A Superscalar 3D Graphics Engine

Andrew Wolfe and Derek B. Noonburg

S3 Incorporated
2841 Mission College Blvd.
Santa Clara, CA 95052
{awolfe,dnoonbur}@s3.com

Abstract

3D graphics performance is increasing faster than any other computing application. Almost all PC systems now include 3D graphics accelerators for games, CAD, or visualization applications. Many of the microarchitectural techniques that have been used to enhance the performance of microprocessors can be applied to graphics systems as well. We present an architecture for an out-of-order, superscalar rasterizer for 3D graphics. This allows the concurrent execution of multiple graphics primitives while maintaining exact sequential semantics. Experimental results show 1.5-3.6X speedups on real applications using a simple model, similar to the results from many integer benchmarks on superscalar processors. Enhanced techniques specific to 3D graphics for decomposing large triangles and breaking false dependence chains increase performance to more than 10x a sequential system.

1. Introduction

Since the widespread introduction of graphical user interfaces in computers more than 15 years ago, special-purpose graphics accelerators have been an integral component of desktop computer systems. Recently, 3D applications such as computer gaming, CAD, and visualization applications have been pushing high-performance 3D acceleration from specialty workstations into mainstream PCs. The demand for increased 3D performance is currently insatiable. Single-chip 3D accelerators today are 50 times faster than those available 3-4 years ago. Each generation substantially improves the quality of the images and yet another factor of 100 still would not produce truly realistic scenes.

Like all computing systems, in order to achieve such rapid increases in performance, it is necessary to improve the microarchitecture of each generation to increasingly take advantage of parallelism. In many ways, the evolution of graphics microarchitectures has paralleled the development of microprocessor microarchitectures. In the past few years, graphics accelerators have used wider data words, registers, and buses to increase performance. PC graphics accelerators have moved in the past few years from 32 to 64 to 128-bit memory buses in much the same way that microprocessors grew from 4 to 8 to 16 and eventually to 64 bits. This mechanism has diminishing returns, especially as applications move to a large number of small polygons in each scene.

Current high-performance microprocessors all use instruction-level parallelism (ILP) to further increase performance. ILP exploits information about dependences between instructions to allow parallel execution of multiple instructions while maintaining the execution semantics of a sequential program [6]. Many different ILP mechanisms can be effectively employed to improve performance, however dynamically scheduled, out-of-order, superscalar microprocessors are the commercially dominant microarchitecture at the present time. This paper presents an approach for using dynamically scheduled, out-of-order, superscalar principles for increasing the available parallelism in a graphics accelerator while maintaining sequential execution semantics.

In the abstract, it would seem that graphics, and particularly 3D graphics, is a massively parallel application that would not need ILP technology for high performance. In fact, in many simple graphics applications, it would be possible to render each pixel independently; however, in practice, graphics applications have very similar characteristics to traditional sequential programs. Standard APIs like Direct3D or OpenGL are used to create graphics applications. These are translated via software drivers to a sequence of graphics primitives to be rendered by a graphics engine that consists of some combination of additional hardware and software. The programming model assumes that these primitives will be executed sequentially and atomically, in much the same manner that it is assumed that instructions in a traditional sequential ISA are executed. In order to implement a parallel system that executes this sequence of primitives, the semantics of sequential execution must be maintained. Several factors cause dependences between graphics primitives that can prevent concurrent execution.

Z-buffering – A realistic 3D rendering systems must include hidden surface removal. Objects that are behind other objects from the perspective of the viewer should not be visible in the final image. A Z-buffer is used to implement hidden-surface removal. The Z-buffer stores the distance from the viewpoint to the currently drawn pixel so that when drawing a subsequent pixel, it can be determined if the new pixel is in front of or behind the currently drawn pixel. A well-implemented Z-buffering algorithm produces the same result if the triangles are drawn in a different order, however, it requires an atomic read-modify-write of Z-buffer data. If two triangles are drawn concurrently and they attempt to concurrently read the same Z-buffer value, modify it, then write it using common read and write operations, incorrect results can be produced. A special type of dependence thus exists between any two primitives that may access the same Z-buffer value. They can execute in either order but not concurrently.

Alpha-blending – Alpha-blending is an operation that uses a transparency value (alpha) to permit some portion of an occluded object to be visible through a foreground object. Unfortunately, the alpha-blending operation that implements transparency is order dependent. Programmers will generate groups of primitives in a known order in order to overcome this limitation of alpha blending. If the rendering engine does not maintain the semantics of the sequence, the image will be incorrect.

Dynamic textures, procedural textures, environment mapping – Realistic 3D graphics map an image (called a texture) onto each polygon. Often these textures have limited lifetimes. Procedural textures are created on-the-fly by program code. Dynamic textures are loaded into the graphics system memory space from some backing store for a limited time. Environment mapping is a technique for reflections where some 3D objects are drawn then copied as an image to be mapped onto a reflective surface. In each of these cases, there is a dependence between the primitives that create the texture and the primitives that render the polygon that uses the texture.

2D BLITs – It is advantageous to be able to mix 2D block copy and drawing operations with 3D rendering. If overlapping 2D objects are read or written out of order, the resulting image is incorrect.

Direct frame buffer access – The common graphics APIs allow blocks of frame buffer memory to be directly read or written by the main processor. This requires that precise frame buffer state be available whenever a direct access executes. This creates a dependence between previous primitives and a direct read or subsequent primitives and a direct write.

As in general-purpose computing systems, it is possible to build massively parallel systems that provide excellent performance on a limited set of applications that have been programmed with parallel execution in mind. However, in order to be compatible with a large existing base of software using widely accepted programming interfaces and programming styles, it is necessary to detect the dependences between graphics primitives, extract independent primitives from the instruction stream, and execute them concurrently. In this paper we present a method to detect and represent dependences between graphics primitives, an out-of-order engine to execute multiple independent primitives concurrently, and a datapath architecture that provides concurrent resources for parallel execution.

The following section of this paper provides a brief introduction to hardware-accelerated 3D graphics and the current mechanisms for implementing hardware. Section 3 describes a new out-of-order ILP architecture for the rendering stage of the graphics pipeline and relates the ILP principles to well-known concepts in microprocessors. Section 4 describes simulation experiments that measure the increased parallelism provided by this microarchitecture under a range of implementation parameters. Section 5 includes conclusions and suggestions for future investigations.

2. Background

2.1. 3D pipeline

A 3D application creates a series of frames. Each of the objects in a frame is broken down into a collection of triangles, typically covering the surface of the object. From the point of view of a 3D engine, then, a frame consists of a collection of triangles. The 3D engine is responsible for turning this list into a rendered frame.

Each triangle is specified by three vertices in a 3D space, one or more surface normals, and a description of how to draw the triangle's surface: texture, alpha blending parameters, etc. The 3D pipeline (see Figure 1) takes the triangles, applies lighting to them, transforms them from the 3D space used by the application into 2D screen coor-

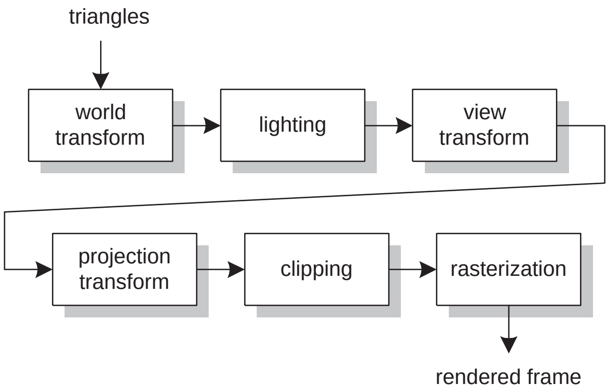


Figure 1: The 3D pipeline.

dinates, and draws the appropriate pixels into the frame buffer [4,5,9].

The first pipe stage, the world transform, converts the vertices and normals from object space, which may be different for each object in the scene, to world space, shared by the entire scene. This transform consists of a matrix-vector multiplication for each vertex and each normal.

The lighting stage takes the triangle's color and surface normal(s) and computes the effect of one or more light sources. The result is a color at each vertex.

The view transform converts the vertices from world space to camera space, with the viewer (or camera) at the origin. This uses a matrix-vector multiplication for each vertex.

The projection transform maps vertices from camera space to the viewport. This includes the perspective transformation. At this point, the vertices are effectively two-dimensional, i.e., perspective effects (depth foreshortening) have been applied, and so the third (z) coordinate is only needed to indicate the relative front-to-back ordering of the vertices. Like the other two transforms, projection requires one matrix-vector multiplication per vertex.

The clipping stage clips triangles to a view volume. Triangles which lie entirely off the side of the screen or behind the viewer are removed. Triangles which are partially out of bounds are trimmed, which generally requires splitting the resulting polygon into multiple triangles.

The rasterization stage converts a triangle into pixels and computes the color value at each pixel. This includes visible-surface determination (dropping pixels which are obscured by a triangle closer to the viewer), texture mapping, and alpha blending (transparency effects).

The current generation of PC graphics accelerators perform only the rasterization stage of the pipeline. The earlier stages are handled in software, and the transformed and lit vertices are then handed off to the hardware. Accordingly, the 3D rendering engine described in this paper performs the rasterization stage of the pipeline, and so this part will now be described in more detail. Next generation PC 3D hardware will encompass the whole pipeline.

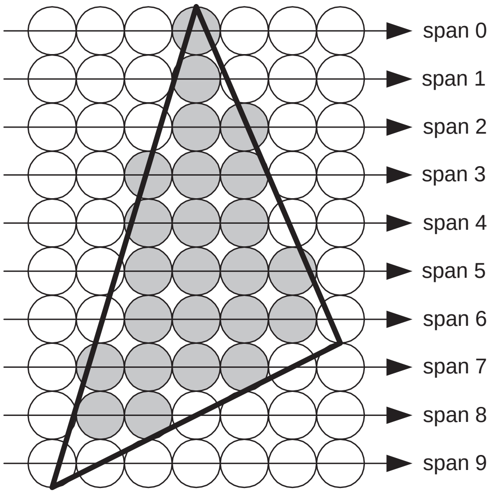


Figure 2: Converting a triangle to spans and pixels. Each horizontal line of pixels is a span. The gray shading shows pixels which are considered part of the triangle.

The rasterizer first converts a triangle into a series of horizontal spans (see Figure 2). Pixels are centered at integer x and y values (in Direct3D; OpenGL uses a different convention). One span is generated for each integer y value that falls inside the triangle. For each span, the rasterizer computes the two endpoints. This includes interpolated color values and perspective-corrected texture coordinates. Next, the rasterizer generates the series of pixels along the span, again interpolating color and texture coordinates. Several operations are performed at each pixel. First the pixel's z (depth) value is compared to the current z value for that pixel in the frame buffer. If the comparison indicates that this new pixel is behind the old one, the new pixel is discarded. If the z test succeeds, the new pixel color is computed. This can include texture mapping and alpha blending.

2.2. Rasterizer

The operations performed by the rasterizer can be divided into three categories:

- operations performed once per triangle: this includes computing interpolation parameters for span endpoint generation
- operations performed once per span: this includes computing the span endpoints and computing interpolation parameters for pixel generation
- operations performed once per pixel: this is where the real work needed to compute each pixel is done

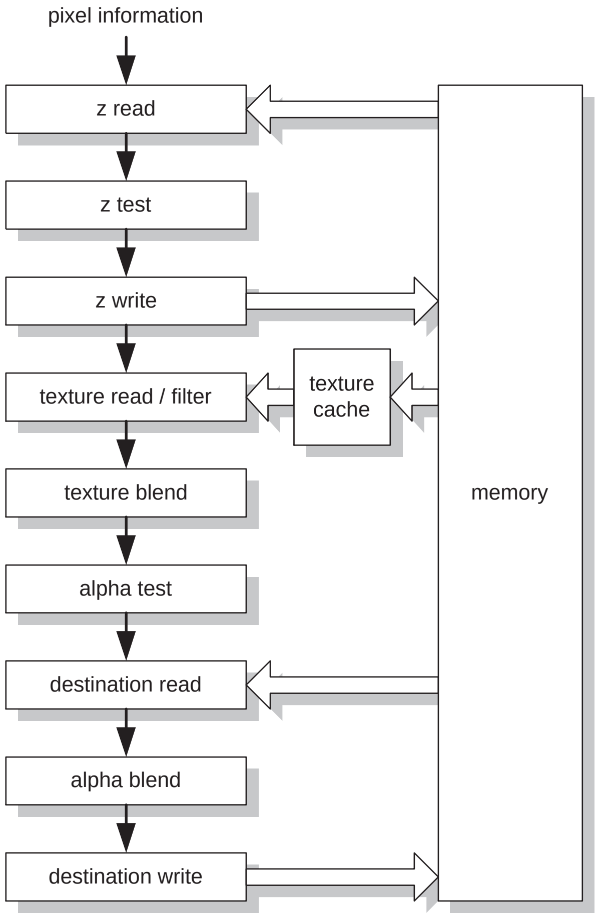


Figure 3: The pixel rasterization pipeline.

The pixel rasterization pipe is shown in Figure 3. The first two stages read the Z value from memory and compare it to the pixel's Z value. If the pixel fails the Z test, i.e., if it is behind a previously drawn pixel, then it is dropped from the pipe. If it passes, the new Z value is written to memory. The next two stages read one or more texels from the texture cache, filter them to produce a single texel, and blend the texel with the pixel color. Next, the alpha value is tested and completely transparent pixels are dropped. Finally, the destination pixel is read, alpha blended with the blended pixel/texture color, and written back to memory.

Several of these pipe stages are optional, depending on mode settings for the current triangle. For example, if the triangle is opaque (no alpha value), the alpha test, destination read, and alpha blend stages are unneeded. A real implementation of this pipeline might look somewhat different, e.g., some of the texture computation might be done in parallel with the Z read because of memory latency.

Assuming that sufficient memory bandwidth is available, this pipeline has a peak throughput of one pixel per clock.

2.3. Previous work

There are many different approaches to high-performance 3D rendering hardware; we list a few here.

Permedia 3 is a single-chip Direct3D/OpenGL rasterizer for high-end PCs [1].

The Neon chip contains eight pixel pipes that interface to eight off-chip SDRAM channels [8]. Several clever techniques are used to maximize memory efficiency.

Sony's second-generation PlayStation chipset uses a MIPS processor with specialized vector units (the Emotion Engine) attached to a 16-pipe rasterizer with embedded DRAM (the Graphics Synthesizer) [2].

The SGI InfiniteReality uses 12 different ASICs on a geometry board, multiple rasterizer boards, and a display generator board [10]. Pixel computations are performed by up to 320 image engines.

The PixelFlow architecture consists of a series of "flow units", each of which has a geometry processor and a SIMD rasterizer board with embedded DRAM ("logic-enhanced memory") [3]. Each flow unit is responsible for a fraction of the primitives; the outputs of all of the flow units are composited using the Z values.

The Imagine architecture is a VLIW media processor with support for a unique "stream processing" model [11].

3. Out-of-order superscalar rendering engine

3.1. Graphics parallelism

The application program, through the device drivers, provides a stream of graphics primitives to the graphics subsystem. These primitives can include 2D operations like block fills, lines, or block moves as well as 3D operations such as triangles and 3D lines. There may be some available parallelism within each primitive, but as 3D applications become more sophisticated, they include a larger number of smaller triangles, each of which has minimal opportunity for parallelism. In current applications (see Section 4.3), triangles are typically hundreds of pixels. As this number decreases, the per-triangle setup time will become a larger and larger component of overall run-time. The best mechanism for increasing performance is to execute multiple graphics primitives concurrently.

The transform, lighting, and clipping operations (see Figure 1) are massively parallel, and their latency is relatively unimportant. Therefore they can be implemented at arbitrarily high performance levels through the use of long pipelines or multiple parallel pipelines. The interesting parallelism problems exist at the rasterization stage where dependences exist. An out-of-order dispatch engine that detects dependences and maintains sequential semantics is

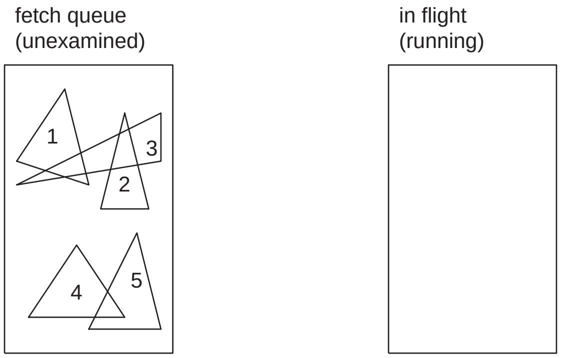


Figure 4: A stream of primitives awaiting dispatch.

used in the rasterizer to extract ILP (where each 'instruction' is a graphics primitive) from the graphics primitive stream. Multiple rasterization engines are used to provide execution resources for the graphics primitives. Since primitives can have long execution times, from several cycles to several hundred thousand cycles, it is not very important to dispatch more than one primitive per cycle, rather it is important to have many primitives executing concurrently. Also, since the primitives do many memory accesses and require an indeterminate number of cycles to execute, it is best to have the rasterization engines run independently, MIMD-style.

3.2. In-order vs. out-of-order dispatch

The basic idea is shown in Figures 4, 5, and 6. The host computer is submitting a sequential stream of graphics primitives to the accelerator (Figure 4). In the simplest implementation, primitives are dispatched from the head of the input list when they are ready. With in-order dispatch (Figure 5), two primitives have been dispatched to execution engines. These primitives are either being executed or they are in some queue approved for immediate execution but waiting for resources. We don't know or care in what order these primitives will begin execution or complete execution. The third primitive cannot be approved for execution since it overlaps with the currently executing primitives. It will be dispatched when primitives 1 and 2 both complete.

In order to extract additional parallelism, it is preferable to allow out of-of-order dispatch [6,7]. In this case (Figure 6), stalled operations are placed into central reservation stations and other operations that can proceed independently are allowed to bypass them. In the actual implementation, all incoming primitives are placed in reservation stations and the oldest eligible candidate is dispatched each cycle.

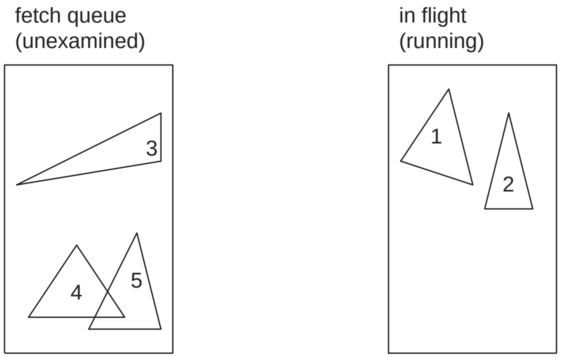


Figure 5: The stream of Figure 4 after in-order dispatch. Primitive 3 is stalled because it overlaps 1 and 2. Primitives 4 and 5 are stalled because they arrived after 3.

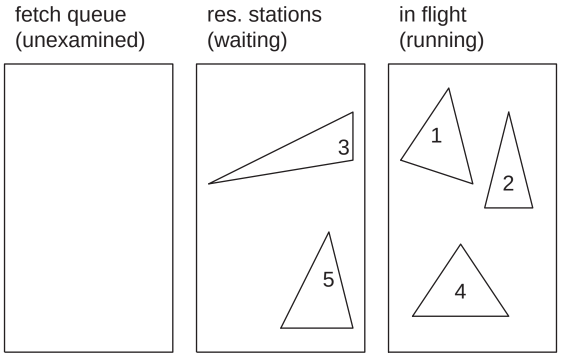


Figure 6: The stream of Figure 4 after out-of-order dispatch. Primitive 4 has been dispatched ahead of primitive 3.

3.3. Dependence checking

In order to implement an out-of-order dispatch engine, mechanisms are required for detecting dependences, for detecting ready operations, and for tracking completion of operations. In processors, dependence checking is accomplished by comparing the source and destination register addresses of a candidate operation to those of previous operations in the sequential instruction stream. A corollary exists for graphics primitives. Any given graphics primitive has a set of destination pixels that it will write and may have one or more sets of source pixels that are read for the operation. In a typical 3D triangle operation, the destination pixels are the ones drawn by the rasterization operation and the source pixels may include the old pixels in those locations that will be alpha-blended by the triangle and the texture map that will be used to draw the triangle. Unlike in a processor, a single register number cannot describe the source or destination operands and it would be

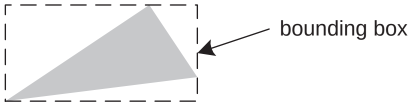


Figure 7: A triangle's bounding box.

impractical to construct lists of source and destination pixels in hardware and compare them for dependences.

We resolve this problem by using a conservative, but simple-to-implement algorithm for detecting dependences. Every dispatched primitive reserves a region of the frame buffer corresponding to the bounding box surrounding each of its operands. The sets of pixels that each primitive *reads* in order to complete are called its *source operands* and the bounding boxes surrounding each of the source operands are called the *source regions*. The set of pixels that each primitive *writes* is called its *destination operand* and the bounding box around it is the *destination region*. Bounding boxes can be computed by taking the minimum and maximum values of the x and y coordinates of each vertex. See Figure 7.

Four numbers, the coordinates of the lower left and upper right corners of the bounding box, can describe each region. For two regions $R_1 = [(a_{\min},b_{\min}),(a_{\max},b_{\max})]$ and $R_2 = [(x_{\min},y_{\min}),(x_{\max},y_{\max})]$, we can determine if the regions overlap by defining an overlap function:

$$\begin{aligned} R_1 \cap R_2 &= (a_{\min} \le x_{\max})(a_{\max} \ge x_{\min}) \\ &\ (b_{\min} \le y_{\max})(b_{\max} \ge y_{\min}) \end{aligned}$$

Graphics primitives can have several source regions as well as a destination regions, but for simplicity consider the case where there is one of each. To determine whether or not a candidate primitive *P* depends on a previously dispatched primitive *D*, the function depend(*P*,*D*) can be computed. S_P is the source region of *P* and D_P is the destination region of *P*. S_D and D_D are the source and destination regions of D. If *depend*(*P*,*D*) is false then *P* can be dispatched concurrently with *D*:

$$\begin{aligned} &\text{depend}(P, D) \\ &\quad = (S_P \cap D_D) + (D_P \cap D_D) + (D_P \cap S_D) \\ &\quad = (\overline{S_P \cap D_D})(\overline{D_P \cap D_D})(\overline{D_P \cap S_D}) \end{aligned}$$

Basically this means that for any two concurrently dispatched primitives, their source regions can overlap but neither source region can overlap with the destination region of the other primitive and the destination regions of the two primitives cannot overlap. This is a similar restriction to that which would be used to determine whether two instructions could issue concurrently in a processor. The first test prevents any RAW hazards caused by reading a source pixel prior to it being written by the earlier primitive. The second test prevents any WAW hazards caused by writing a pixel from the second primitive prior to writing the same pixel from the first. The third test prevents any WAR hazards caused by writing a pixel from the second primitive prior to reading the same pixel from the first. The WAR hazard is somewhat unique in that it would be prevented in any well constructed processor pipeline by reading the source operands of the first instruction from the register file early in the pipeline and retaining a copy of the correct value until it is consumed in the pipeline. It is impractical to copy the values of all of the source pixels for a graphics primitive, so the source region in memory must be reserved until an operation completes, and the WAR test must be performed against any subsequently issuing primitives.

The hardware required to implement the out-of-order dispatch mechanism is shown in Figures 8 and 9. A fetch unit holds a command queue containing graphics primitives. As reservation stations in the issue unit become available, primitives are passed to the issue unit for dependence testing and scheduling. The issue unit can issue one primitive per cycle to any one of several accelerators for rasterization. As each primitive is completed, the accelerator notifies the issue unit so that the primitive can be retired and its reservation stations cleared. The accelerators access memory through a global memory interface unit that in practice is likely to contain a large switch and many independent memory banks to provide adequate bandwidth. For our experiments, we have assumed an ideal multi-ported memory.

The reservation mechanism uses three sets of registers to store data about pending and executing primitives. The destination reservation station indexes are the primary identifiers for a new primitive. It is assumed that every primitive has exactly one destination region and the bounding box for that region is stored in the destination reservation station for that primitive. A primitive may also have zero, one, or two source regions. Source reservation stations are allocated in a separate register file for each source region. The source register file contains a bounding box descriptor for each source region and a tag that is the index of the destination reservation station corresponding to each source. The candidate buffers hold relevant information about unissued primitives. This includes the opcode and additional operands for the primitive (such as vertex colors, alpha values, etc.) and a dependence vector that describes the restrictions on issue. A pointer to the opcode and operand can be included rather than the actual data. Once again, a tag is used to specify the address of the destination reservation station of the primitive. Every register has a valid bit that is used to indicate that the register contains live data. Ideally, the number of destination reservation

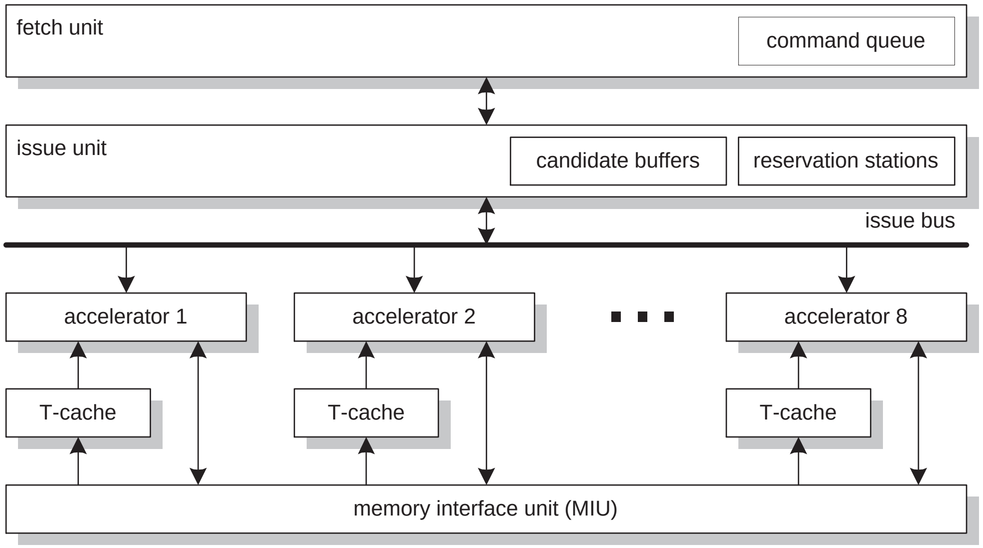


Figure 8: Top-level architecture of the superscalar 3D rendering engine.

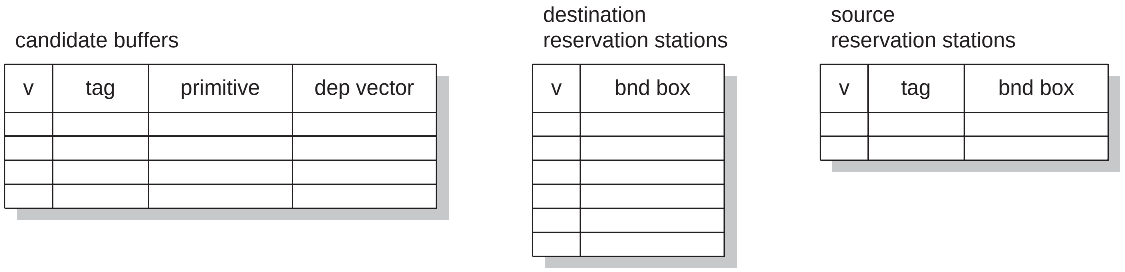


Figure 9: Registers used in the superscalar rendering engine.

stations should equal the number of candidate buffers plus the number of accelerators.

A primitive is moved into the issue unit from the fetch unit each cycle provided that there is an empty destination reservation station, an empty candidate buffer, and enough empty source reservation stations. Hardware in the fetch unit has already decoded the primitive type and computed the bounding box for each source and destination region. These bounding boxes are stored in the source and destination reservation stations and the index of the destination reservation station is stored in the tag field of the source reservation stations and the candidate buffer. A dependence vector is computed using a variation of the *depend* function extended to support two source regions. The hardware to compute this dependence vector is shown in Figure 10. The bounding box coordinates (four numbers) of the new primitive are driven onto vertical buses for each operand. The bounding box coordinates for each valid reservation station are driven on horizontal buses. At the intersections that correspond to potential hazards, bounding box overlap comparators implement the overlap function described earlier. A dependence vector is thus calculated by computing whether or not the new primitive overlaps with each previously tested primitive from earlier in the command sequence. Bit *k* in the dependence vector is set if the new primitive must wait for the primitive stored in destination reservation station *k* to complete. This dependence vector is stored in the candidate buffer reserved for the new primitive.

At the same time, the issue unit is testing the existing issue candidates to see if any are ready to be issued. If any of the accelerators are available, then the issue unit tests all of the dependence vectors in the candidate buffers. If any valid candidate buffer contains a dependence vector of all zeros, then it can issue this cycle. If more than one candidate can issue, one is selected at random. The candidate is issued by transferring the primitive data and the tag to an accelerator and clearing the valid bit to free the candidate buffer.

Finally, any accelerators that complete a primitive in a given cycle return the tag of the completed primitive to the issue unit. The issue unit clears the valid bit from the corresponding destination reservation station. It also passes the tag to all of the source reservation stations, which compare it to their own tag and clear their valid bit if the tags match. Finally, the tag is decoded and the corresponding

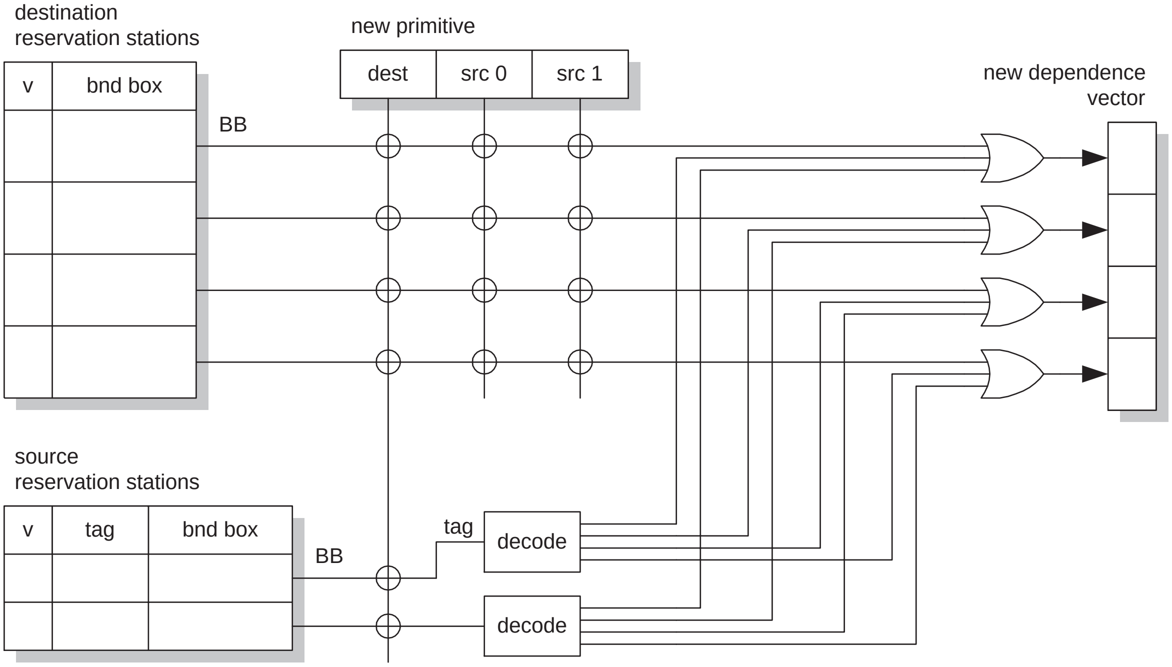


Figure 10: Dependence-checking hardware. Each small circle is a bounding box overlap comparator.

bit in the dependence vector of each candidate buffer is cleared. This removes the completed primitive from the reservation stations and clears any dependences in any pending primitives. Once all of the dependence bits in a candidate's dependence vector are cleared, that candidate can issue on a subsequent cycle.

The logic required to implement the issue unit is quite modest. The issue logic for a system with 8 accelerators and 32 candidate buffers could be built with well under 100k gates, including storage.

4. Experimental results

4.1. Modeling

We have built a cycle-accurate simulator for the superscalar rendering engine. An internally-developed Direct3D logging tool is used to generate frame traces. These traces contain triangles (vertex coordinates, vertex colors, texture coordinates), mode information (texture blending mode, alpha blending mode, etc.), and the textures. The simulator reads a parameter file describing the rendering engine configuration, then processes a trace and writes a summary of utilization and throughput. The simulator also writes the rendered frame so that its functionality can be verified.

The current simulator has a few limitations:

- It only handles 3D triangles. Direct3D also supports 3D lines and points, which are not handled by the simulator. In addition, some games use DirectDraw to render 2D primitives (e.g., scores, counters, instrument panels)
- It assumes a perfect memory system. Each accelerator uses a peak of 128 bits per clock (32-bit Z read, 32-bit Z write, 32-bit destination read, 32-bit destination write, not counting textures) of memory bandwidth. We assume that a banked memory system can sustain this bandwidth with appropriate queueing to deal with the latencies.

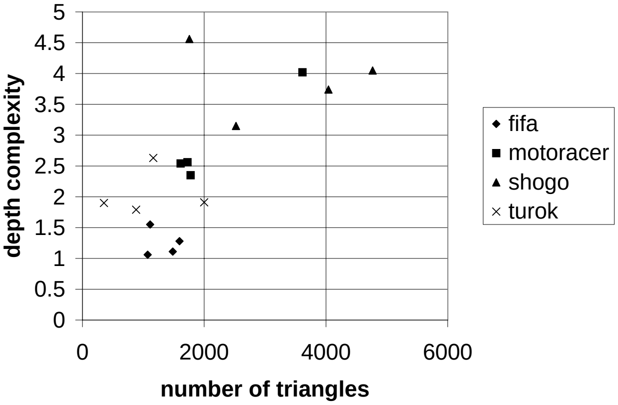
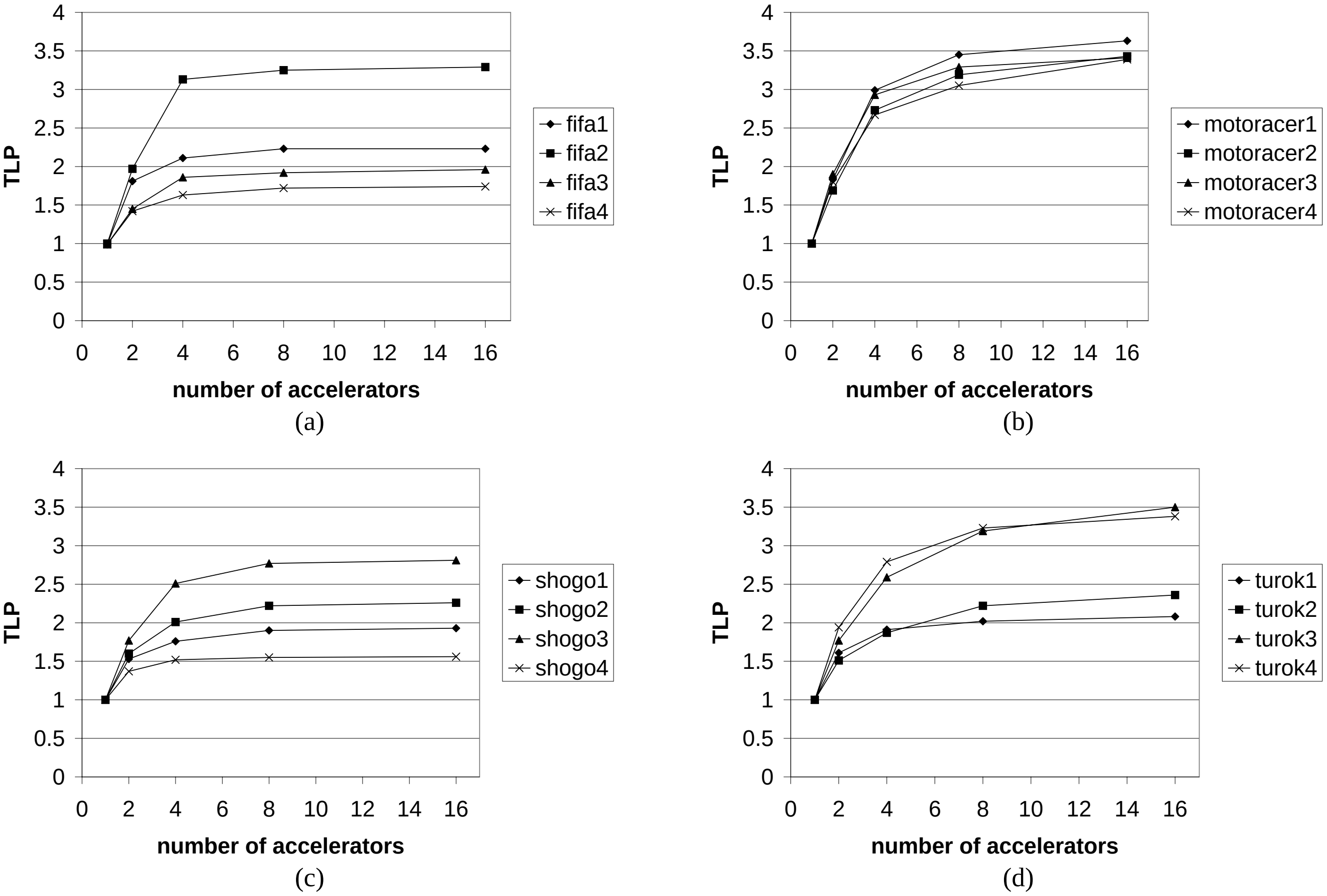


Figure 11: Triangle counts and depth complexities for the sixteen test frames.

A 3D engine's memory system is critical. We plan to add a cycle-accurate memory model to this simulator to allow experimentation with different memory configurations. It is common, even in today's PCs, for graphics controllers to have very high performance memory systems.



(a)

(b)

(c)

(d)

Figure 12: Triangle-level parallelism (accelerator utilization) versus number of accelerators for the four test frames from each of the four games – (a) FIFA 99, (b) MotoRacer 2, (c) Shogo, and (d) Turok.

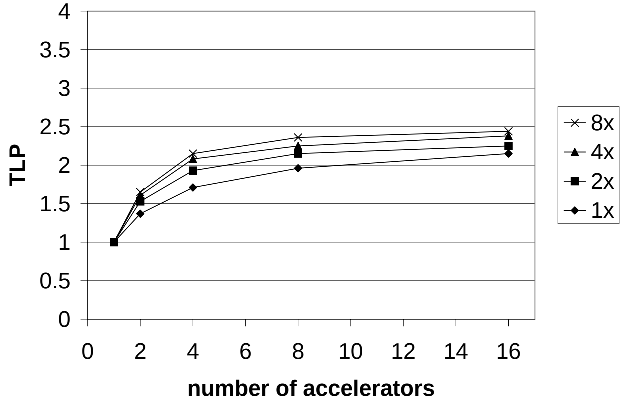


Figure 13: Triangle-level parallelism versus number of accelerators for different numbers of candidate buffers (issue window size). The number of candidate buffers is a multiple (1x, 2x, 4x, 8x) of the number of accelerators. Average TLP across all 16 test frames is shown.

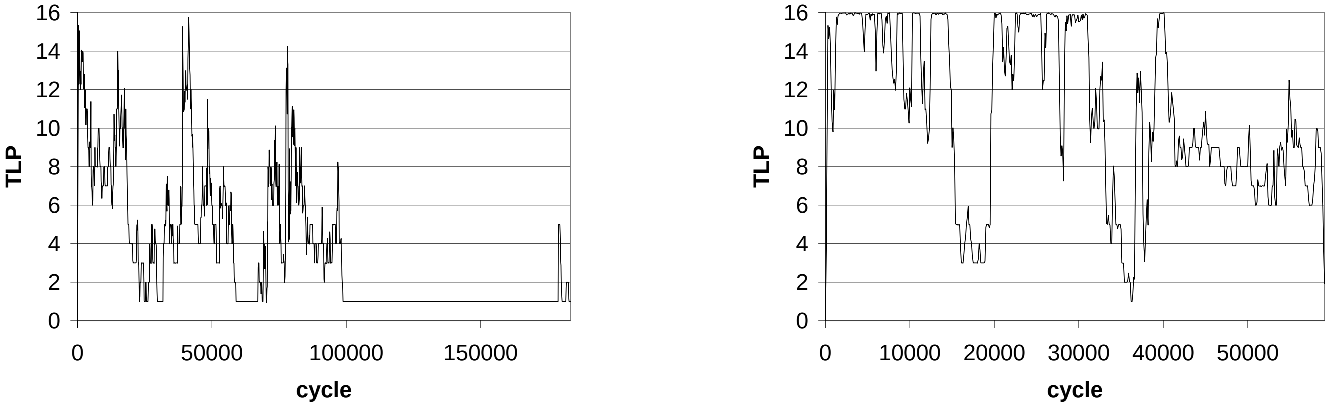
4.2. Performance metrics

The primary performance metric used in this paper is triangle-level parallelism (TLP). Because the accelerators run at a peak of one pixel per clock and triangles generally consist of hundreds of pixels, the issue rate is relatively low (much less than one). So, instead of issue rate, TLP is used. TLP is simply the accelerator utilization, i.e., the number of busy accelerators at any given cycle. An accelerator is considered busy if any of its pipeline stages are busy.

4.3. Benchmarks

As test cases, we used four arbitrarily selected frames from each of four Direct3D games:

- FIFA 99: soccer simulation (Electronic Arts) [the traces are labeled fifa1, fifa2, fifa3, and fifa4]
- MotoRacer 2: motorcycle racing (Delphine Software / Electronic Arts) [motoracer1 - motoracer4]
- Shogo: Mobile Armor Division: first-person shooter (Monolith Productions) [shogo1 - shogo4]
- Turok 2: Seeds of Evil: first-person shooter (Acclaim Entertainment) [turok1 - turok4]



(a)

(b)

Figure 14: Triangle-level parallelism over time for the turok3 trace with 16 accelerators, without (a) and with (b) triangle splitting. Each point represents the average parallelism over 100 cycles.

To give an idea of the nature of these sixteen frames, Figure 11 shows the triangle count and depth complexity for each one. Depth complexity is the number of pixels produced by the pipeline divided by the total number of pixels in the screen (all tests here used 640x480 screen resolution). In other words, depth complexity is the average number of triangles covering any given pixel. The frames in our tests range from 1000 to 5000 triangles with depth complexities of 1 to 5. Equivalently, the total number of processed pixels per frame (depth complexity x 640 x 480) is 0.3 to 1.5 million.

4.4. Results

The first set of results (Figure 12) shows the increase in triangle-level parallelism for each of the sixteen frames as the number of accelerators is increased. For these simulations, the number of candidate buffers was set to eight times the number of accelerators, with the goal of measuring available parallelism, unrestricted by a small issue window. In all simulations in this paper, the number of destination reservation stations is set to the sum of the number of accelerators and the number of candidate buffers. Note that these curves are very similar to ILP curves for a superscalar processor. As the number of accelerators is increased, the TLP initially increases almost linearly. But after a certain point, triangle dependences come into play and limit the available parallelism, causing the curves to flatten out. (Rendering a triangle is a coarser-grain operation than a typical microprocessor instruction. In this sense, TLP can be considered more similar to thread parallelism than to ILP.)

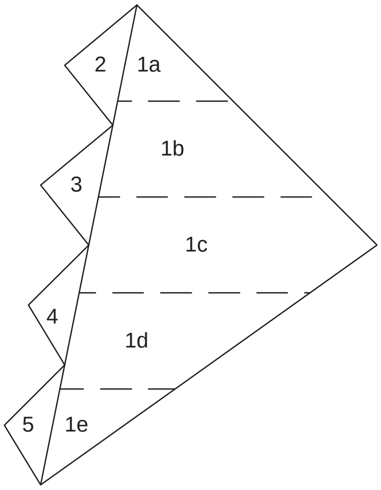


Figure 15: An example of triangle splitting. If triangle 1 is not split, triangles 2-5 are dependent on it, and must wait until it is completely rendered. When triangle 1 is split into slices (1a - 1e), the available parallelism is significantly higher. Here, triangles 3, 4, and 5 are independent of slice 1a; triangles 4 and 5 are independent of 1b; etc.

We also varied the number of candidate buffers as a multiple of the number of accelerators. With one accelerator, we ran simulations with 1, 2, 4, and 8 candidate buffers; with two accelerators, we used 2, 4, 8, and 16 candidate buffers, etc. Figure 13 shows the results, as an average across all sixteen test frames. (The average was weighted by the number of cycles used by each frame.) Increasing the number of candidate buffers gives the issue logic a larger window of triangles to examine. These curves are analogous to those produced by varying the issue window size of a superscalar processor.

4.5. Large triangles

Amdahl’s Law applies to triangles just as it does to instructions. When a very large triangle is encountered, the

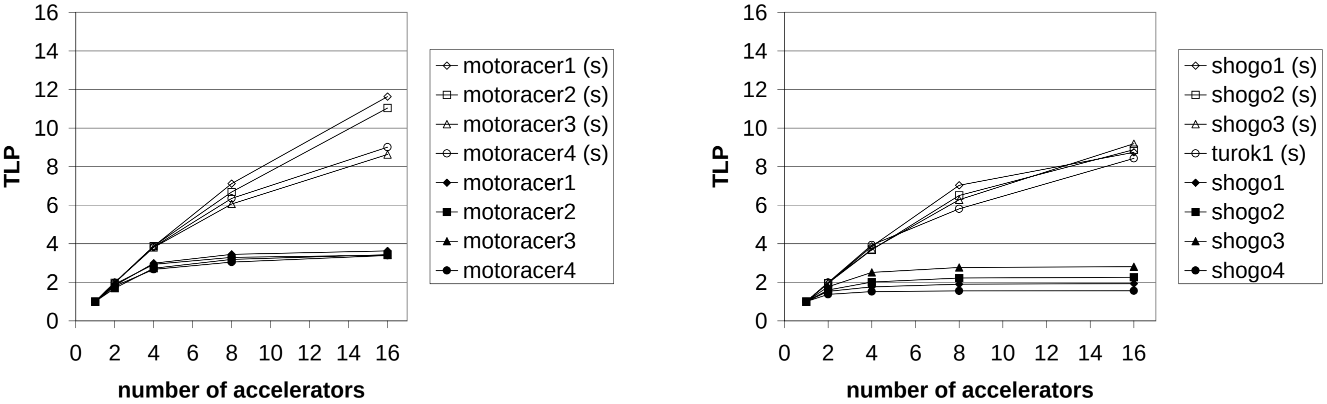


Figure 16: The Motoracer and Shogo frames, with (denoted by '(s)') and without triangle splitting.

pipeline is essentially serialized. Subsequent triangles cannot issue because most or all of them overlap the large triangle. The candidate buffers quickly fill with dependent triangles, and issue is stalled until the large triangle completes. The result is that average triangle-level parallelism is limited by large triangles, just as instruction-level parallelism is limited by serial sections of code. Figure 14a shows a typical graph of accelerator usage over time. In this particular case (the turok3 trace, with 16 accelerators), two very large triangles (92,000 pixels in total) are used to simulate a foggy sky. As shown in the figure, the TLP is reduced to 1 for about 80,000 cycles.

To solve this problem, large triangles can be split into smaller slices. See Figure 15. The slices can be rendered in parallel with each other and with other triangles. Figure 14b shows the same trace, this time with all triangles greater than 32 pixels tall being split into 32-pixel-tall slices at the fetch stage. Note that the TLP is clearly higher, and the long serial portion is gone. In this case, the overall average TLP increased from 3.50 to 10.90 – a speedup of 3.1. Figure 16 shows the results for the Motoracer and Shogo frames (compare this to Figure 12(d)). (Note that comparing the TLP with and without splitting is not strictly fair since the splitting creates new triangles. In the scenes used here, triangle setup time is negligible compared to pixel rasterization time. However, as triangles get smaller, this will become a more important issue.) In this work, we only split triangles vertically. For even better results, a more complex system could choose to split horizontally, vertically, or both, depending on the shape of the triangle.

4.6. Performance enhancements

The TLP exploited with 16 accelerators in the initial experiments averages around 2.5 concurrent primitives, and ranges from 1.5 to 4 across the various scenes. There are two identified factors that limit the available parallelism. One of these was the problem with large triangles that was addressed by decomposing these large objects. This improved the performance on the Turok benchmarks by up to 300%. The other limiting factor for parallelism is the existence of long dependence chains. These occur naturally from the practice of tessellating surfaces into strips, meshes, and fans. Figure 17 shows an example of a triangle strip. The bounding box of each triangle overlaps the bounding box of the adjacent triangles so when the primitives are issued they produce a single long dependence chain. In fact, many of these triangles should not be dependent on each other. The dependence chain is an artifact of the bounding box approximations. It is impractical to do a truly accurate overlap test, but we have developed an improvement to the algorithm. For non-alpha blended triangles, the Z-buffer will resolve ordering problems even if two triangles overlap as long as they are not issued concurrently. The vast majority of the triangles are of this type and they can be marked as independent triangles. We have augmented the algorithm to add a conflict vector to the dependence vector that is used to compare currently issued triangles to independent candidates. This allows non-overlapping triangles to issue concurrently without ordering constraints caused by dependence chains. For example, in Figure 17, triangles 1, 3, and 5 can issue in parallel, despite the 1←2←3←4←5 dependence chain.

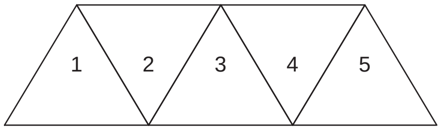


Figure 17: A triangle strip. Note that each triangle's bounding box overlaps its immediate neighbors (left and right) – 3 overlaps 2 and 4 but not 1 or 5.

Figure 18 shows the results for Motoracer and Shogo, with both splitting and dependence chain breaking. Note that the system achieves almost perfect utilization on Motoracer, while breaking improves the Shogo TLP numbers by only 5-10% with a slight slowdown in one case.

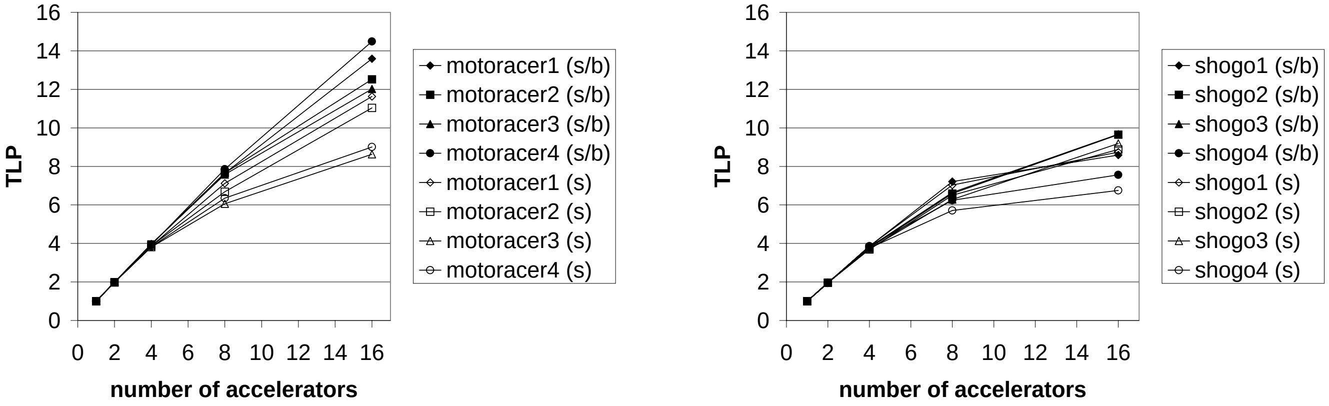


Figure 18: The Motoracer and Shogo frames, with splitting (s) and with both splitting and dependence chain breaking (s/b).

(The Turok results fall in between these two; the FIFA frames do not improve at all with breaking.) We also applied dependence chain breaking without triangle splitting. In this case, the TLP increased by 0 (FIFA) to 95% (motoracer4). We expect that the impact of dependence chain breaking will be larger for typical CAD applications than for the games used here.

5. Conclusion

Instruction-level parallelism has become a commonplace feature of general-purpose processor microarchitectures but has been slow to impact special-purpose computing. We have presented a model for using out-of-order superscalar principles in a parallel graphics accelerator. We have explained the importance of retaining sequential semantics while executing multiple primitives concurrently. We have also described an algorithm and hardware implementation for implementing out-of-order operations in a graphics system. The similarities with processor mechanisms are apparent, but subtle differences such as the management of source operands and the complexity of comparisons require some new mechanisms. Experimental results show 1.5-3.6X speedups on real applications using a simple model, similar to the results from many integer benchmarks on superscalar processors. Enhanced techniques specific to 3D graphics for decomposing large triangles and breaking false dependence chains increase performance to more than 10x a sequential system.

This is a new application of ILP principles, but it shows great promise. We believe that there is great room for continued research in this type of architecture and in applying ILP principles to other special-purpose computing platforms.

References

[1] 3Dlabs. “Permedia 3: Inside the Third Generation of Desktop 3D.” *Proceedings of Hot Chips 10*, 1998, 233-243.

[2] Diefendorff, Keith. “Sony’s Emotionally Charged Chip.” *Microprocessor Report*, April 19, 1999, 1,6-11.

[3] Eyles, John et al. “PixelFlow: The Realization.” *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1997, 57-68.

[4] Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*, 2nd ed. in C. Addison-Wesley, 1996.

[5] Glaskowsky, Peter N. “A Concise Review of 3D Technology.” *Microprocessor Report*, June 21, 1999, 14-18.

[6] Hennessy, John L. and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[7] Johnson, Mike. Superscalar Microprocessor Design. Prentice Hall, 1991.

[8] McCormack, Joel et al. “Implementing Neon: A 256-Bit Graphics Accelerator.” *IEEE Micro*, March-April 1999, 58-69.

[9] Microsoft. *Direct3D Immediate Mode*. From the DirectX 6.1 SDK.

[10] Montrym, John S. et al. “InfiniteReality: A Real-Time Graphics System.” In Proceedings of SIGGRAPH, 1997, 293-301.

[11] Rixner, Scott at al. “A Bandwidth-Efficient Architecture for Media Processing.” *Proceedings of MICRO-31*, 1998, 3-13.