Parallel Texture Caching

Homan Igehy Matthew Eldridge Pat Hanrahan

Computer Science and Electrical Engineering Departments

Stanford University

Abstract

The creation of high-quality images requires new functionality and higher performance in real-time graphics architectures. In terms of functionality, texture mapping has become an integral component of graphics systems, and in terms of performance, parallel techniques are used at all stages of the graphics pipeline. In rasterization, texture caching has become prevalent for reducing texture bandwidth requirements. However, parallel rasterization architectures divide work across multiple functional units, thus potentially decreasing the locality of texture references. For such architectures to scale well, it is necessary to develop efficient parallel texture caching subsystems.

We quantify the effects of parallel rasterization on texture locality for a number of rasterization architectures, representing both current commercial products and proposed future architectures. A cycle-accurate simulation of the rasterization system demonstrates the parallel speedup obtained by these systems and quantifies inefficiencies due to redundant work, inherent parallel load imbalance, insufficient memory bandwidth, and resource contention. We find that parallel texture caching works well, and is general enough to work with a wide variety of rasterization architectures.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – color, shading, shadowing, and texture.

1 INTRODUCTION

The problem of computer graphics is very challenging computationally, particularly in the face of real-time constraints and realistic image synthesis. On today's computers, real-time interactive images can be created at several dozen frames per second, but the most realistic images take several hours to compute, a computational gap that spans six orders of magnitude. In the quest for ever-higher performance, one basic direction of computer architecture has been *parallelism*: the ability to use many lower performance components to build a higher performance system. Parallelism has been exploited in a variety of ways to accelerate sections of the graphics pipeline, including geometry processing, rasterization, and the host interface.

In recent years, texture mapping has become an integral part of real-time graphics hardware. Simultaneously, the burden has shifted from providing adequate computation for the available memory bandwidth to providing adequate bandwidth for abundant computation capability. Thus, it has become increasingly critical to be thrifty with memory bandwidth. Texture caching is one effective technique that minimizes bandwidth demands by leveraging locality of reference [3, 5, 6]. Unfortunately, however, parallel rasterization algorithms diminish locality because work is divided into smaller tasks. Consequently, the efficiency of a parallel graphics architecture, as measured against a serial rasterizer, is in part determined by how well texture caching extends to parallel rasterization.

An architecture that employs parallel rasterization can be deployed in many ways. First, the architecture can divide work among the rasterization units according to either an object-space partitioning or an image-space partitioning. With regards to texture, it is most straightforward to provide each rasterization unit with a *dedicated* texture memory that holds a replicated copy of the scene's texture data. Another interesting option is to provide support for a *shared* texture memory system, obviating the need for texture replication in each memory and allowing the bandwidth load of each rasterizer to be distributed over the entire memory system.

In this paper, we evaluate parallel texture caching architectures suitable for both dedicated and shared texture memories. We first present a specific set of rasterization algorithms and study the effects of parallel rasterization on texture locality and working set size. Using this information to configure the caches for each rasterization scheme, we then study the effects of load imbalance on bandwidth requirements. We find that parallel texture caching works well from low to high levels of parallelism, resulting in high parallel efficiency. Moreover, it is general enough to work with a wide variety of rasterization architectures.

2 PREVIOUS WORK

2.1 Parallel Texture Mapping

Until recently, it had been difficult to provide the amount of computation required for texturing at high fragment rates within a single chip, so solutions were naturally parallel. Although texturing was used in the earlier dedicated flight simulators, one of the first real-time texture mapping workstations was the SGI RealityEngine [1]. This system parallelizes rasterization by interleaving vertical stripes of the framebuffer across 5, 10, or 20 fragment generator units. Each fragment generator is coupled with an independent texture memory. Because texture access patterns are independent of framebuffer interleaving patterns, any

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

fragment generator needs to be able to access any texture data, so each fragment generator replicates the entire texture state. The successor to the RealityEngine, the InfiniteReality [10], uses 1, 2, or 4 higher performance fragment generators, and thus replicates texture memory up to 4 times, instead of up to 20 times. The texture subsystems of these architectures made minimal use of texture locality. The stripe interleaving of rasterization used in aforementioned high-end machines has recently appeared as scan-line interleaving in consumer-level graphics accelerators such as the Voodoo2 SLI from 3Dfx. As with the SGI systems, texture memory is replicated across all the rasterizers.

One other class of scalable graphics architectures in which texture mapping has been implemented is the image composition architecture, as exemplified by PixelFlow [9]. In such a system, multiple independent pipelines each generate a subset of the pixels for a scene, and these pixels are merged for display through an image composition network. Because each of the independent pipelines has its own texturing unit and texture memory, the amount of texture memory available to an application could be scaled. However, the problem is not straightforward since there must be explicit software control to only send primitives to the particular pipeline holding its texture data. Such a scheme is greatly at odds with dynamically load balancing the amount of work in each pipeline, particularly given the irregularities of human interaction. If shading (and thus, texturing) is deferred until after the pixel merge is completed, the problem of dynamically load balancing shading work according to texture accesses is equally, if not more, challenging. There have been no published works to date that address these challenges. As with other parallel hardware architectures, the PixelFlow replicates texture memory [8]; furthermore, the locality of texture access is not exploited.

2.2 Texture Locality

Several papers have analyzed cache locality in texture mapping. There is also evidence that all current consumer 3D graphics accelerators are using texture caches to exploit texture locality, but there have been no published studies on their effectiveness.

Hakura and Gupta [5] analyzed the effects of varying cache parameters, texture memory layout, and rasterization order on the effectiveness of caches for a single rasterizer. The small on-chip cache sizes studied in that paper are able to take advantage of redundant accesses between adjacent pixels due to both filtering and repeated textures. One of the most important ideas presented was the importance of organizing texture data in a tiled fashion. Because texture data can appear at any angle with respect to the screen, it is important to store 2D texture data in a 4D tiled order so that each block in the cache holds a square or almost-square region of texture. Furthermore, it is important to use an additional level of tiling based on the number of cache sets to reduce conflict misses, thus leading to 6D texture tiling. This is explained in Section 4.3.1. Another important conclusion was that rasterization should also be done in a 4D tiled order rather than in a 2D scan line order to maximize locality.

Cox et al. [3] examined the use of a large secondary cache as a mechanism to take advantage of frame-to-frame coherence in texture data, finding that the inter-frame working set of texture data is on the order of several megabytes.

Vartanian et al. [12] have evaluated the performance of texture caching with both image-space parallel and object-space parallel rasterizers. They find that while object-space parallelism provides good speedup in a caching environment, image-space parallelism generates poor speedup. We believe that these results can be attributed to focused architectural choices and benchmark scenes that favor object-space parallelism both in terms of caching and rasterizer load balancing. In contrast, we find it more insightful to separate rasterizer load imbalance from texture load imbalance, and by exploring a more complete set of architectural choices, we find efficient design points for texture caching with both types of parallelism.

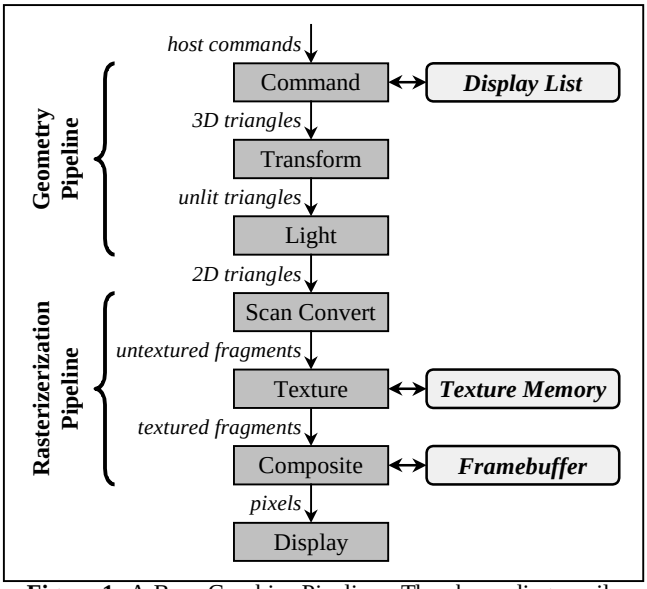


Figure 1: A Base Graphics Pipeline. The above diagram illustrates a typical graphics pipeline. A parallel rasterization architecture replicates the rasterization pipeline to achieve higher performance.

2.3 Unique Texel to Fragment Ratio

Igehy et al. [6] described and analyzed a prefetching architecture designed for texture caching that is able to tolerate arbitrarily high and variable amounts of latency in the memory system. In that paper, they present a measure of a scene's inherent intra-frame texture locality called the *unique texel to fragment ratio*. This ratio is the total number of unique texels that must be accessed in order to draw a frame divided by the total number of fragments generated for the frame, and it represents an upper bound on the effectiveness of a cache that cannot exploit inter-frame locality.

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 characterized by 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.

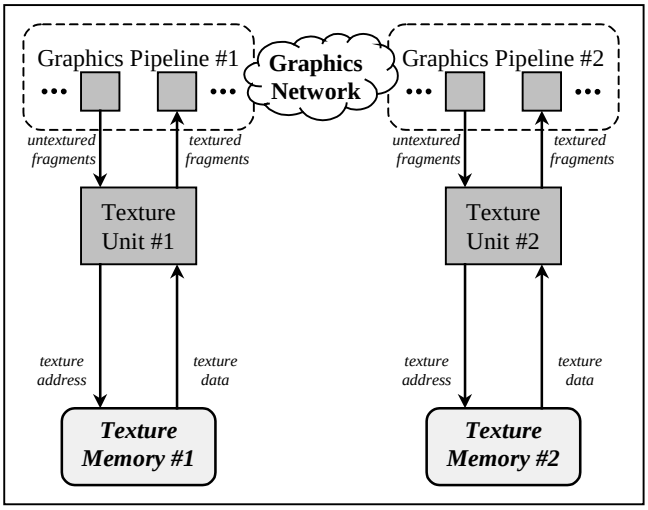


Figure 2: Dedicated Texture Memory. Multiple graphics pipelines simultaneously draw a scene by coordinating work over a graphics network. To apply texture, a fraction of the untextured fragments is distributed to each texture unit that holds a replicated version of the scene's texture data in a dedicated texture memory. Texture is cached to reduce texture memory bandwidth.

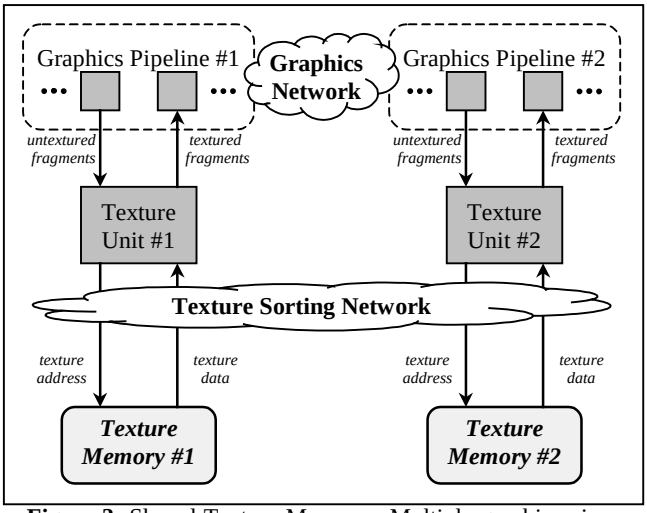


Figure 3: Shared Texture Memory. Multiple graphics pipelines simultaneously draw a scene by coordinating work over a graphics network. A fraction of the untextured fragments is distributed to each texture unit, and each texturing unit can access the texture data of any texture memory, allowing for a single copy of the texture data system-wide. Texture caching reduces both network and memory bandwidth.

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 across the objects in a scene.

3 PARALLEL TEXTURE CACHING

A serial graphics pipeline is illustrated in Figure 1; performance can be increased by deploying multiple copies of some or all of the stages. Parallel rasterization distributes rasterization work amongst multiple copies of the rasterization stages. Looking at the texturing stage specifically, the role of the texture mapping units in a system is to take as input untextured fragments with texture coordinate information, access the appropriate data in the texture memory based on these coordinates and filtering modes, filter the data, and combine this filtered texture value with the value of the untextured fragment. In order to scale the fragment rate (i.e., the rasterization performance), the number of texturing units must be increased to provide the necessary processing power. Additionally, the number of texture memories must also be scaled to provide the correspondingly increased bandwidth requirements.

Figure 2 shows a *dedicated* texture memory scheme for scaling the texture subsystem of a graphics pipeline. Each additional rasterization pipeline brings with it a dedicated texturing unit and texture memory. As the system scales, the total amount of texture memory increases, but due to replication, the *unique* texture memory remains constant. Figure 3 diagrams a *shared* texture memory scheme for scaling the graphics pipeline. In this architecture, an all-to-all texture sorting network is introduced between the texturing units and the texture memories. This allows any texturing unit to access the data in any texture memory, allowing a single shared image of the texture data to be present across all of the texture memories. Many topologies exist for such networks [4], and highly scalable networks can be built if the system balances the data going in and out of the network. We will not be focusing on the network in this paper.

With the architectures of Figure 2 and Figure 3, as with any parallel system, it is important to minimize the amount of redundant work introduced by parallelization and to balance the amount of work in each processing unit. Efficient parallel rasterization algorithms deal with presenting each texturing unit with a balanced number of untextured fragments that minimizes redundant work; this problem has been extensively studied, and we make use of a few such algorithms, as described in Section 4.1. The main focus of this paper is to study the effects of parallel rasterization on texture locality. Assuming that the number of untextured fragments presented to each texturing units is balanced, one requirement for good parallel performance is that the redundant fetching of the same texture data across texturing units be minimized. Furthermore, it is important to load balance the texture bandwidth required by each texturing unit, and in the case of a shared texture memory, the texture bandwidth required from each texture memory.

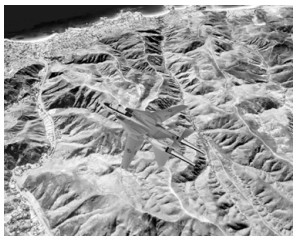
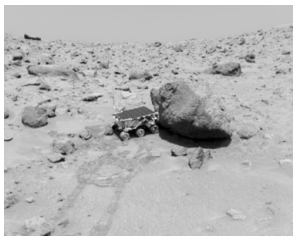
4 METHODOLOGY

While it is clear that the parallel architectures of Figure 2 and Figure 3 do potentially increase performance, the actual performance gains are still unclear. In this section, we lay out a framework that will allow us to evaluate the performance of a parallel texture caching architecture.

4.1 Parallel Rasterization Algorithms

The characteristics of parallel texture caching are highly dependent on the parallel rasterization algorithm because this algorithm determines which fragments are processed by which texturing units and in what order. There are a great number of different rasterization algorithms, and each algorithm has a number of parameters that can be varied. Because of the large number of variables, it is impractical to analyze every rasterization algorithm, and thus we choose a few representative algorithms.

Parallel rasterization algorithms can be characterized along three axes with regard to texturing. The first distinction to be

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%	38%	87%	0%	100%
unique texels/frag	0.033	0.092	0.706	1.55	0.569	2.83

Table 1: The Scenes.

made is whether work is partitioned according to image-space (each texturing unit is responsible for a subset of the pixels on the screen) or object-space (each texturing unit is responsible for a subset of the fragments generated by triangles). The second distinction is whether the texturing unit processes fragments immediately in the order primitives are submitted or buffers fragments and processes them in a different order. The third distinction is whether fragments destined for the same location in the framebuffer are processed in the order presented by the application. For this paper, all of the algorithms we present preserve application order.

- *tiled* In a tiled architecture, the screen is subdivided uniformly into fixed-size square or near-square tiles and each texturing unit is responsible for a statically interleaved fraction of tiles. We have empirically found that 32 pixel by 32 pixel tiles work well up to moderate levels of parallelism, and for this paper, we will assume that tile size. In *tiled-prim*, fragments are processed in primitive order. This means that if a triangle overlaps several tiles belonging to the same rasterizer, the fragments of that triangle are completely processed before moving on to the fragments of the next triangle. In *tiled-frame*, the fragments of a frame are processed in tile order. This means that a texturing unit processes all of the fragments for its first tile before moving on to any of the fragments that fall in its second tile.
- *osi* Algorithms that subdivide work according to object-space usually distribute groups of primitives in a round-robin fashion amongst rasterizers, giving each rasterizer approximately the same amount of per-primitive work. Because the number of fragments generated by each primitive can vary greatly, it is important to also load balance fragment work either by dynamically distributing primitives, by subdividing large primitives, or by combining the two techniques. In *object-space ideal* (*osi*), we idealize the load balancing of fragments. First, we serially rasterize all the primitives to form a fragment stream, and then we round-robin groups of 1024 fragments amongst the texturing units.
- *striped* Similar to both the RealityEngine and the InfiniteReality, fragments are subdivided according to an image-space subdivision of 2 pixel-wide vertical stripes. Each texturing unit is responsible for an interleaved fraction of the stripes, and processing is done in primitive order, as in *tiled-prim*.

4.2 Scenes

In order to quantify the effectiveness of parallel texture caching, we need to choose a set of representative scenes that cover a wide range of texture locality. A good measure of texture locality is the scene's unique texel to fragment ratio, and this ratio varies over nearly two orders of magnitude in our test scenes. The scenes we chose originated from three traces of OpenGL applications. In the future, we expect to see more texture for a given screen resolution, increasing 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 some key statistics from our six scenes, described below:

- *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 heavy 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. In *quake2x*, texture resolution is doubled, and only the light maps are magnified.
- *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. The order in which triangles are drawn is irregular: the terrain data is drawn in large tiles, but each tile of the terrain is drawn in sets of triangles that are not contiguous in screen-space. In *flight*, the majority of the texture (62%) is magnified, while in *flight2x*, only 13% is magnified, resulting in lower texture locality.
- *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 that are drawn in a regular order. Even though all of the texture is magnified, the lack of repeated texture keeps the number of unique texels per fragment high in *qtvr*. In *qtvr2x*, all of the texture data is minified. Furthermore, because the level-of-detail in most of the scene is just barely below a new mip map level, texture accesses incur a high bandwidth demand.

All of the above scenes make use of mip mapping for texture filtering. Mip mapping is crucial for providing locality in texture access patterns under minification, a characteristic that all texture caching rasterization architectures depend upon to run at full speed. Scenes that lack mip mapping will experience significant performance degradations under texture minification. The scenes we use in this paper load balance fragment work relatively well with respect to the parallel rasterization algorithms of Section 4.1, as will be quantified in Section 5.3. Because these scenes load balance well under our parallel rasterization algorithm, texture bandwidth imbalance will not be hidden by fragment imbalance.

4.3 Simulation Environment

A cycle-accurate simulator of a parallel texturing subsystem was written in C++ to provide an environment for collecting the data presented in this paper. Our simulation infrastructure is based on a methodology developed by Mowry [11] for simulating hardware architectures. The simulator takes as input texture data and untextured fragment data, and produces rendered images as well as trace data on the texture subsystem. The simulator is able to partition this fragment data among multiple instances of texturing units in accordance with the parallel rasterization algorithms of Section 4.1. Each texturing unit and each texture memory is made up of multiple functional units that run as threads in the C++ environment. The forward progress of each functional unit and the communication between functional units adhere to a simulation clock by making clock-awaiting function calls within each thread at the appropriate points in the code. This allows us to simulate a graphics architecture with cycle-accuracy at varying levels of detail and collect data from the system in a non-intrusive fashion.

4.3.1 Data Organization

Given the high-level architecture of parallel texturing units that are connected to memories through a texture cache, we must decide how data is organized throughout the system. In accordance with previous work [5], we group 2D texture data into 4D tiles so that each cache block holds a square or near-square region of texture and use an additional level of tiling (6D tiling based on the number of cache sets) to reduce conflict misses. This is illustrated in Figure 4. The exact parameters of the tiling are dependent on the cache parameters entered into the simulator.

Based on previous studies regarding rasterization order [5], we rasterize according to screen-aligned 8 by 8 tiles. We also add another level of tiling to rasterization (every 32 by 32 pixels), resulting in 6D tiled rasterization. To give a consistent rasterization order across the studies in this paper, a serial rasterizer generates untextured fragments in this order and distributes them to the appropriate texturing unit according to the parallel rasterization algorithm.

For the purposes of this paper, we assume that the texturing unit has enough computational power to perform a trilinear mip mapped texture filter for each fragment with no performance loss. According to Igehy et al. [6], a texture cache that is partitioned into two caches (one for even levels of the mip map, and one for odd levels of the mip map) allows conflict-free access to the eight texels of a trilinear interpolation. They demonstrate that in such a configuration, the miss rate does not significantly improve with increased associativity. We therefore use two direct-mapped caches for the studies in this paper.

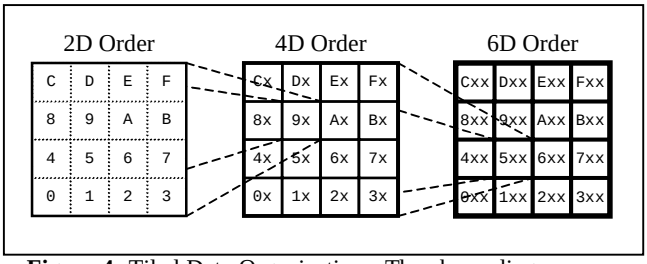


Figure 4: Tiled Data Organization. The above diagram correlates a location in an image with an address, expressed in hexadecimal. An 'x' represents an arbitrary hexadecimal number. The left block shows the layout of a 4x4 image in 2D order. A 4x4 grid of these 2D blocks gives rise to a 16x16 image laid out in 4D order, illustrated in the center. The first hexadecimal digit determines the block, and the second hexadecimal digit determines the pixel within that block. Similarly, a 4x4 grid of 4D blocks gives rise to a 64x64 image laid out in 6D order. Note that our choice of 4x4 blocking at each stage can be replaced with an arbitrary blocking factor.

For a shared texture memory architecture, we must decide on the distribution of texture data across the multiple texture memories. Texture data should be distributed in a finely interleaved fashion to prevent hot-spotting on any single memory for any significant period of time. In order to minimize the chance that nearby texture tiles fall onto the same texture memory, we distribute each cache block of texture data across the texture memories in a 4D tiled fashion. The exact parameters for this tiling are dependent on the cache block size and the number of texture memories used for a particular simulation. For example, with 16 texel cache blocks (organized in a 4 by 4 tile) and 4 texturing memories, each cache block in a 2 by 2 tile of cache blocks is given to a different texture memory.

4.3.2 Performance Model

While caching characteristics may be analyzed statically without a performance model, such a model must be introduced in order to analyze resource contention and parallel speedup. Our simulated texturing unit is capable of texturing a single fragment every cycle, and we provide 2 texels per cycle of bandwidth to each texture memory, an amount large enough to cover most of the bandwidth demands of our scenes. This is a typical bandwidth in modern systems – see, for example, a calculation by Kirk [7]. The latency of each texture memory is set to 20 fragment clocks, and a 64 fragment FIFO is used to hide the latency of the memory, values we replicate from previous work based on modern memory devices [6]. Because arbitrary amounts of latency can be hidden through the use of prefetching and a fragment FIFO, our results are not dependent on these values.

In a serial texturing unit, a fragment FIFO serves not only to hide the latency of the memory system, but also to smooth out variations in temporal bandwidth requirements. Even if a scene's overall bandwidth requirement is low, temporal bandwidth requirements can get extremely high when several consecutive fragments miss in the texture cache. If this temporal imbalance is

microscopic (e.g., over tens of fragments), then a fragment FIFO can smooth out the contention for the memory system. However, this imbalance is often macroscopic (e.g., over tens of thousands of fragments): a fragment FIFO is unable to resolve the fragment-to-fragment contention for the texture memory and performance suffers.

In a parallel texture caching system with shared texture memories, contention can also occur between texturing units for the texture memories, and thus, the network. In order to reduce the number of free variables in this paper, we choose to model the network as perfect (no latency and infinite bandwidth) and therefore focus on memory contention effects. Network contention is related to memory contention in a fully simulated system, and prefetching is able to successfully hide arbitrary amounts of network latency in texture caching.

5 RESULTS

Parallel texture caching can be analyzed according to common parallel algorithm idioms. First, parallel texture caching incurs redundant work in the form of repeated texture data fetching. This reduction in locality is quantified in Section 5.1. The effect of multiple caches on working set size is described in Section 5.2. Second, it is essential that parallel texture caching be load balanced, and we quantify this in Section 5.3. Finally, in Section 5.4, we use a cycle-accurate simulation to demonstrate that good parallel speedup does in fact occur.

Contrary to traditional microprocessor cache studies, we present cache efficiency data in terms of bandwidth per fragment rather than miss rate per access. In a microprocessor architecture, miss rate is of primary importance because only one or a few outstanding misses can be tolerated before the processor stalls. Because of the lack of write hazards, texture caching can tolerate arbitrary numbers of outstanding reads [6], and thus, performance is related more to its bandwidth demands.

5.1 Locality

As with most parallel algorithms, parallel texture caching induces inherent overhead beyond that found in a serial algorithm due to redundancies. For parallel texture caching, this is best characterized by the redundant fetching of the same texture data by multiple texturing units — a reduction of locality. In a serial graphics system, an ideal texture cache would fetch each texel used in the scene only once (assuming the cache is sized to exploit only intra-frame locality). The bandwidth required for such a cache can be computed by counting the number of compulsory misses (i.e., cold misses) taken by the cache that employs a block size of a single texel. As we make the block size larger, fewer misses are taken, but the amount of data read by each miss increases. Overall, we expect the total amount of data fetched due to compulsory misses to increase with the block size because of *edge effects*: whenever a texture falls across the edge of a screen, the silhouette edge of an object, or the edge of a parallel work partitioning, larger block sizes force the cache to fetch larger portions of the texture data that are never used. By measuring the bandwidth attributable to compulsory cache misses, Figure 5 illustrates the reduction of locality caused by the various rasterization algorithms as the number of texturing units and block size are varied.

The lightest portions of the bars in Figure 5 indicate the average bandwidth required to satisfy the compulsory misses of a serial texture cache for the various rasterization algorithms. For a serial texturing unit, all of the algorithms perform equally because the number of compulsory misses is scene-dependent. We see that as block size is increased, the bandwidth requirement for a serial rasterizer increases slightly for the *flight* data set pair and negligibly for *quake* and *qtvr* data set pairs. In *qtvr*, the edge effects occur only near screen edges, which accounts for a negligible portion of the total work. In *quake*, the texture used at the edge of polygons is repeated from the middle of the polygons, thus negating edge effects from polygons.

The bottom portion of each bar represents the optimal bandwidth requirements of a serial texture cache, and each successive portion of the bar represents the additional bandwidth required to satisfy additional texturing units. We simulate an *infinite* number of texturing units, the top-most portion of each bar, by assigning the smallest granularity of work for each rasterization algorithm to a unique texture unit. For a tiled architecture this quantum of work corresponds to a single tile, for object-space interleaving this corresponds to a single contiguous block of fragments. This defines the locality present in a rasterization algorithm's minimal unit of work. Also note that because we are counting compulsory misses, the order in which fragments are processed has no effect, and thus the results for *tiled-prim* and *tiled-frame* are identical.

As a detailed example, for a *tiled* architecture on the *flight2x* scene, we see that for a block size of 16 texels (arranged in a 4 by 4 tile), a single texturing unit requires approximately 1.67 texels per fragment. If work is distributed amongst two texturing units, then the bandwidth required increases to approximately 2.17 texels per fragment. This occurs because edge effects are introduced at the boundaries of tiles, reducing the tile-to-tile locality. For four texturing units, the bandwidth requirement slightly increases to 2.26 texels per fragment, but as work is distributed amongst additional texturing units, the bandwidth requirements do not increase significantly. The reason for this is that most of the texture in the scene is unique, and while the tiles of a two-way parallel system touch at their corners and thus share some of the texture data of an object (tile-to-tile locality), this adjacency goes away completely in four-way parallel and larger systems. We also see that as block size is increased from 1 to 16 to 64 texels, the bandwidth requirements increase significantly because the over-fetching of larger block sizes is multiplied by the large number of tile edges. These aforementioned behaviors are all mirrored in *flight*, *qtvr*, and *qtvr2x*.

Although the effects of larger block sizes are the same, the bandwidth requirements of *quake* and *quake2x* on the *tiled* architecture are quite different as the number of rasterizers is increased. The first thing to notice about these scenes is the low bandwidth requirements of the serial case due to the heavy use of repeated textures. Furthermore, as opposed to the other scenes, as the number of texturing units is increased, the bandwidth requirements always increase. The use of repeated textures causes this because the same texture data is used repeatedly across the image-space partitioning. However, even with an infinite number of texturing units, the total bandwidth requirement is still quite limited. In effect, although parallel rasterization diminishes texture locality due to repeated texture, locality due to filtering remains. This means that texturing subsystems that are designed to perform well only in conjunction with repeated textures do not parallelize well.

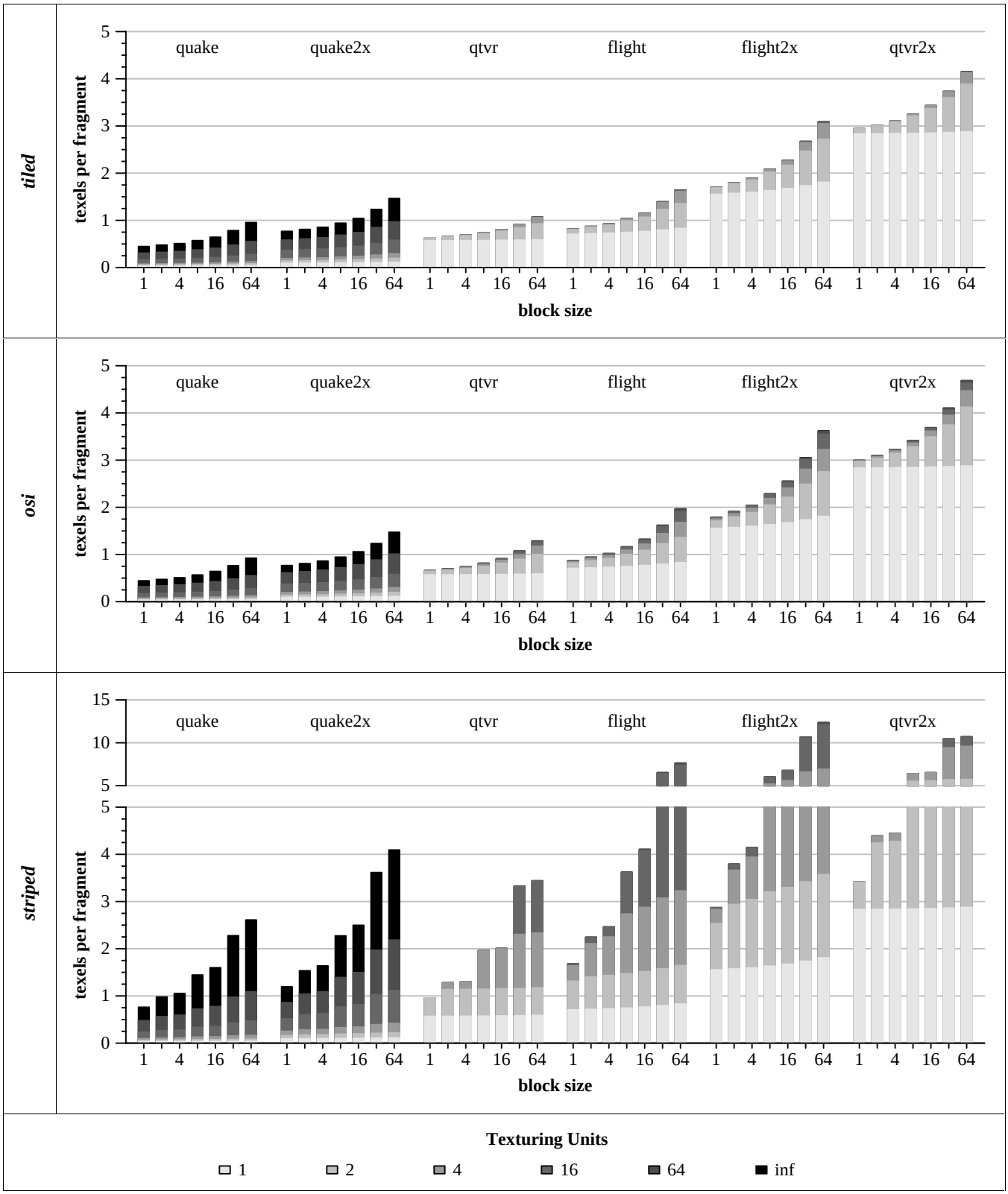


Figure 5: Bandwidth Due to Compulsory Misses. This study shows the bandwidth requirements (measured in average number of texels fetched per fragment) associated with compulsory misses as a function of rasterization algorithm, scene, block size, and number of texturing units. The top row represents the data for the *tiled* rasterization architecture, the middle row for the *osi* architecture, and the bottom row for the *striped* architecture. Scenes are sorted left to right by their unique texel to fragment ratio, which indicates the minimum bandwidth required. Each bar chart shows the bandwidth requirements for a different block size, and the shades of gray show the bandwidth requirements for differing numbers of texturing units. The shades of gray increase in darkness as the number of texturing units is increased, and the bandwidth required for greater numbers of texturing units increases monotonically. Finally, note that for clarity, the bandwidth values for *striped* rasterization are shown with a split scale axis.

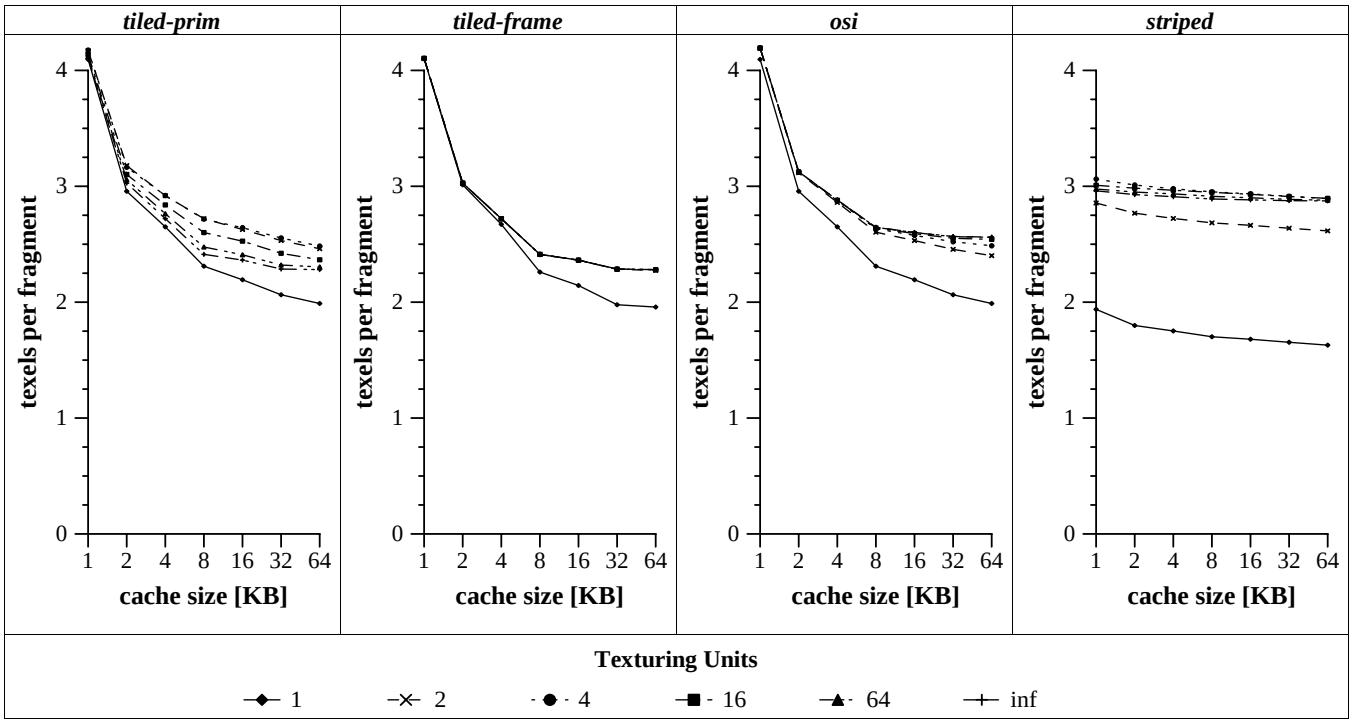


Figure 6: The Effects of Cache Size. The total bandwidth (measured in average number of texels per fragment) required to render *flight2x* is plotted as a function of the cache size. Each chart shows a different rasterization architecture, and each curve represents a different number of texturing units. Block size is set to 16 texels for all the graphs except *striped*, which has a block size of 1 texel.

The behavior of *osi* largely mirrors the performance of *tiled*, with the exception that bandwidth requirements continue to increase as additional texturing units are utilized. This is explained by the fact that *osi* is fragment-interleaved, and the chance that a texturing unit's consecutive fragment groups utilize adjacent portions of a texture map decreases smoothly as the number of texturing units is increased. For both *tiled* and *osi*, we see that a block size of 16 texels provides reasonable locality given the granularity of access needed for efficient memory utilization and efficient network utilization. Thus, for the remainder of the paper, we assume a block size of 16 texels for *tiled* and *osi*.

The behavior of the *striped* rasterization algorithm is markedly different from both *tiled* and *osi*. The most important thing to notice is that bandwidth requirements increase dramatically with increased block size. Because interleaving is done at every 2 pixels in the horizontal direction, edge effects occur very frequently. As block size is increased, a drastically larger number of texels that fall beyond a stripe's required set of texels are fetched. Thus, striped architectures reduce texture locality drastically. Even at a block size of 1 texel for *striped*, locality is much worse than at a block size of 16 for *tiled* or *osi*. We note that a single texel is a very small granularity of access for modern networks and memories, and that most modern devices perform highly suboptimally at such granularities. Nonetheless, this is the only block size that preserves a modicum of locality for *striped*, and for the remainder of the paper, we assume a block size of 1 texel for the *striped* architecture.

5.2 Working Sets

Now that we have an understanding of the effects of cache block size on locality under parallel rasterization, we move onto the effects of parallel rasterization on working set sizes by using limited-size caches that suffer misses beyond compulsory misses. As the number of texturing units is increased, the total amount of cache in the system increases, and thus we expect better performance. Figure 6 quantifies this notion by showing the bandwidth requirements of the various architectures with differing numbers of texturing units for the *flight2x* data set as the total cache size is varied. In general, there is a correlation between an algorithm's working set size and the point of diminishing returns in increasing cache size, illustrated as the "knee" in the curves of Figure 6. We see that as the number of texturing units increases, the working set size for each texturing unit decreases.

These same characteristics were found for all of the data sets. Because we want to pay attention to low levels of parallelism and systems that scale a serial texturing unit, we focus on a single cache size that works well for a serial algorithm. Choosing such a parameter outside of hardware implementation constraints is a bit of a black art, and thus we use parameters from previous work [6] for consistency's sake and allocate a cache size of 16 KB (configured as two direct-mapped 8 KB caches) for the remainder of this paper. Figure 7 shows the bandwidth requirements of the various algorithms on the various scenes with a 16 KB cache. The first trend we notice is that while there is an initial jump in bandwidth demand when going from one texture unit to two texture units, the bandwidth demands are largely flat with increasing numbers of texture units. Moreover, for some traces, particularly flight and flight2x, the bandwidth demands actually decrease after the initial increase. This is a well-known phenomenon from parallel systems wherein the aggregate cache size increases more rapidly than the aggregate miss rate, resulting in improved cache behavior with increasing parallelism.

One interesting result is that although *tiled-frame* performs better than *tiled-prim* for the *flight* and *qtvr* data set pairs, the opposite is true for the *quake* data set pair. In *flight*, and to a lesser extent *qtvr*, the disjoint drawing of triangles in image-space

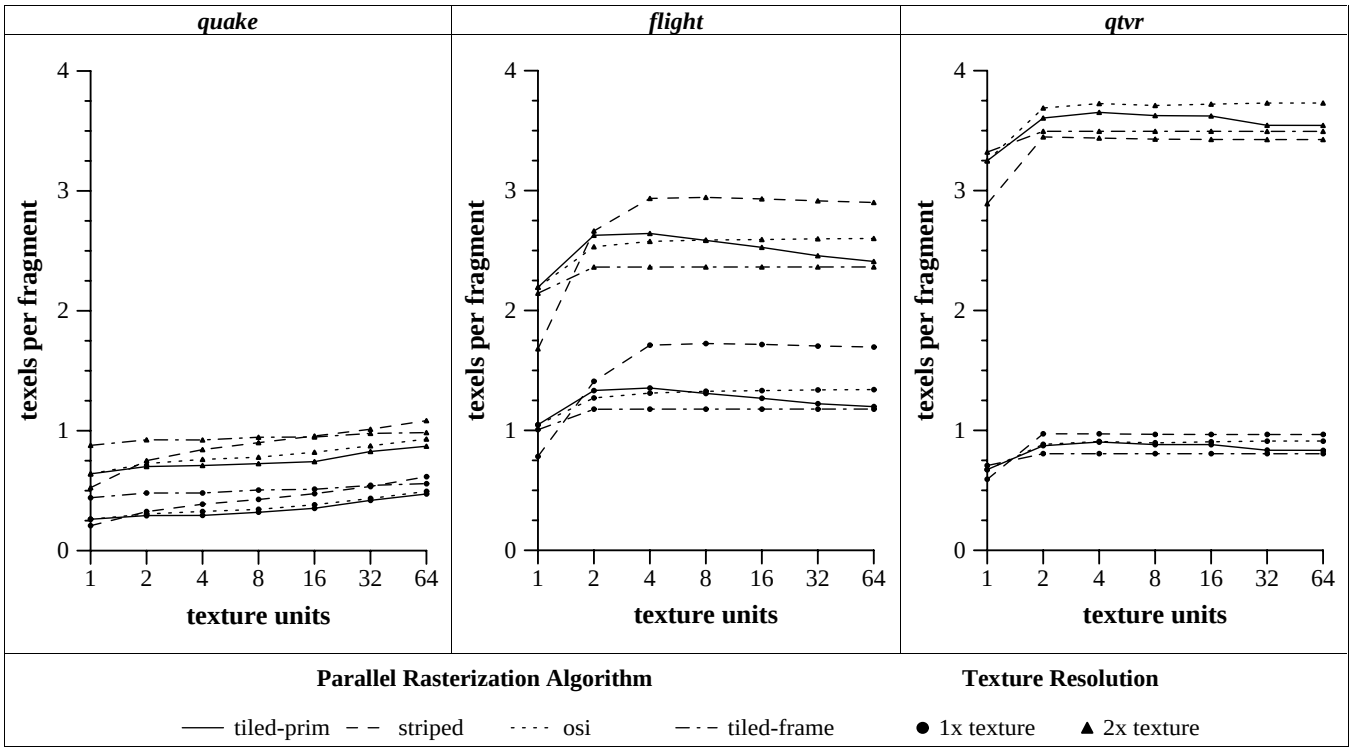


Figure 7: Bandwidth Requirements of a 16 KB Cache. In these graphs, bandwidth is displayed as a function of the number of rasterizers. Both the normal and the 2x resolution versions of each scene are shown on the same graph. Block sizes are the same as in Figure 6, and each curve shows the bandwidths for a different parallel rasterization algorithm.

makes it advantageous to wait until all of the triangles of a tile are present before texturing due to increased temporal locality. In *quake*, however, it is more advantageous to texture large polygons that fall into multiple tiles immediately because the different regions of the polygon all use the same repeated texture data.

5.3 Load Imbalance

The performance of any parallel system is often limited by load imbalance: if one unit is given significantly more work than the other units, it will be the limiting factor, and performance will suffer. In parallel texture caching, load imbalance can occur in one of three ways. First, the number of untextured fragments presented to each texturing unit can differ. Second, the bandwidth required for texturing the fragments of a texturing unit may vary. Third, the bandwidth required from each texturing memory can differ. In a dedicated texture memory system, the last two sources of imbalance are identical because each texturing unit is paired with a single texture memory.

Figure 8 shows the various types of load imbalance of the various scenes on the different architectures. The first trend to note is that all of the configurations load balance well in all respects when there are 16 or fewer texturing units (the worst imbalance is 9.7%). However, as the number of texturing units is increased to 32, and especially 64, there is a large imbalance in the bandwidth requirements of the texturing units. This imbalance is significantly larger than fragment imbalance, and the trend occurs in all of the rasterization algorithms except *striped* and on all of the data sets except the *qtvr* pair, which exhibits extreme regularity in texture access patterns. The *striped* algorithm is highly load balanced even at high numbers of texturing units because of its fine interleaving pattern. However, this positive result is moderated by the fact that the baseline for the *striped* data includes significantly more redundant bandwidth than the other rasterization algorithms.

The second important trend in Figure 8 is the effect of shared texture memory on texture memory load imbalance. In a dedicated texture memory system, the load imbalance between texture memories is equal to the load imbalance between texturing units. In a shared texture memory architecture, the load imbalance between texture memories is relatively small. Thus, we see that distributing blocks in a tiled fashion across the texture memories does in fact balance texture load well, usually to such an extent that shared texture memory imbalance is much lower than the dedicated texture memory imbalance

5.4 Performance

While the experiments of the previous sections illuminate many characteristics of parallel texture caching, they say nothing about realized performance. In particular, the temporal effects of erratic intra-frame bandwidth demands are ignored. Even though a scene's memory bandwidth demands may be low when averaged over an entire frame, its memory bandwidth demands averaged over a few fragments can actually be quite high. Figure 9 divides the execution time of a serial texturing unit into three categories: time spent on fragment processing, time lost due to insufficient memory bandwidth, and time lost due to fragment-to-fragment contention for the memory. We see that only *flight2x* and *qtvr2x* have average memory bandwidth requirements beyond 2 texels per fragment, and thus even perfect smoothing of fragment-to-fragment contention could not achieve a performance of one cycle per fragment. We also see that contention time is nearly uniform across all rasterization architectures with the exception of *striped*, which performs better due to smaller block sizes. In general, uncovered contention occurs in scenes that have large variations

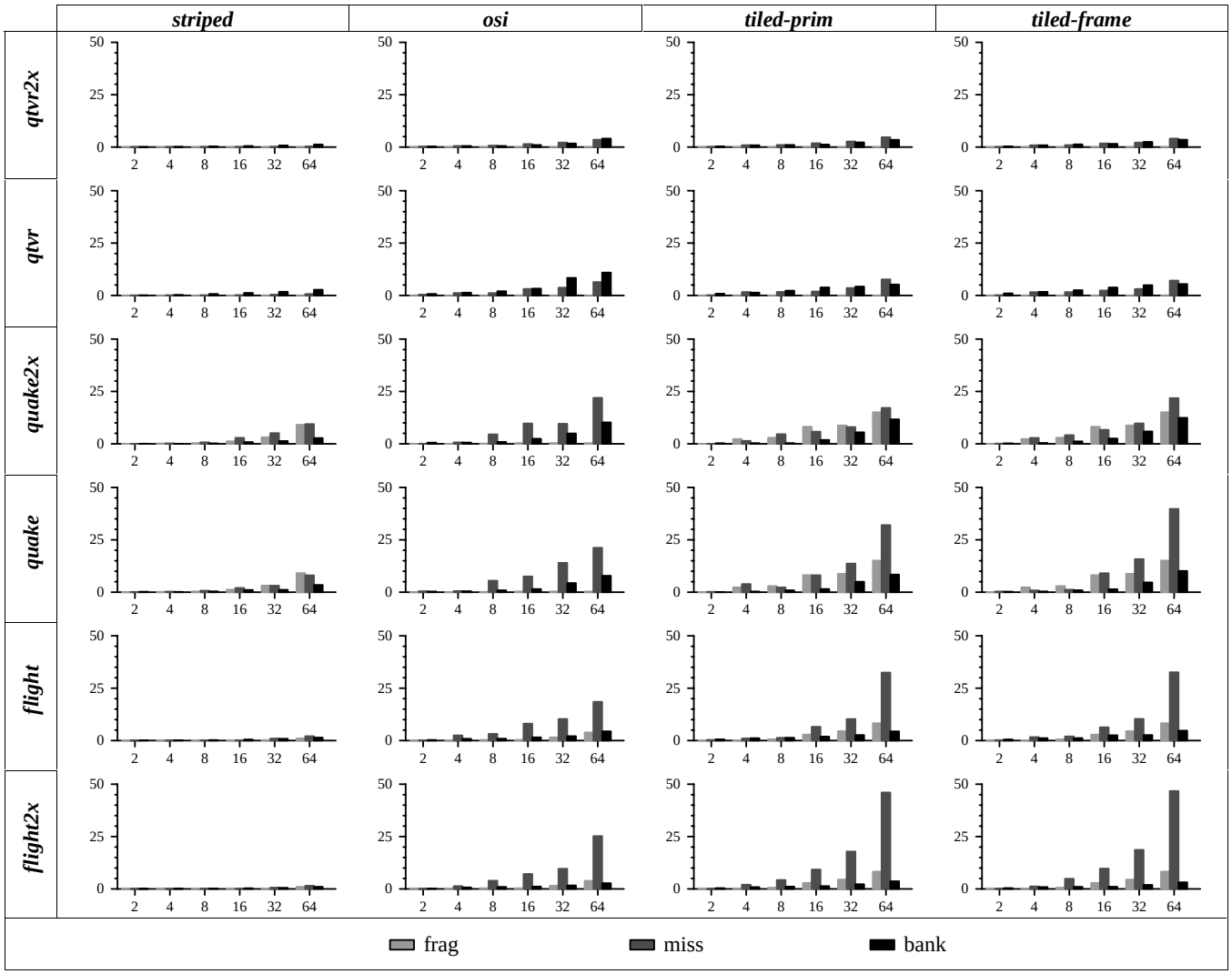


Figure 8: Load Imbalance. Each graph shows load imbalance for different numbers of rasterizers. The y-axis of each graph shows the percent difference in the work performed by the busiest unit and the average unit. Each row shows a different scene, and each column shows a different parallel rasterization algorithm. The rows and columns have been sorted so that the scenes and rasterization algorithms that perform best are at the upper left and the ones that perform worst are at the lower right. Three types of load imbalance are shown. Fragment load imbalance is the maximum number of fragments given to a texturing unit divided by the average number of fragments per texturing unit. Miss load imbalance is the worst rasterizer's number of misses per fragment divided by the average number of misses per fragment. Bank load imbalance is the maximum number of access per texture memory in a shared memory architecture divided by the average number of accesses per memory. For these experiments, we use the same cache and block sizes as the experiments in Figure 7.

in macroscopic bandwidth requirements. Whenever bandwidth requirements peak beyond 2 texels per fragment over large periods of time, temporal memory contention cannot be resolved by the 64-entry fragment FIFO. This occurs most in *flight2x*, and to a lesser extent, in *flight* and *quake2x*. In *qtvr2x*, temporal bandwidth requirements are always over 2 texels per fragment, and in *quake* and *qtvr*, bandwidth requirements are consistently under 2 texels per fragment, resulting in little temporal contention that is not covered by the 64-entry fragment FIFO.

The serial runs of Figure 9 serve as a baseline for computing speedup of parallel texture caching runs. In the first row of Figure 10, we graph the speedup of a dedicated texture memory architecture. Across all of the runs, excellent speedup is achieved through 16 texturing units. For the scenes whose bandwidth requirements are usually met by the 2 texel per cycle memory system – *quake, quake2x, flight* (except in *striped*), and *qtvr* – the speedup is near-linear. For the scenes that exceed this bandwidth – *flight2x* and *qtvr2x* – the speedup efficiency is dictated by the inefficiency of the cache with respect to a serial cache, as graphed in Figure 10. Beyond 16 texturing units, some of the speedup curves exhibit lower speedup efficiency. Referring back to the load imbalance graphs in Figure 8, we see that this occurs in the configurations that exhibited significant load imbalance in the amount of bandwidth requested by each texturing unit. As expected intuitively, the fragment-to-fragment memory contention plays an insignificant role in speedup efficiency.

In the second row of Figure 10, we graph the speedup of a shared texture architecture. The speedup efficiencies realized are almost identical to a dedicated texture architecture at or below 16 texturing units. At higher numbers of texturing units, however,

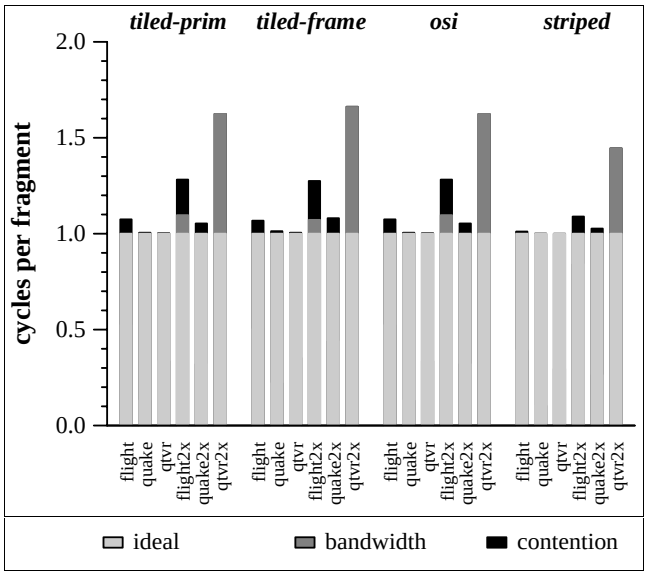


Figure 9: Breakdown of Serial Time. This graph breaks down the execution of a serial texturing unit across different scenes and rasterization algorithms. Execution time is normalized to cycles per fragment, and the light gray bar shows the ideal cost, assuming one fragment is processed per cycle. The dark gray bar shows the cost of insufficient memory bandwidth for a 2 texel per cycle memory system. If the ratio of a scene's average memory bandwidth requirement to the memory system's bandwidth supply is greater than one, then this cost is tallied. The black bar represents the cost of fragment-to-fragment memory contention incurred by the 64-entry fragment FIFO's inability to smooth out temporal bandwidth variations. These experiments use the same cache parameters as used in Figure 7.

these speedup efficiencies are generally better than a dedicated architecture for the configurations that exhibited large load imbalance. This can be explained by the fact that in a shared texture architecture, the texturing unit with the highest miss rate spreads its memory requests across many texture memories, and thus performance becomes dependent on load imbalance amongst the texture memories, which is much lower than texturing unit imbalance. In effect, the busiest texture unit is able to steal memory bandwidth that would go unused in a dedicated texture memory system.

6 DISCUSSION

6.1 Texture Updates

One reason parallel texture caching is so straightforward to implement is that texture mapping is mostly a read operation. This allowed us to use multiple caches on the same data without the need for a complex coherence protocol to detect changes in the texture memory caused by another texturing unit. However, any real system with a shared texture memory will have to deal with such texture updates. There are two potential hazards when a texture is updated. First, a texture read for a fragment generated by a polygon submitted before the texture update could read the new contents of the texture rather than the old contents. The converse could also occur when a fragment generated by a polygon submitted after a texture update reads texture values that are stale. These problems arise both because texture updates are large events which do not necessarily occur atomically, and, more simply, because the work being performed at any point in time by one texture unit (e.g., downloading a texture) is generally not tightly coupled with the work being performed by another texture unit (e.g., processing fragments).

One solution to these hazards is to force texture updates to occur atomically and to introduce a strict ordering of operations between the texturing units during texture update. This is most naturally expressed by performing a barrier operation across all of the graphics system at the start of the texture update to ensure that previous accesses to the old texture have completed, and a second barrier operation at the end of the texture update to ensure that the new texture is in memory before any new fragments access it. The barriers could be implemented either in hardware via a shared signal among the texturing units, thus allowing the rest of the pipeline to make progress, or in the software driver, forcing a flush of the graphics pipelines. Additionally, texturing units must flush stale data from their caches in conjunction with texture updates.

6.2 Feasibility

While we have deliberately avoided the consideration of what interconnection network to use in a system which implements a shared texture memory, this is not due to inherent difficulty. A number of well-known networks exist that provide scalable, pin-efficient point-to-point interconnections suitable for carrying the relatively high bandwidth demands of a shared memory system. Our ability to hide memory latency, and by extension network latency, for texture mapping makes it easy to incorporate such a network into a typical rasterization architecture. While architectures to date have focused on replicated memories, and thus necessarily try to minimize the number of rasterizers, this study shows that a high degree of rasterization parallelism can be efficiently supported on a shared texture memory system.

7 CONCLUSION

In this paper, we demonstrated that parallel texture caching works well across nearly two orders of magnitude of parallelism, from a serial texture unit up to 64 texture units. For example, *tiled* rasterization using parallel texture caching exhibits over 84% utilization at 32 processors across all the scenes in our studies. Parallel texture caching is general enough to work with a variety of algorithms, including both image-space and object-space, as well as primitive order and tile order architectures. While *tiled* architectures typically exhibit significantly better cache behavior that *striped* architectures, parallel texture caching remains quite effective with a sufficiently small block size for *striped* architectures. We confirmed that the working set size *decreases* with increasing parallelism, allowing the use of a serial architecture's texture cache for a parallel architecture. We also demonstrated that a shared texture memory system not only has the obvious benefit of eliminating replicated textures, but it also further increases performance over a dedicated memory system by distributing contention over multiple memories.

Acknowledgements

We would like to thank Kekoa Proudfoot, Gordon Stoll, Milton Chen, John Owens, and the rest of the Stanford Graphics Lab for their insights about this work. Financial support was provided by NVIDIA, Intel, the Fannie and John Hertz Foundation, and DARPA contract DABT63-95-C-0085-P00006.

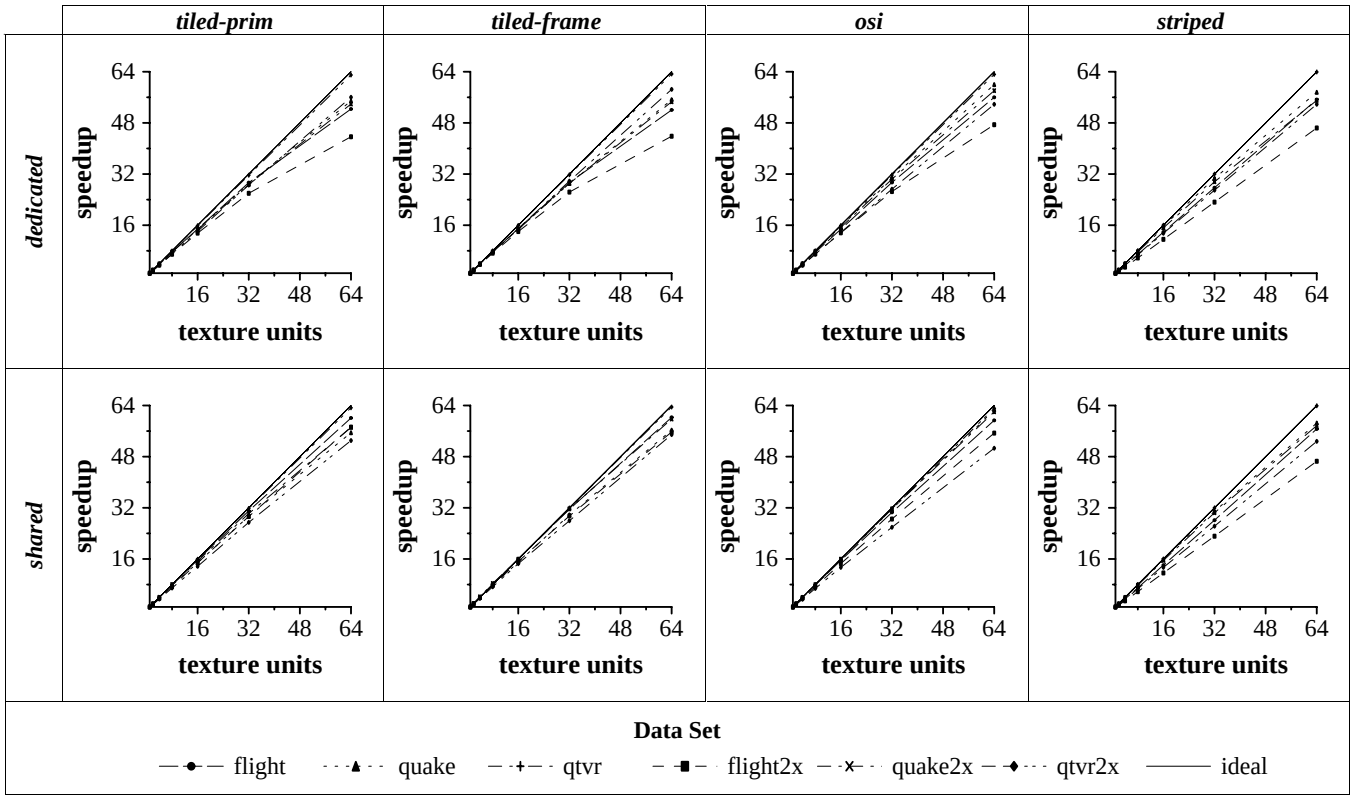


Figure 10: Speedup Graphs. These speedup graphs show speedup as a function of the number of texturing units. The top set of graphs show speedup curves for dedicated texture cache architectures, and the bottom set of graphs show speedup curves for shared texture memory architectures. Each column holds a different parallel rasterization algorithm, and the curves on each graph show speedup for a different scene.

References

[1] K. Akeley. RealityEngine Graphics. *Computer Graphics* (SIGGRAPH 93 Proceedings), **27**, 109-116, 1993.

[2] S. E. Chen. QuickTime VR: An Image-Based Approach to Virtual Environment Navigation. *Computer Graphics* (SIGGRAPH 95 Proceedings), **29**, 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] J. Duato, S. Yalmanchili, and L. Ni. *Interconnection Networks*. IEEE Computer Society Press, 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] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a Texture Cache Architecture. *Proceedings of the 1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 133-142, 1998.

[7] D. Kirk. Unsolved Problems and Opportunities for High-quality, High-performance 3D Graphics on a PC Platform. *Proceedings of the 1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 11-13, 1998.

[8] S. Molnar. The PixelFlow Texture and Image Subsystem. *Proceedings of the 10th Eurographics Workshop on Graphics Hardware*, 3-13, 1995.

[9] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. *Computer Graphics* (SIGGRAPH 92 Proceedings), **26**, 231-240, 1992.

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

[11] T. Mowry. *Personal Communication*. Carnegie Mellon University, 1999.

[12] A. Vartanian, J. Béchennec, and N. Drach-Temam. Evaluation of High Performance Multicache Parallel Texture Mapping. *Proceedings of the 12th ACM International Conference on Supercomputing*, 289-296, 1998.