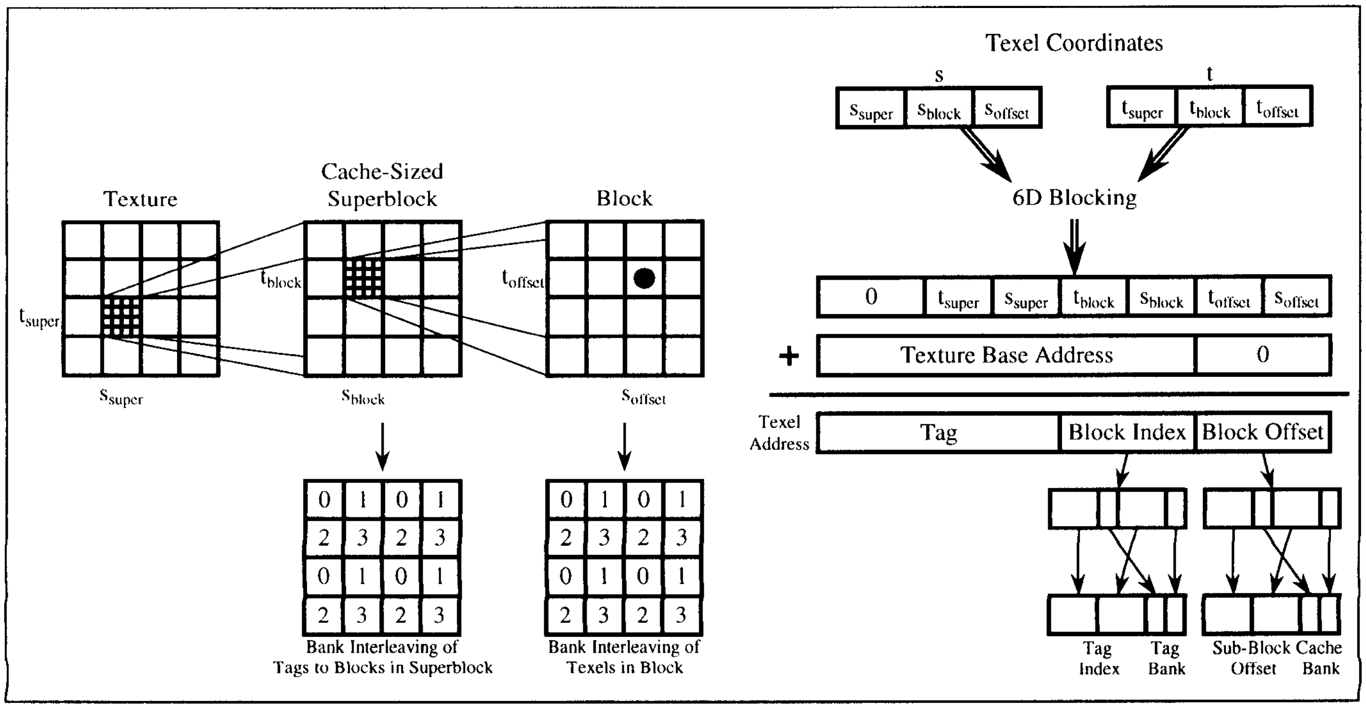


Figure 3: Texture Data Organization. In our architecture, textures are stored using a 6D blocking pattern. Each mip map level is divided into cache-sized superblocks, with each superblock further divided into blocks. Each block is a rectangular, linearly-addressable region of the original mip map level. Each of eight texel addresses is computed by adding an offset (formed by permuting the texel's coordinates) to a corresponding texture base address. The eight resulting texel addresses, four from each of two adjacent mip map levels, are then directed to two caches, each of which has four banks and services alternating levels of the mip map. Within each superblock, tags are interleaved on a block basis, causing all 2x2 texel accesses to fall onto one, two, or four adjacent blocks, with each block's tag stored in a separate bank of the tag memory. This interleaving is accomplished by permuting the bits of the block index, yielding a tag bank and a tag index for each texel address. Similarly, texels are interleaved within each block, causing all 2x2 texel accesses to fall into separate banks of the cache memory even if the texels of the 2x2 access do not all fall into the same block. A permutation of the block offset results in a cache bank and a sub-block offset for every texel address; used in conjunction with the block index, these values locate each texel in the cache memory. Note that both of these permutations extract the least significant bit of the corresponding *s* and *t* fields to determine the tag or cache bank.

- *flight2x* As texture systems become more capable of handling larger amounts of texture, applications will use larger textures to avoid the blurring artifact of filtered magnification. In *flight2x*, the textures of *flight* were zoomed by a factor of two. This results in a scene which only magnifies 13% of the texture.
- *qtvr* This scene comes from an OpenGL-based QuickTime VR [2] viewer looking at a panorama from Mars. This huge panorama, which measures 8K by 1K, is mapped onto a polygonal approximation of a cylinder made of tall, skinny triangles. Even though all of the texture is magnified, the lack of repeated texture keeps the number of unique texels per fragment high.
- *qtvr2x* The texture of *qtvr* was scaled up to 16K by 2K. This increases the number of unique texels accesses by the scene since all the texture is minified. Furthermore, the fractional portion of the level-of-detail value is always high in *qtvr2x* because the panorama is viewed more or less head-on at just the wrong zoom value. Note that these same effects would occur if *qtvr* was run at quarter-sized screen resolution, and that *qtvr2x* is by no means a hand-tailored pathological case. In fact, it was while gathering trace data on *qtvr* that we first observed the texture locality effects of a level-of-detail fraction close to one. This scene is representative of a bad-case frame in a real-world application.

5 MEMORY ORGANIZATION

In designing our prefetching cache architecture, careful attention was paid to choosing the proper parameters for the cache and the memory system. To narrow our search space, we leveraged Hakura's findings on blocking [5]. First, Hakura demonstrates the importance of placing texture into tiles according to cache block size. This addressing scheme is referred to as 4D blocking. Furthermore, rasterization should also occur in a 2D blocked fashion rather than in scan line order. And finally, tiles should be organized in a 2D blocked fashion according to the cache size in order to minimize conflict misses. This is called 6D blocking. In accordance to with these guidelines, we employ 6D blocking for texture maps according to the cache block size and the cache size, and we rasterize triangles in 8 pixel by 8 pixel blocks. The layout of texture data is illustrated in Figure 3. Figure 3 also illustrates how texture data is banked in both the cache tags as well as the cache memory in order to allow conflict-free access for bilinear interpolation. Note that for the purposes of this study, all texture data is stored as 32-bit RGBA values.

5.1 Cache Efficiency

Since we have decided to provide a separate cache for each of the bilinear accesses which needs to occur during every trilinear texture access, three cache parameters need to be chosen. The first choice is the cache block size. A small block size increases miss rates, but keeps bandwidth requirements low. A large block size can decrease miss rates, but bandwidth requirements and latency can skyrocket. An additional factor that needs consideration is that modern DRAM devices require large transfer sizes to sustain bandwidth. Hakura found that 16 texel tiles (64 bytes) work well, and most next-generation DRAM chips can achieve peak efficiency at such transfer sizes [4].

Given a 16 texel block size, we are left with choices for cache associativity and cache size. Figure 4 shows the miss rates for our six test scenes. We see that increasing associativity does not decrease the miss rate significantly. Intuitively, this make sense since having a separate cache for alternate levels of a mip map minimizes conflict misses. Thus, a direct-mapped cache is quite acceptable if we use 6D blocking when alternate levels of the mip map are cached independently. According to Hakura, if a unified cache is used for trilinear accesses (and thus the bilinear accesses do not occur simultaneously), a 2-way set associative cache is appropriate. In the more general case of multi-texturing, m independent n-way set associative caches are well suited towards providing texture accesses at the rate of m bilinear accesses per cycle to $m*n$ textures (in this scheme, trilinear accesses count as two accesses to two textures). Since we are limiting our study to a single trilinear access per cycle, two independent direct-mapped caches are appropriate.

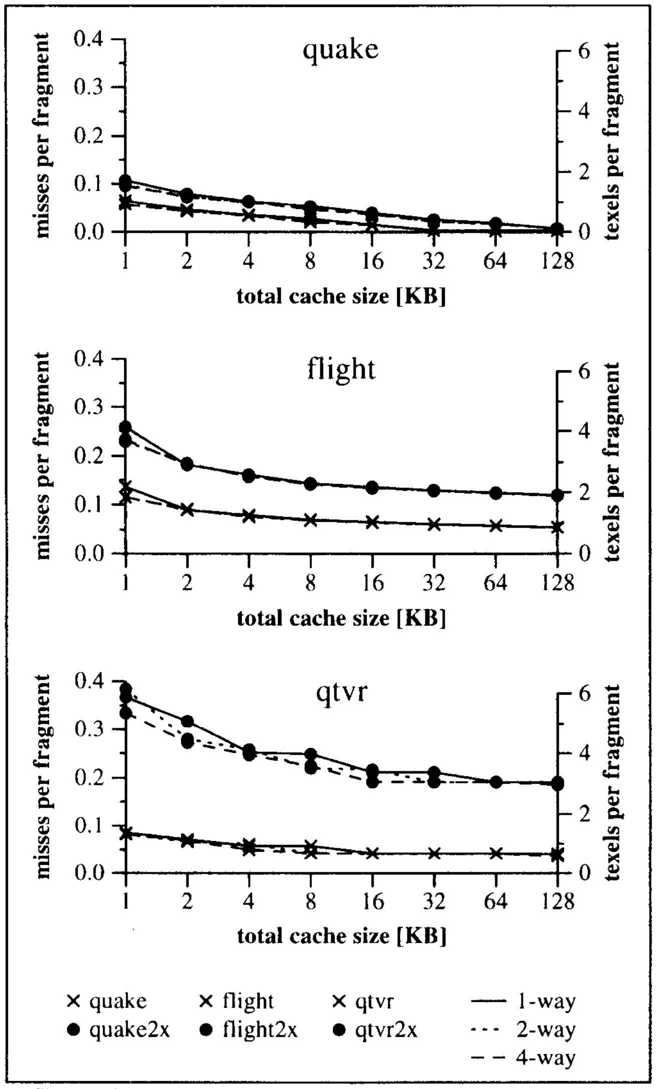


Figure 4: Cache Efficiency. Block size was set to 4 by 4 texels, and the six workloads were sent through the cache simulator with various cache sizes and cache associativities. Results are reported in terms of cache block misses per fragment rather than in terms of misses per access since most texturing architectures have clock cycles based on fragments. The cache block miss rate corresponds to a memory bandwidth requirement which can be expressed in terms of texels fetched per fragment.

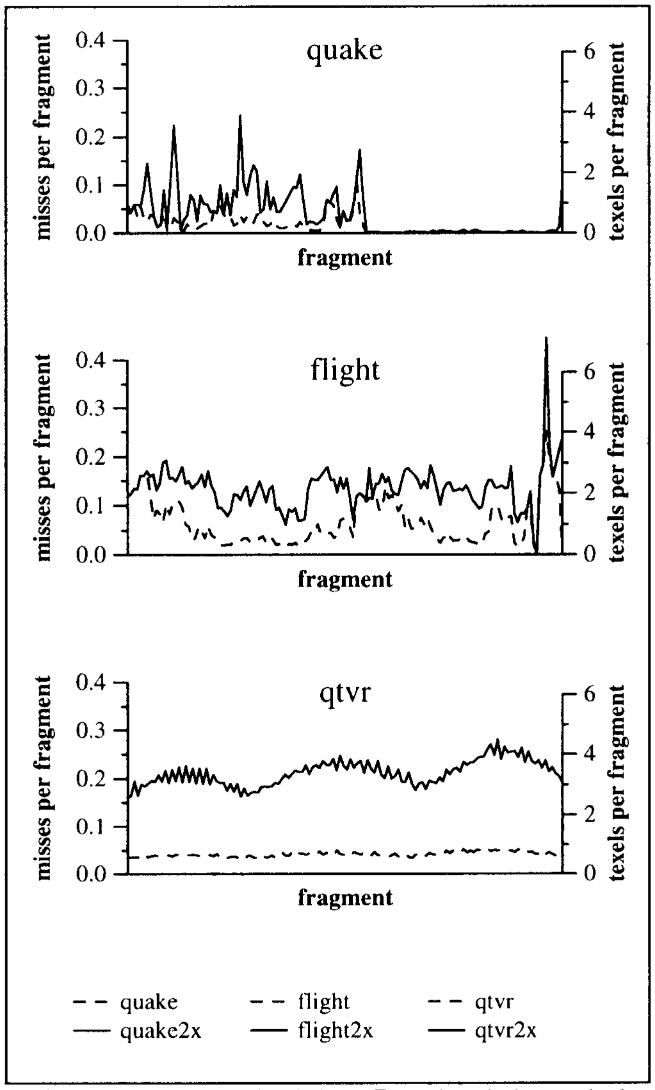


Figure 5: Bandwidth Variation. Even though the required memory bandwidth can be low on average, this value can vary widely over time. The graphs above show this variation with a pair of direct-mapped 8 KB caches. Each data point is a windowed average over 30,000 fragments in *quake* and 10,000 fragments in *flight* and *qtvr*. The variance in the required bandwidth is quite extreme in the cases of *quake* and *quake2x* as the application transitions from applying color maps to applying light maps.

Figure 4 also illustrates the effects of modifying the total cache size on the miss rates of the various scenes. We see that for scenes in which texture locality is not dependent on repeated textures (*flight, flight2x, qtvr, qtvr2x*), the miss rate curves flatten somewhere between a total cache size of 4 KB and 16 KB. This cache size represents the working set for filtering locality when rasteriztion is done in 8 by 8 blocks. On scenes that contain repeated texture (such as *quake* and *quake2x*) the miss rates are lower, but the miss rate curves flatten later (at 32 KB and 128 KB, respectively). These points correspond to the working set sizes of the repeated textures in each scene. The miss rate realized once any of the curves flattens corresponds closely to the unique texel to fragment ratio of the respective scene.

We chose to use a 16 KB cache (composed of two direct-mapped 8 KB caches) for our study. According to our workloads, this size is large enough to exploit nearly all of the coherence found in scenes which demonstrate poor locality (such as *flight2x* and *qtvr2x*), and even though a larger size could help in scenes with repeated textures (such as *quake* and *quake2x*), these scenes already perform very well. This cache size is in consensus with the cache sizes proposed in other texture cache analyses [3, 5]. Though we stress that different choices can also be reasonable, for the rest of the paper, we assume a cache architecture with two direct-mapped 8 KB caches (interleaved by mip map level) with 64-byte blocks.

5.2 Bandwidth Requirements

In formulating bandwidth requirements, we can relate the number of texels of memory bandwidth required per fragment to the cache miss rate by the cache block size. These equivalent measures are shown as left- and right-axes in Figure 4. One key point of Table 1 and Figure 4 is that even though caching can work well sometimes, there are cases when the bandwidth requirements are extremely high. In the case of *qtvr2x*, nearly 3 texels have to be fetched for each fragment no matter what size on-chip cache is utilized. This is quite high considering that eight texels are required to texture a trilinearly mip mapped fragment. However, this should not be seen as an argument for not having a cache: the cache still provides a way of matching the access patterns of mip mapping with the large block requests required for achieving high memory bandwidth. If a system wants to provide high performance robustly over a wide variety of scenes, it needs to provide high memory bandwidth even with the use of caching. If a system's target applications have high texture locality, or if cost is a primary concern, a memory system with lower memory bandwidth can be employed.

Figure 4 can also be a bit misleading because the average bandwidth requirement does not tell the whole story. From the data, one could falsely infer that a memory system which provides enough bandwidth to supply 1 texel per fragment will perform perfectly on *quake2x* since, according to the graph, only 0.63 texels per fragment are required given the 16 KB cache size. Figure 5 illustrates why this is not the case. The average cache miss rate does not properly encapsulate temporal variations in bandwidth requirements. Even though the average bandwidth requirement is 0.63 texels per fragment over the whole frame, large amounts of time exist when the system needs double that bandwidth, and large amounts of time exist when the system does not need most of that bandwidth (i.e., when light maps are drawn). Because of the large separation in time between these two phases, a system cannot borrow from one to provide for the other, and thus the overall performance will decrease.

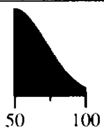
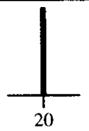
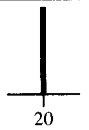
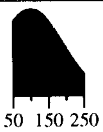
	agp	rdram	rdram2x	numa
period	16	8	4	4
latency	 50 100	 20	 20	 50 150 250

Table 2: Memory Models. The values reported here are in terms of a fragment clock cycle of 200 MHz, which corresponds to 5 nsec. The memory period determines the rate at which 64 byte blocks of memory can be provided. Thus, bandwidths of 1, 2, 4, and 4 texels per fragment are provided on *agp*, *rdram*, *rdram2x*, and *numa*, respectively.

5.3 Memory Models

In order to validate our texture prefetching architecture more precisely, we now explore the bandwidths and latencies provided by memory systems. For our study, we examine an architecture which can sustain the texturing performance projected for the near future. Current high-end architectures such as the SGI InfiniteReality [9] provide approximately 200 million trilinear fragments per second from a single board. Low-end professional-level architectures provide approximately 30 million trilinear fragments per second [7], as do many consumer-level graphics accelerators. Given these rates, we decided to set our nominal fragment clock rate at 200 MHz, meaning that under optimal memory conditions, the architecture provides a trilinearly sampled fragment every 5 nanoseconds. Based on this fragment clock rate, we decided to simulate four memory models, summarized by bandwidth and latency histogram in Table 2.

- *agp* — This models a system in which the texture cache requests blocks from system memory over Intel's Advanced Graphics Port [6]. The AGP 4X standard can provide a sustained bandwidth of 800 MB/sec. Because system memory is shared with the host computer, we estimate that the latency of *agp* varies between 250 nsec and 500 nsec.
- *rdram* — Direct RDRAM from Rambus [4] will serve as our baseline dedicated texture memory. These devices provide extremely high bandwidth (a sustainable 1.6 GB/sec) with reasonable latency (90 nsec) at high densities for commodity prices. We estimate that on-chip buffering logic adds 10 nsec of latency to this memory.
- *rdram2x* — In order to sustain the high and variable bandwidth requirements of scenes such of *flight2x* and *qtvr2x*, a texture architecture may choose to utilize 2 RDRAM parts for double the bandwidth of *rdram* at the same latency.
- *numa* — Although not based on any existing specification, we use the *numa* memory model to examine the feasibility of our prefetching architecture in novel and exotic texture memory architectures. The bandwidth of this memory model is the same as the bandwidth of *rdram2x*, but the latency of such a system is extremely high and highly variable. It can range anywhere between 250 nsec and 1.25 usec. This latency is in the range of what can be expected if texture is distributed across the shared memory of a NUMA multiprocessor [8].

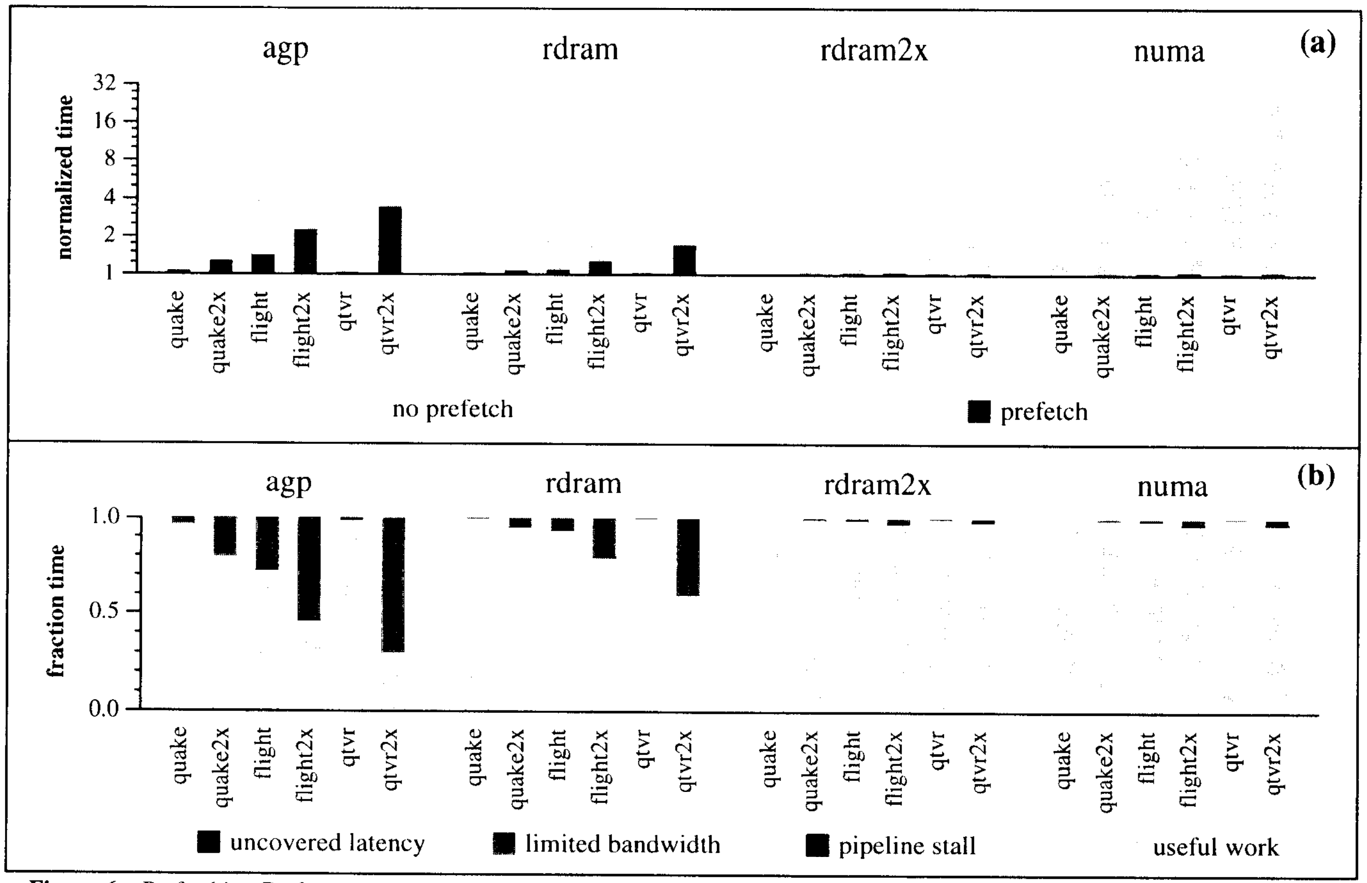


Figure 6: Prefetching Performance. In (a), we compare our prefetching architecture and one in which no prefetching takes place against an ideal architecture (one in which a fragment is generated on every clock cycle) on a logarithmic scale. On many configurations, the prefetching architecture is able to achieve near-ideal performance (as indicated by the near-total absence of a dark gray bar). Configurations which do not achieve near-ideal performance are bandwidth-limited, as illustrated in (b). This graph characterizes the architecture's execution time by useful work, pipeline stalls, limited memory bandwidth, and uncovered latency across the four memory models and the six scenes. For all of the cases in which near-ideal performance was not attained, memory bandwidth is by far the limiting factor. Thus, the architecture is able to hide nearly all of the latency of the memory system with little overhead.

6 PERFORMANCE ANALYSIS

A cycle-accurate simulator was written to validate the robustness of the prefetching texture cache architecture proposed in this paper. We analyze the architecture by running each scene with each memory model. First, the architecture is compared against an ideal architecture and an architecture with no prefetching. We then account for all of our execution time beyond the ideal execution time.

Figure 6a presents the execution time for each of the scenes with each of the memory models on both our architecture and an architecture with no prefetching. Performance is normalized to the ideal execution time of 1 cycle per fragment. In all cases, our architecture performs much better than an architecture lacking prefetching. However, we do not achieve an ideal 1 cycle per fragment across many of the scenes when running the *agp* and *rdram* memory models.

In order to account for lost cycles, we enumerate four components of our architecture's execution time:

1. A cycle is required to move each fragment through the texture pipeline.
2. If either cache has more than one miss for any fragment, the pipeline must stall.
3. The pipeline may stall due to insufficient texture memory bandwidth.
4. Cycles may be lost to uncovered latency in the prefetching architecture.

Each of these components can be calculated as follows. The number of cycles spent moving fragments through the pipeline is simply the number of fragments in the scene. The number of pipeline stalls attributed to multiple misses per fragment can be measured by counting the number of misses per cache per fragment beyond the first miss. Stalls occur infrequently, and our experiments show the performance lost to such pipeline stalls is typically less than 1%. Performance lost to insufficient memory bandwidth is determined by the execution time of the trace with the memory latency set to zero. Finally, when the scene is simulated with our memory latency model, any additional cycles not attributed to the first three categories are counted as uncovered latency in our architecture. Experimental results show that most of the latency of the memory system is indeed covered by our architecture, with at least 97% utilization of hardware resources using nominal sizes for the fragment FIFO, the memory request FIFO, and the reorder buffer. Most of the performance difference from an ideal system is caused by insufficient memory bandwidth. The breakdowns of the execution times for our configurations are presented in Figure 6b.

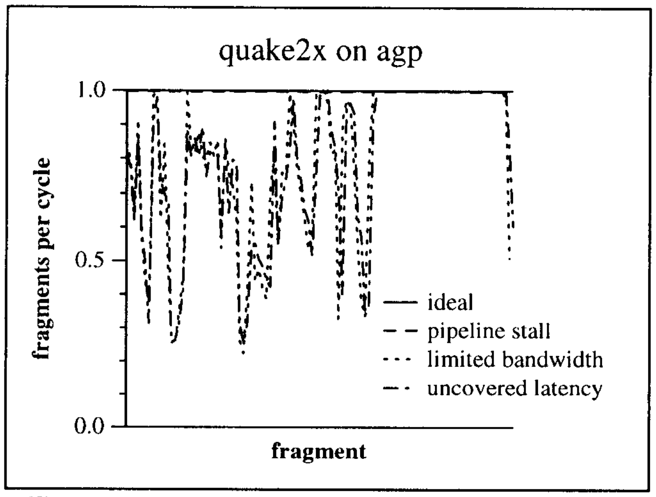


Figure 7: Time-Varying Execution Time Characterization. As predicted in Section 5.2, the performance of a workload can vary greatly over time if not enough memory bandwidth is provided. The graph above characterizes the execution time of the *quake2x* workload on the *agp* memory system. Even though the 1 texel per fragment bandwidth of *agp* by far exceeds *quake2x*'s average requirement of 0.63 texels per fragment, performance suffers due to the time-varying bandwidth requirements of *quake2x*.

6.1 Intra-Frame Variability

A typical scene provides both an overall memory bandwidth demand over the course of the frame (several milliseconds) as well as localized memory bandwidth demands over several microseconds, as illustrated in Figure 5. Figure 7 shows how this translates into lost performance. The performance of the *quake2x* scene on the *agp* memory system is very different in the first and second half of the frame due to switching between color map textures and light maps. As predicted in Section 5.2, the fragment rate while drawing the color texture is limited by memory bandwidth while the pipeline runs at full speed while drawing the light maps. This does indeed cause an overall performance penalty even though the 1 texel per fragment bandwidth of *agp* far exceeds the average texel per fragment bandwidth requirement of *quake2x*. Figure 7 also illustrates that the performance of our architecture closely tracks the performance of a zero-latency memory system over time.

6.2 Buffer Sizes

The data in Figure 6 and Figure 7 was derived with a specific set of buffer sizes for each memory model. These sizes are presented in Table 3, and in all cases the buffers are reasonable in size when compared to the 16 KB of cache employed.

We determined the sizes of the three buffers—the fragment FIFO, the request FIFO, and the reorder buffer—by inspection and then validated them by experimentation. The fragment FIFO primarily masks the latency of the memory system. If the system is not to stall on a cache miss, it must be able to continually service new fragments while previous fragments are waiting for texture cache misses to be filled. Thus, the fragment FIFO depth should at least match the latency of the memory system. The fragment FIFO also provides elasticity between the burstiness of texture misses and the constant rate at which the memory system can service misses, and therefore should be larger than just the memory system latency. The memory request FIFO also provides elasticity between the potentially bursty stream of miss addresses generated by the fragments and the fixed rate at which the memory system can consume them. The size of this buffer was determined primarily by experimentation. Finally, in order to provide a robust, deadlock-free solution which can handle out-of-order memory responses, our architecture requires that a reorder buffer slot be reserved when a memory request is made. Since a memory response will not be received and applied to the cache at least until after the memory latency has passed, the reorder buffer should be sized to be at least the ratio of the memory access time (latency) to the memory cycle time (period) entries deep.

The above guidelines were used to determine the approximate buffer sizes for each memory model, and then the choices were adjusted by measuring the performance of the system. We fine-tuned the buffer sizes by holding two of the buffer sizes constant and varying the third. If the buffer is sized appropriately, the performance of the overall system should decrease significantly when the buffer is made much smaller, and performance should increase very slowly if the buffer is made larger. The data for this process with the *flight2x* workload is shown in Figure 8. This process provided useful information in the cases of the *rdram* and *rdram2x* memory systems. The fragment FIFOs were originally sized to be 32 entries deep. However, simulation revealed that this did not provide enough elasticity, and increasing the FIFO depth to 64 entries improved performance by several percent. Similarly, simulation revealed that performance increased slightly when the reorder buffer size was increased to 8 slots and 16 slots for *rdram* and *rdram2x*, respectively.

7 FUTURE WORK

In formulating a model for measuring the performance of our prefetching texture cache architecture, we assumed that the entire scene is rasterized by a renderer which is able to provide a fragment to the texture subsystem on every clock cycle. In a real system, this may not be the case. When triangles are smaller, caching does not work as well; but smaller triangles may also imply a lower fill rate (i.e., the scene is geometry limited), thus alleviating some of the penalty associated with the caching. A more detailed analysis of bandwidth requirements in a rasterization architecture should take this effect into account.

Another issue not addressed by our paper is the effect of parallel rasterization. Rasterization work may distributed amongst multiple processors in some fashion (e.g., pixel interleaved, triangle interleaved, tiled) that reduces the effectiveness of caching. Again, this affects the bandwidth requirements of a prefetching texture cache architecture. An additional possibility to explore

	Fragment FIFO Size	Request FIFO Size	Reorder Buffer Size
agp	128 slot 2.0 KB	8 slot 64 byte	8 slot 576 byte
rdram	64 slot 1.0 KB	8 slot 64 byte	8 slot 576 byte
rdram2x	64 slot 1.0 KB	16 slot 128 byte	16 slot 1.1 KB
numa	256 slot 4.0 KB	16 slot 128 byte	64 slot 4.5 KB

Table 3: Buffer Sizes. The numbers in each entry represent the sizes of the various buffers used in the various memory systems. Fragment FIFO entries are 16 bytes, memory request FIFO entries are 8 bytes, and reorder buffer entries are 72 bytes.

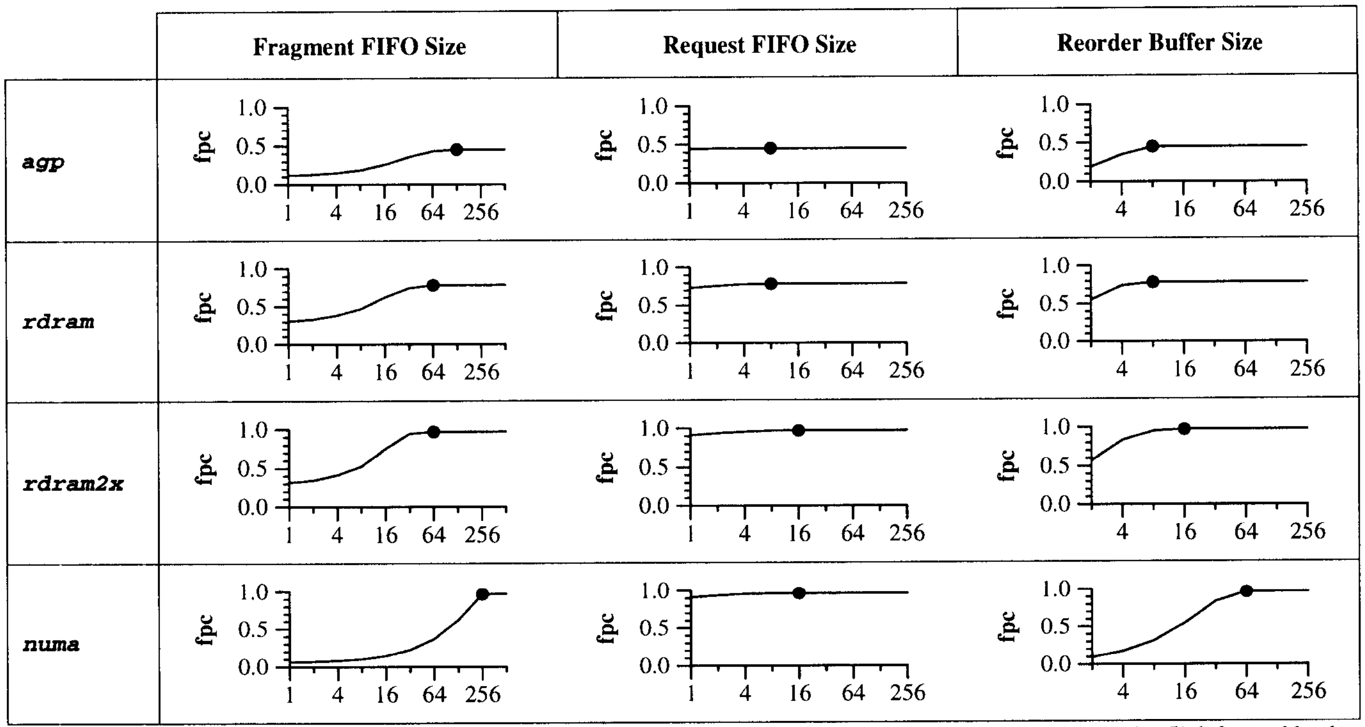


Figure 8: The Effects of Varying Buffer Sizes. The graphs above show the effects of varying buffer sizes on the *flight2x* workload across the different memory models. For each graph, one buffer size is varied while the other two are held fixed (at the values specified in Table 3). The results are reported in fragments per cycle (fpc), and the dot on each graph represents the final values used for the architecture on each memory model. The memory models whose fragments per cycle values do not approach 1.0 are bandwidth-limited.

with parallel rasterization is shared texture memory and its effect on latency.

8 CONCLUSION

In this paper, we have presented and analyzed a prefetching texture cache architecture. We designed the architecture to take advantage of the distinct memory access patterns of mip mapping. In order to validate the architecture, we presented a measurement of texture locality and identified six workloads that span over two orders of magnitude in texture locality. By examining the performance of these workloads across four different memory models, we demonstrated the architecture's ability to hide the memory latency with a 97% utilization of the available hardware resources.

Acknowledgements

We would like to thank Pat Hanrahan, Gordon Stoll, Milton Chen, and the rest of the Stanford Graphics Lab for their reviews of this paper and their insights about this work. We thank Phil Lacroute for providing *glstrace*. Financial support was provided by Silicon Graphics, Intel, and DARPA contract DABT63-95-C-0085-P00006.

References

[1] B. Anderson, R. MacAulay, A. Stewart, and T. Whitted. Accommodating Memory Latency In A Low-Cost Rasterizer. *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 97-102, 1997.

[2] S. E. Chen. QuickTime VR: An Image-Based Approach to Virtual Environment Navigation. *Computer Graphics* (SIGGRAPH 95 Proceedings), volume 29, pages 29-38, 1995.

[3] M. Cox, N. Bhandari, and M. Shantz. Multi-Level Texture Caching for 3D Graphics Hardware. *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[4] R. Crisp. Direct Rambus Technology: The New Main Memory Standard. *IEEE Micro*, pages 18-28, Nov/Dec. 1997.

[5] Z. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[6] Intel Corporation. *Accelerated Graphics Port Interface Specification*, revision 2.0, Intel Corporation, 1998.

[7] M. Kilgard. Realizing OpenGL: Two Implementations of One Architecture. *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 45-56, 1997.

[8] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *Proceedings of the 24th Annual Symposium on Computer Architecture*, 1997.

[9] J. Montrym, D. Baum, D. Dignam, and C. Migdal. InfiniteReality: A Real-Time Graphics System. *Computer Graphics* (SIGGRAPH 97 Proceedings), volume 31, pages 293-302, 1997.

[10] J. Neider, T. Davis and M. Woo. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[11] J. Torborg and J. Kajiya. Talisman: Commodity Real-Time 3D Graphics for the PC. *Computer Graphics* (SIGGRAPH 96 Proceedings), volume 30, pages 57-68, 1996.

[12] L. Williams. Pyramidal Parametrics. *Computer Graphics* (SIGGRAPH 83 Proceedings), volume 17, pages 1-11, 1983.