Exploiting Parallelism in Geometry Processing with General Purpose Processors and Floating-Point SIMD Instructions

Chia-Lin Yang,[1] Barton Sano,[2] and Alvin R. Lebeck[1]

[1]Department of Computer Science
Duke University
Durham, North Carolina 27708 USA
{yangc, alvy}@cs.duke.edu

[2]Western Research Lab
Compaq Computer Corporation
sano@pa.dec.com

Abstract

Three dimensional (3D) graphics applications have become very important workloads running on today's computer systems. A cost-effective graphics solution is to perform geometry processing of 3D graphics on the host CPU and have specialized hardware handle the rendering task. In this paper, we analyze microarchitecture and SIMD instruction set enhancements to a RISC superscalar processor for exploiting parallelism in geometry processing for 3D computer graphics.

Our results show that 3D geometry processing has inherent parallelism. Adding SIMD operations improves performance from 8% to 28% on a 4-issue dynamically scheduled processor that can issue at most 2 floating-point operations. In comparison, an 8-issue processor, ignoring cycle time effects, can achieve 20% to 60% performance improvement over a 4-issue. If processor cycle time scales with the number of ports to the register file, then doubling only the floating-point issue width of a 4-issue processor with SIMD instructions gives the best performance among the architectural configurations that we examine (the most aggressive configuration is an 8-issue processor with SIMD instructions).

Index terms: 3D graphics, geometry pipeline, superscalar processors, SIMD instructions, paired-single instructions

This work supported in part by NSF CAREER Award MIP-97-02547, DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637 and CDA-95-12356, Duke University, and an equipment donation through Intel corporation's Technology for Education 2000 Program. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government

1 Introduction

The increasing number of multi-media applications produces a commensurate increase in demand for cost-effective multi-media processing [11]. Traditionally, media processing was implemented in expensive custom hardware specialized for specific applications (e.g., speech, video, and graphics). Advances in conventional microprocessor design now permit offloading some functionality to a general-purpose processor, possibly sacrificing performance in return for reduced cost. The key is to minimize this performance degradation, potentially by adding architectural support for media processing.

Many current microprocessors have Single Instruction Multiple Data (SIMD) type instructions to accelerate audio, video and 2D image processing, such as Intel MMX [16], Sun UltraSPARC VIS [10] and HP PA-RISC [8]. This type of SIMD instruction operates only on integer data. Today, several processor vendors, such MIPS Technology Inc. [15], Cyrix, IDT, AMD [2], Intel [7], and Motorola [19] are in various stages of incorporating floating-point SIMD instructions to speedup geometry processing for three dimensional (3D) graphics.

Typically, 3D graphics processing is a 3-stage pipeline [5]: 1) database traversal, 2) geometry computation, and 3) rasterization. Display models representing graphics scenes are generally stored in a database that must be traversed (stage 1) to extract the appropriate information for display, such as the drawing primitive (e.g., line or triangle), lighting models, etc. The information is then passed to the geometry subsystem (stage 2), which is responsible for transforming 3D coordinates to 2D coordinates. Finally, the rasterization stage (stage 3) converts transformed primitives into pixel values and stores them in the frame buffer for display.

In high-end graphics systems [17][18], the host CPU is only responsible for database traversal, and custom hardware is used for geometry processing and rasterization. The cost of building these high-end systems is generally too high for the mass market. To reduce cost, the host CPU could execute some, or all, of the graphics pipeline. This paper focuses specifically on host CPU execution of geometry computation using a single dynamically scheduled superscalar microprocessor.

Geometry computation is floating-point intensive. Vertex coordinates, color and transformation matrices are stored in single-precision floating-point format. Previous studies [1] have shown that 90 floating-point arithmetic operations are required to process a single vertex. Current superscalar processors can issue 2 floating-point operations per cycle. The above analysis implies that a 500 MHz processor could theoretically process 11 million vertices per second. This value is close to the computing capability of today's specialized hardware [18]. However, because of instruction scheduling and resource limitations, a general purpose processor is unlikely to achieve this theoretical rate. The goal of this paper is to examine the performance of geometry processing on a general purpose processor and evaluate the benefits of recently proposed instruction set enhancements.

Geometry processing is an inherently parallel task, since each object vertex can be processed independently. Dynamically scheduled processors can exploit this parallelism by looking ahead in the instruction stream to identify and execute the operations associated with different vertices. Recall that vertex computations require only 32-bit floating-point values. Since most modern microprocessors have 64-bit floating-point registers, geometry calculations using 32-bit operands are utilizing only half the floating -point datapath (registers, functional units, and busses). Another way to exploit this parallelism is using SIMD type instructions to perform operations on multiple vertices in one instruction—called paired-single instructions. Paired-single instructions fully utilize the 64-bit datapath by performing two independent 32-bit operations, each using half the datapath.

As mentioned above, most processor vendors are incorporating paired-single instructions. AMD's 3DNow! Technology [2] is currently available. However, we are unaware of any published quantitative evaluation of their performance using full graphics applications. Parthasarathy et al. performs detailed performance evaluation for Sun VIS instruction set but they focus only on image and video processing applications [23]. In this paper, we simulate Viewperf [20], an industry standard benchmark suite, on an out-of-order superscalar processor both with and without paired-single instructions. We modified the geometry computation routines in MESA [13] (a public domain implementation of OpenGL [24]) to utilize paired-single instructions. We first analyze the effects of increasing the resources available in a conventional processor. This is followed by a comparison to paired-single execution, both with and without clock cycle time effects.

The contributions of this paper are as follows:

1. Although geometry processing presents substantial parallelism, we discover that certain aspects of application implementations can significantly impact the available parallelism that can be exploited by a superscalar processor. In the best case, an 8-way issue processor can achieve 60% performance improvement over a 4-way with a 64-entry dispatch queue and 128 registers, but for certain benchmarks, the performance only increases by 20%. Furthermore, if the CPU cycle time scales with the number of ports to the register file, the performance improvement is less than 5% for all the benchmarks.
2. We analyze the effect of adding paired-single instructions on a set of industry standard 3D graphics benchmarks instead of small kernels. We found that the performance improvement from pairing up single-precision floating-point operations ranges from 8% to 28% on a 4-way issue processor that can issue at most 2 floating-point operations per cycle.
3. We quantify the benefits of paired-single instructions over increasing only the floating-point issue width in superscalar processors. Our results indicate that adding paired-single instructions to 4-issue processor performs within 7% of doubling the floating-point issue width. For certain bench-

marks, the former even outperforms the latter. The performance advantage of paired-single instructions increases when considering the clock cycle time effect.

The remainder of this paper is organized as follows. Section 2 provides background information on geometry computation, and presents benchmark characteristics. We review paired-single instructions in Section 3. Section 4 presents our simulation infrastructure, and Section 5 presents our simulation results. In Section 6, we compare the performance achieved by a general processor v.s. a high-end graphics system. Finally, Section 7 concludes the paper.

2 Background

To understand the architectural aspects of geometry processing, we first describe the six stages of the 3D geometry pipeline. Then we characterize a set of OpenGL performance evaluation benchmarks (Viewperf [20]).

2.1 The Geometry Pipeline

3D applications usually use polygonal primitives (e.g., triangles) to represent objects in an application database. These primitives are represented in their own coordinate space. How those objects are displayed on the screen is determined by the following factors: the positions and orientations of objects in the scene, the viewpoint, the surface properties of objects, and the light sources. Geometry processing transforms primitives from the object coordinates to the screen coordinates, and calculates the color for each vertex according to the object surface and light properties. Geometry processing only operates on vertices. The rasterization stage takes those transformed vertices and fills in the interiors of polygons.

Similar to the overall 3D computation, geometry processing can be divided into a set of pipeline stages. In a typical geometry pipeline, there are six stages as shown in Figure 1:

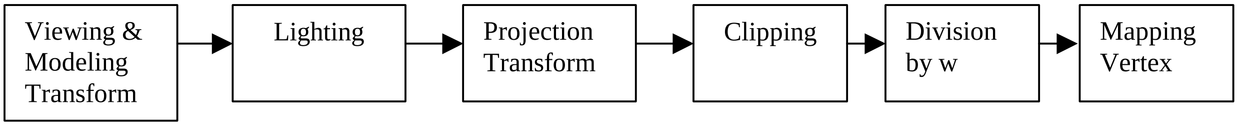


Figure 1: 3D geometry pipeline.

View and model transformation: graphics primitives (e.g., line, triangle or polygon) are transformed to the viewer's frame of reference. Transformations involve vector matrix multiplication on either 1×4, 4×4 or 1×3, 3×3 vector and matrix sizes.

Lighting: the light position, color and material properties are used to calculate the object color.

Projection transformation: this stage determines how objects are projected to the screen. This again requires multiplication of a 1×4 vector and a 4×4 matrix.

Clipping: objects are clipped to the viewable area to avoid unnecessary rendering.

Division by w: the x, y, z components of each vertex are divided by its w component. Geometry processing usually works in the homogenous coordinate system, where all the vertices are represented with four coordinates (x, y, z, w). With coordinate positions expressed in the homogenous form, the transformations (i.e., viewing, modeling and projection transformation) can be simplified and performed as matrix multiplications [6]. The w component is initially set to one. After applying the projection transformation, it may not equal to one. We then perform the division to get (x/w, y/w, z/w), which are the Cartesian coordinates of the homogeneous point.

Mapping vertex coordinates to screen coordinates: vertices are mapped to the screen coordinates.

Note that lighting (stage 2) is optional. For those applications that only perform wireframe rendering or implement a global illumination algorithm (i.e., the color of each vertex is precomputed), the lighting stage is unnecessary. However, the other 5 stages are mandatory.

. The remainder of this section begins this investigation by characterizing a set of 3D applications.

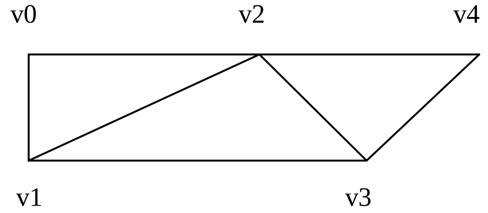
2.2 Benchmark Characterization

To characterize the architectural aspects of 3D applications, we used ATOM’s pixie tool [3] to analyze the Viewperf OpenGL performance evaluation benchmarks [20]. OpenGL is an API for graphics hardware initially defined by Silicon Graphics [26]. We use Mesa [13], a public-domain software implementation of the OpenGL specification, in this study. Mesa contains a complete software implementation of the rendering pipeline, allowing OpenGL applications to execute on machines without specialized graphics hardware.

The Viewperf suite contains five different graphic model sets including CAID (Computer Aided Industrial Design) and digital content creation models. Each set has seven to ten tests using different OpenGL primitives, lighting models and rendering parameters. In this section, we characterize three different aspects of the Viewperf benchmark set: 1) the dynamic instruction distribution, 2) the average number of vertices per glBegin/glEnd pair and 3) the amount of execution time spent in the various geometry pipeline stages.

Dynamic Instruction Distribution

The dynamic instruction distribution of the Viewperf benchmarks (average over the five different benchmarks) indicates that 42.5% of all of the instructions executed by the geometry routines involve single-precision floating- point instructions. A significant amount of integer instructions are needed for executing mode changes (e.g. using different texture file or changing the lighting model). The four most frequently executed instructions are load (13.4%), multiply (12.2%), add (9.7%) and store (6.8%) for single-precision floating-point data. Most of the load instructions come from loading the transform matrices and vertices. Similarly, the store instructions are used to save the processed vertices back to memory. The multiply and add instructions are primarily from the transform and lighting operations.



```
glBegin (GL_TRIANGLE_STRIP)
glVertex3fv(x0,y0,z0); /* coordinates for vertex v0*/
glVertex3fv(x1,y1,z1); /* coordinates for vertex v1*/
glVertex3fv(x2,y2,z2); /* coordinates for vertex v2*/
glVertex3fv(x3,y3,z3); /* coordinates for vertex v3*/
glVertex3fv(x4,y4,z4); /* coordinates for vertex v4*/
glEnd();
```

Figure 2: Example of using GL_TRIANGLE_STRIP primitive.

Average Number of Vertices per glBegin/glEnd Pair

OpenGL implements ten drawing primitives (e.g., GL_LINES, GL_TRIANGLES and GL_POLYGON). To draw an object, a set of vertices are bracketed between a call to glBegin() and glEnd(). The argument passed to glBegin() determines which geometric primitive is constructed from the vertices. 3D surfaces are usually broken down into triangles. The most efficient way for drawing a series of triangles that are connected to each other is using the GL_TRIANGLE_STRIP primitive (as shown in Figure 2). However, some 3D content creation applications do not store objects in a format amenable to this drawing method. In this case, the OpenGL viewing applications may have to invoke a drawing primitive for each triangle. Thus, the number of vertices per glBegin/glEnd will be small. Profiling results show that the average number of vertices per glBegin/glEnd pair varies across the Viewperf benchmark. Awadvs uses the GL_POLYGON primitive and has only 3.4 vertices on average, while some of the CDRS tests use the GL_TRIANGLE_STRIP primitive and have up to 400 vertices per glBegin/glEnd pair.

There are four ways to exploit parallelism in geometry computation: 1) processing individual components of a vertex (e.g., coordinate (x, y, z, w) or color (R, G, B, A)), 2) processing multiple vertices of each primitive within the same pipeline stage, 3) processing vertices of each primitive in different pipeline stages and 4) processing different primitives. In the MESA implementation, the computations for vertices of each primitive (i.e., those vertices bracketed by glBegin/glEnd) in the same pipeline stage are performed in loops. Several internal library routines are executed before starting the next stage or a new set of geometry drawings.

A superscalar processor that can only exploit ILP from instructions stored in the dispatch queue is more likely to exploit the parallelism in the first two scenarios. A small number of vertices between glBegin/glEnd indicates that fewer independent floating-point instructions can be issued close in time. Thus, we do not expect benchmarks with very small number of vertices on average per glBegin/glEnd pair to achieve IPC as high as benchmarks with a large number of vertices, unless a very large dispatch queue is used.

Execution Time Distribution of the Geometry Pipeline

We divide the execution time for the geometry pipeline into five portions:

Light (gl_color_shade_vertices):[1] This portion corresponds to the lighting stage, which calculates the color for each vertex.

XformV (gl_xform_normals_4fv): This portion includes the vertex transformation of both the viewing/modeling and projection transform stages. It performs multiplication of a matrix by a vector.

XformN (gl_xform_normals_3fv): This portion includes the normal vector transformation in the viewing/modeling transform stages.

[1] The corresponding routine name in the Mesa implementation is listed in parenthesis.

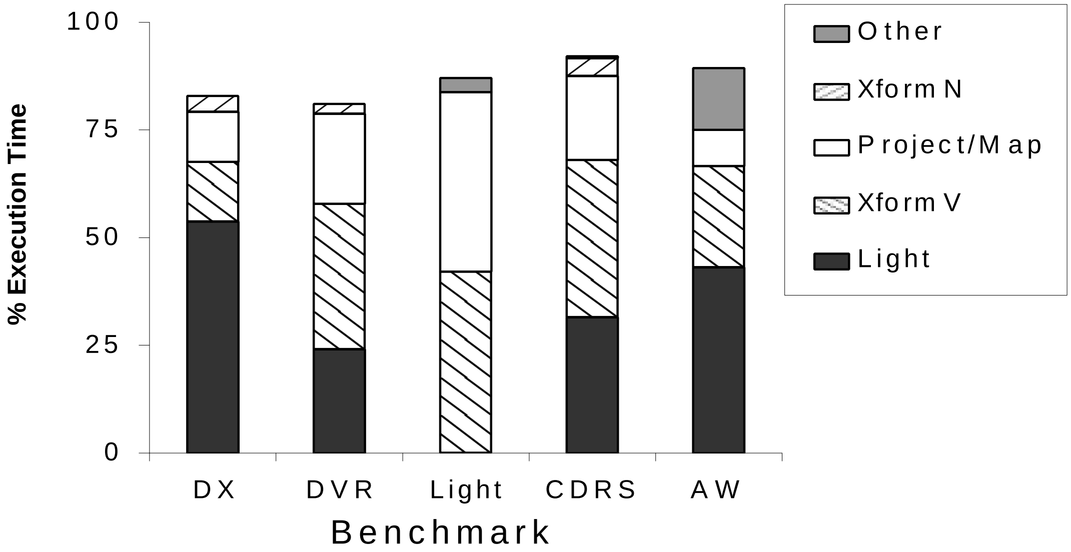


Figure 3– Execution time distribution in the MESA geometry pipeline.

Div by w/Map (gl_transform_vb_part2): This portion includes the computation of div by w and mapping vertex stages. It selects the appropriate lighting routine (e.g. line, polygon, and type of shading) and calls the fog, texture, and clipping routines before finally projecting the primitives to screen coordinates.

Other: This portion includes the clipping stage and the library routines executed between different pipeline stages and drawing primitives.

As shown in Figure 3, Light and XformV are the two portions where geometry processing spends the most time. Note that the Light benchmark gets its name because each vertex color is pre-computed using a global illumination algorithm, therefore, it does not actually execute the lighting functions. Awadvs spends almost 15% of the execution time in the routines executed between different pipeline stages and drawing primitives as indicated by Other in Figure 3. This is significantly higher than the

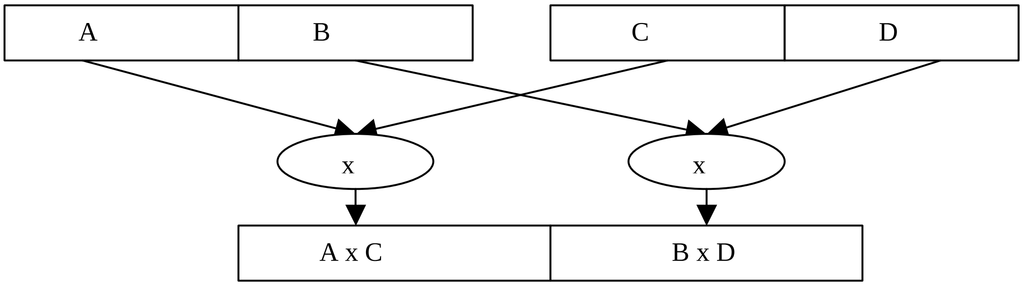


Figure 4: Operation of paired-single multiply.

Instruction Format	Latency(cycle)
LDPS dest, index(base)	2
STPS src, index(base)	2
PMUL src1, src2, dest	4
PADD src1, src2, dest	4
PSUB src1, src2, dest	4
CVT.S.PL/U src, dest	1
CVT.PS.S src1,src2,dest	1
ADD_HL src, dest	4
LDS_HL dest index(base)	2

Table 1: Instruction format and latency. All instructions are fully pipelined.

other benchmarks. Awadvs has very small average number of vertices (3.4) per glBegin/glEnd pair, which implies that switching between pipeline stages and glBegin/glEnd pairs occurs more frequently.

3 SIMD Instruction Extensions

From the benchmark profiling discussed in the previous section, we observe that most of the arithmetic floating-point instructions are multiply and add, and these operations are all performed on single-precision values (32-bit). Thus, the SIMD type instructions that perform multiply or add operations on two single-precision floating -point values could fully utilize the 64-bit floating point registers in current superscalar processors and potentially eliminate a significant number of instructions. The MIPS V ISA Extension [14] proposes adding a new data type called paired-single, which packs two single precision floating-point values into one 64-bit floating-point register. The multiply and addition operations are performed on the paired-single data in the manner illustrated in Figure 4.

The SIMD instruction extensions that we consider in this paper are based on the MIPS V ISA Extensions [15]. The instruction formats and latency assumptions are summarized in Table 1. The LDPS and STPS instructions load/store a paired-single value (64 bits) from memory ignoring alignment. The PMUL (PADD/PSUB) instruction performs multiplication (addition/subtraction) of paired-single values. These paired-single instructions have 4 cycle latency and are fully pipelined. CVT.S.PL (CVT.S.PU) is used to extract the lower(higher) part of a paired-single value, and CVT.PS.S is used to create a paired-single value from two single-precision values.

The ADD_HL and LDS_HL instructions are not present in the MIPS V instruction extensions. ADD_HL adds the higher and lower parts of a paired-single value together. One example to show the usefulness of the ADD_HL instructions is the inner product operation commonly seen in the lighting stage. The inner product of two vectors (x1, y1, z1) and (x2, y2, z2) is x1*x2 + y1*y2 + z1*z2. The first two multiplication operations can be paired up. But the results must be added together. Without the ADD_HL instruction, we need to use the CVT.S.PU or CVT.S.PL instruction to extract the higher or lower half to a separate register before performing the addition.

The LDS_HL instruction duplicates a single-precision value to form a paired-single value. We use the computation of transforming normal vectors (multiplication of a 1x3 vector and 3x3 matrix) to illustrate the use of this instruction. The pseudo C codes are as follows (u and m represent an array of vertex coordinates and the transformation matrix respectively):

```
for  (i=0;i< number of vertices;i++)
{ q[i][0] =u[i][0] * m[0,0]+u[i][1] * m[1,0]+u[i][2] * m[2,0];
  q[i][1] =u[i][0] * m[0,1]+u[i][1] * m[1,1]+u[i][2] * m[2,1];
  q[i][2] =u[i][0] * m[0,2]+u[i][1] * m[1,2]+u[i][2] * m[2,2];
}
```

To exploit the parallelism across two vertices, we unroll the loops once and reorder instructions such that the independent floating-point operations can be easily paired up. The modified version is listed below:

```
for (i=0;i< number of vertices;i=i+2)
{
  q[i][0]     = u[i][0]     *  m[0,0] + u[i][1]     * m[1,0] + u[i][2]     * m[2,0];
  q[i+1][0] = u[i+1][0] * m[0,0] + u[i+1][1] * m[1,0] + u[i+1][2] * m[2,0];
  q[i][0]     = u[i][0]     *  m[0,1] + u[i][1]     * m[1,1] + u[i][2]     * m[2,1];
  q[i+1][0] = u[i+1][0] * m[0,1] + u[i+1][1] * m[1,1] + u[i+1][2] * m[2,1];
  q[i][0]     = u[i][0]      * m[0,2] + u[i][1]     * m[1,2] + u[i][2]     * m[2,2];
  q[i+1][0] = u[i+1][0] * m[0,2] + u[i+1][1] * m[1,2] + u[i+1][2] * m[2,2];
}
```

To perform the paired-single multiplication over vertices i and i+1, we need to form paired-single values for each element of the transformation matrix (i.e., (m[0,0],m[0,0]), (m[1,0],m[1,0])…..). The instruction LDS_HL is used for this purpose. Without the LDS_HL instruction, it will require one load and CVT.PS.S instruction to form each pair.

4 Experimental Methodology

In this section, we describe the simulation environment and processor models considered in this paper.

4.1 Simulation Framework

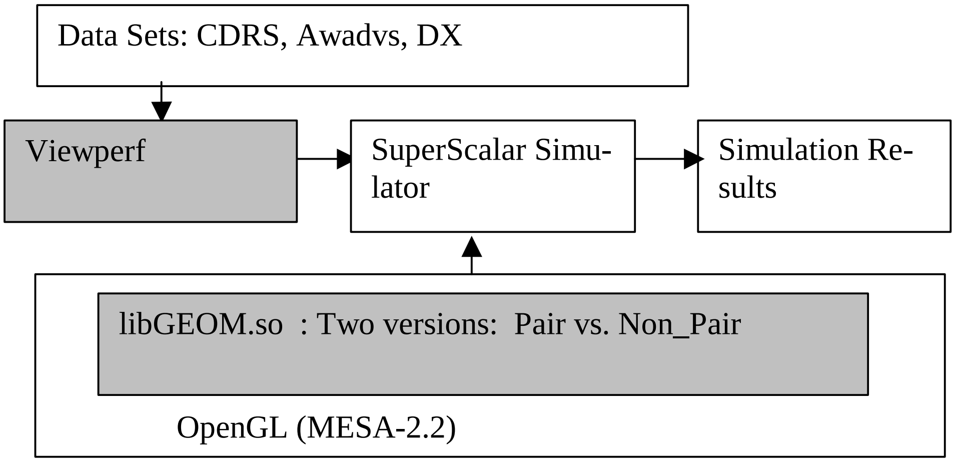

Data Sets: CDRS, Awadvs, DX
Viewperf
SuperScalar Simulator
Simulation Results
libGEOM.so : Two versions: Pair vs. Non_Pair
OpenGL (MESA-2.2)

Figure 5: Simulation framework.
libGEOM.so is a shared library including all the routines associated with the geometry processing. We only instrument code in the highlighted boxes.

Our simulation environment (shown in Figure 5) uses ATOM [25] to perform execution-driven simulation. This simulation framework consists of two components. The first component is MESA, a software implementation of the OpenGL specification. A shared library that contains all of the routines associated with geometry computation is separated from the complete MESA implementation. ATOM allows us to instrument only this geometry library and the application itself. In this way, we can simulate the environment where the host CPU is responsible for database traversal and geometry processing, while specialized hardware is used to process the remaining tasks in the graphics pipeline. We modified four routines, which account for 75% to 90% of the total execution time for all the benchmarks we ran, to incorporate paired-single instructions. These routines correspond to the Light, XformV, XformN and Div by w/Map as described in Section 2.2.

The second component of our simulation framework is an ATOM-based simulator that models an out-of-order superscalar processor with speculative execution [4], whose instruction set is based on the DEC Alpha processor [24].

Processor Model	Total Issue Width	# of Integer Functional Units				# of Floating-Point Functional Units				
		Issue Limit	loads& stores	Control flow	other	Issue Limit	mul	div	sqrt	other
Base	4	4	2	2	4	2	1	1	1	1
2xBase	8	8	4	4	8	4	2	1	1	2
4xBase	16	16	8	8	16	8	4	1	1	4
2xFP	6	4	2	2	4	4	2	1	1	2
4xFP	10	4	2	2	4	8	4	1	1	4

Table 2: Instruction issue rules (ready instructions are issued in fetch order).

To simulate the new instructions, we place innocuous (but unique) “marker” instructions where we want to replace the original code with new instructions. The operands of the marker instructions indicate different instruction types (e.g., LDPS, MULPS, etc). The appropriate operands of the new instructions are passed through the next 2 or 3 marker instructions, depending on the number of the operands required. In this way, instruction dependencies are accurately maintained. The simulator decodes each instruction and takes appropriate actions to simulate the paired-single execution when it encounters the marker instruction.

4.2 Processor Models

The baseline processor model studied in this paper is a 4-way, out-of-order issue superscalar processor. The issue rules and the functional unit latencies are summarized in Table 2 and Table 3. The maximum number of instructions that can be inserted into the dispatch queue or committed is equal to the issue width. When an instruction is inserted into the dispatch queue, its destination register is mapped to a physical register and a reorder buffer entry is allocated. Once an instruction is issued, it is removed from the dispatch queue, but the register mapping remains active until this instruction commits in program order from the reorder buffer [5]. Note, the reorder buffer size is determined by the number of physical registers. We implement a precise exception model, so an instruction can only commit when all the instructions preceding it in program order have completed. We assume a perfect

Instruction Type		latency	pipeline
Integer	multiplication	6	yes
	load	2	yes
	store	1	yes
	control flow	1	yes
	other	1	yes
Floating-point	32-bit div	8	no
	64-bit div	16	no
	square root	33	no
	other	4	yes

Table 3: Instruction latencies.

memory system (i.e., every memory reference and instruction fetch hit in the L1 cache)[2], a unified dispatch queue and separate register files for the integer and floating-point functional units. Speculative execution is enabled by implementing the branch prediction scheme proposed by McFarling [11] and precise exceptions are imposed.

To investigate the effect of a wider issue superscalar processor on the performance of geometry processing, we examine the following 4 models: 2xBase, 4xBase, 2xFP and 4xFP as listed in Table 2. The 2xBase and 4xBase models are 8-way and 16-way issue processors respectively. The issue rules are similar to the Base model. However, for most instruction types, two or four times the number can be issued in one cycle. The exceptions are division and square root, which remain the same as the baseline model. The reason for not doubling these two functional units is for a fair performance comparison between 2xFP and a baseline processor with the paired-single instruction since the paired-single operations are not implemented for division and square root. For the 2xFP (4xFP) configurations, we double (quadruple) only the floating-point functional units and issue width. The number of

[2] The miss rates for a 64-K, 2-way set associative D-cache and 8-K direct-mapped I-cache are both less than 2% for most of the benchmarks. Thus, we assume a perfect memory system to reduce simulation time.

the integer functional units remains the same as the baseline processor. Then total issue width becomes 6 and 10 for 2xFP and 4xFP respectively.

5 Simulation Results

We use CDRS, Awadvs and DX from Viewperf as our benchmarks due to lengthy simulation time. Each of these benchmarks is composed of several tests. For space reasons, we only present test1 from each benchmark. These three tests are chosen because they are representative of all the tests (the complete simulation results can be found in [27]). CDRS test1 is a wireframe rendering application and both DX and Awadvs have at least one light source. Awadvs has only 3.4 vertices on average per glBegin/glEnd, while CDRS and DX test1 have 30 and 96 vertices respectively.

We present our simulation results in three parts. First, we investigate how well conventional superscalar processors exploit the parallelism in geometry processing. Then we present the performance of paired-single execution on different processor models. Finally, we compare the relative performance of different processors with and without paired-single instructions accounting for potential increases in CPU clock cycle time.

5.1 Scaling a Conventional Design

The dispatch queue and register file sizes have significant impact on how much ILP can be exploited in a superscalar processor. A wider issue machine usually requires a larger dispatch queue and register file. In order to evaluate the potential performance improvement achieved by increasing the issue width, the superscalar simulator is first configured with 2048 floating-point and 2048 integer registers. With such a large register file, the CPU never stalls due to a lack of free registers. We then vary

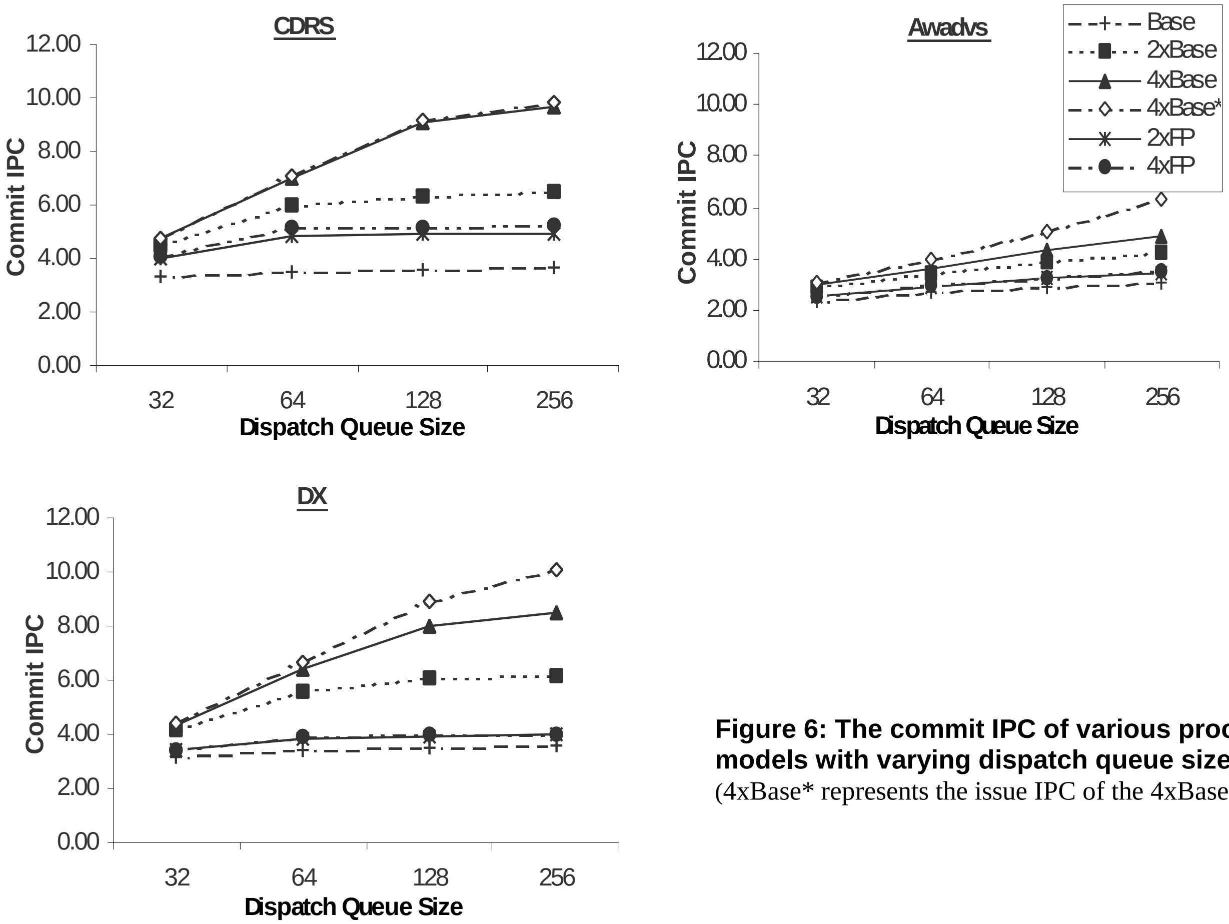


Figure 6: The commit IPC of various processor models with varying dispatch queue size.
(4xBase* represents the issue IPC of the 4xBase processor)

the dispatch queue size from 64 to 256. The commit IPC[3] for the various processor models is shown in Figure 6.

CDRS test1 has the highest IPC for all the configurations among all the benchmarks we ran. With the largest dispatch queue (256), the commit IPC of 2xBase (8-way issue) is 6.5, almost twice that of Base (3.4). Doubling only the floating-point issue width (2xFP) achieves 36% performance improvement. However, quadrupling only the floating-point issue width (4xFP) does not perform any better than the 2xFP because the loads that read the source operands for the floating-point operations become the bottleneck. The commit IPC of the 4xBase processor (16-way issue) is 9.7, about 2.7 times that of Base.

The continuous growth of commit IPC as the issue width increases indicates that a lot of parallelism does exist in geometry processing for this benchmark. Note that the commit IPC grows with larger dispatch queue size, but the degree of improvement diminishes after a certain size. This point occurs around a dispatch queue size of 32 for the Base model, 64 for both the 2xBase and 2xFP, and 128 for 4xBase.

For DX test1, the commit IPC of the processor models smaller than 4xBase are comparable to CDRS test1, except for 2xFP, which only achieves 13% performance improvement. The ratio of floating-point arithmetic operations to load instructions is 1:1 for DX test1 and 2:1 for CDRS test1. Thus, increasing the floating-point issue width alone does not improve the performance of DX as much as that of CDRS. For DX test1, the commit IPC of 4xBase is 8.48, lower than CDRS test1 (9.7). The lower commit IPC is due to more mispredicted branches. The issue IPC for 4xBase is plotted in Figure 6 to illustrate this scenario. Issued instructions can not commit if a preceding branch in the program order was mispredicted. DX test1 has a larger difference between the issue and commit IPC than CDRS test1. This is because the lighting computation has more conditional branches than the transform, thus the performance of a light-intensive applications like DX test1, is more subject to branch prediction accuracy than a wireframe rendering application like CDRS test1.

Awadvs test1 has the lowest IPC, primarily because of its small number of vertices (3.4) per glBegin/glEnd. Observe that the commit IPC increases linearly with the dispatch queue size, hence, for this benchmark, the dispatch queue is still the bottleneck even when it has 256 entries. Because the 4xFP processor performs equal to 2xFP for all the benchmarks we ran, we no longer consider this configuration in the following analysis.

[3] The commit IPC is the ratio of the number of instructions that commit to the total execution cycles.

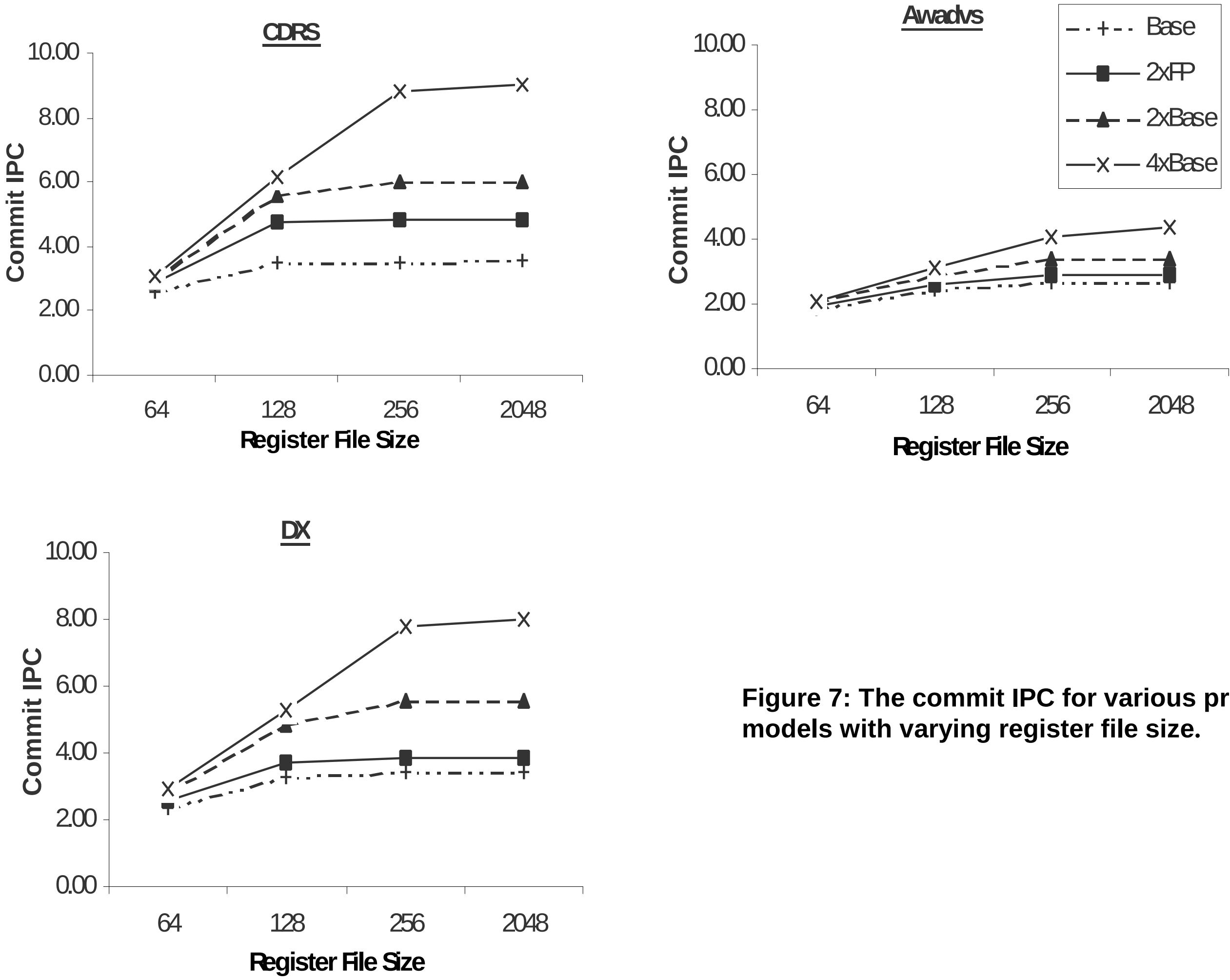


Figure 7: The commit IPC for various processor models with varying register file size.

To evaluate how the register file size affects performance, we keep the dispatch queue size constant (64 entries for Base, 2xFP and 2xBase and 128 entries for 4xBase) while varying the register file size from 64 to 256. The results are shown in Figure 7. The 2048 entry register file size is shown as a reference point. Using more than 128 registers for Base, 2xFP and 2xBase and 256 registers for 4xBase does not improve performance significantly.

In the next section, we analyze the benefit of the paired-single execution on the Base, 2xFP and 2xBase processors. 4xBase is a 16-way issue machine and requires a 128-entry dispatch queue and 256 registers. This configuration is too large to achieve a practical implementation by simply scaling

	CDRS			AW		
Inst type	Inst count		Reduction %	Inst count		Reduction %
	Orig	Pair		Orig	Pair	
add	7645898	3965927	48	18884890	11315926	40
sub	5141	5141	0	2195114	2073434	6
mul	10760847	5666089	47	29192094	18531619	37
ld	8052969	7625453	5	43112841	36494561	15
sts	3936594	2521807	36	16402547	12928214	21
cvt.s.pl/u	0	0	-	0	65905	-
cvt.ps.s	0	0	-	0	1111734	-
other	31413092	31413092	0	139816628	139816628	0
total	61809400	51192400	17	247409000	220068000	11

	DX		
Inst type	Inst count		Reduction %
	Orig	Pair	
add	7577899	4293152	43
sub	4879	4879	0
mul	11572039	7058061	39
ld	21965717	19932776	9
sts	10809407	9211454	15
cvt.s.pl/u	0	93461	-
cvt.ps.s	0	80171	-
other	109218938	109218938	0
total	161144000	149823000	7

Table 4: Instruction distribution for non-paired and paired-single execution

the Base configuration, hence we do not consider it further.

5.2 The Performance Improvement of Paired-Single Execution

Adding paired-single instructions not only reduces the number of single-precision floating-point add subtraction, and multiply instructions, it can also eliminate load/store instructions if the LDPS instruction can be used to load two single-precision floating-point values together. Table 4 shows the instruction distribution for both non-paired and paired-single execution. The number of multiply and add instructions is reduced by approximately 50% for CDRS test1, 40% for DX and Awadvs test1. The number of load and store instructions are reduced by 5% to 15% and 15% to 36%, respectively. Recall that paired-execution requires extra instructions (CVT.S.PL/U and CVT.PS.S) to create a paired-single value or extract the lower (higher) part of a paired-single value. However, for the benchmarks we tested, they are negligible. They account for less than 1% of all instructions for DX and Awadvs test1, and CDRS test1 does not use these instructions. All paired-single values are created using the LDPS

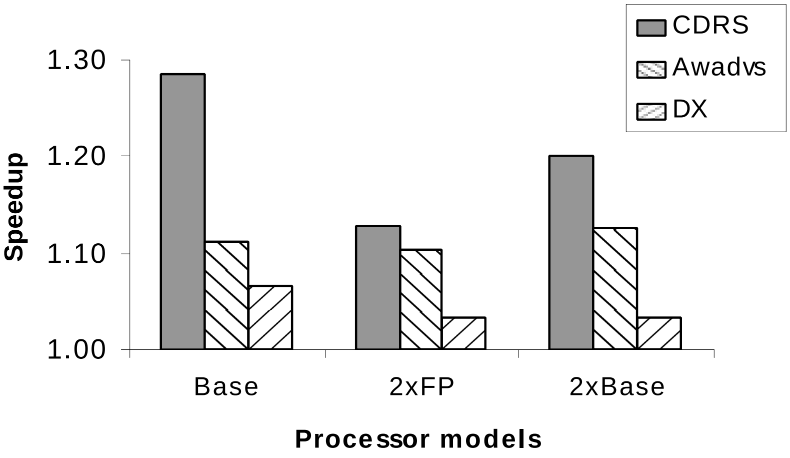


Figure 8: Performance improvement of paired-execution.

instruction. After taking into account these extra instructions, the overall instruction reduction is 17% for CDRS, 11% for Awadvs and 7% for DX.

Reducing the number of instructions has two potential advantages. First, combining two floating-point operations together effectively enables the CPU to look further ahead to find independent instructions to issue. In other words, adding paired-single instructions could achieve the same effect as increasing the dispatch queue size. Second, it can improve the instruction cache performance. We did not analyze this due to the low instruction cache miss rate for the benchmarks we ran. We can expect higher performance impact on an embedded system, which is usually configured with a smaller instruction cache.

We evaluate the performance improvement of paired-single execution on the Base, 2xFP and 2xBase models. The simulation results are shown in Figure 8. The y-axis shows the speedup of paired-single over non-paired execution. CDRS test1 has the best performance improvement, 28% on the Base model, 13% on the 2FP and 20% on the 2xBase. DX test1 has the smallest performance improvement since it only reduces the number of instructions by 7%. Note that our speedups may not be

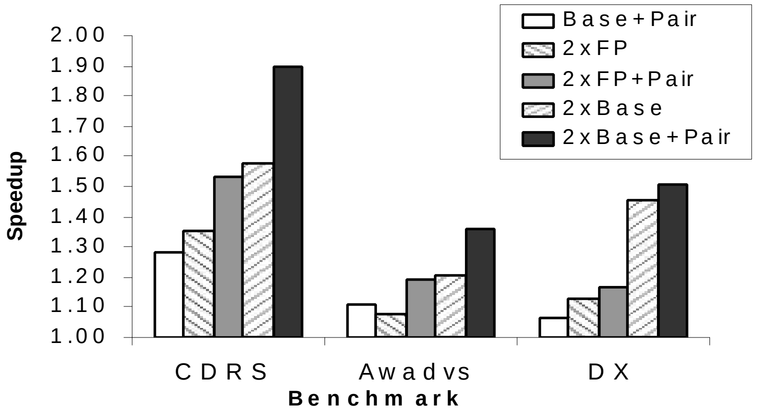


Figure 9: Relative speedup of various processor models over the Base with a dispatch queue of 64 entries and 128 registers.

optimal. First, there are some routines required for geometry processing that we have not converted to use paired-single instructions. However the impact on performance of these procedures should not be substantial. Second, we have not optimized the instruction schedule of the paired-single sequence. Different computation sequences incur different register allocation and instruction scheduling, and analyzing these effects requires further research.

5.3 Paired-Single vs. Wider Issue

In this section, we discuss the relative performance of various processor models with and without the paired-single instruction set. First, we compare relative performance assuming that CPU cycle time remains the same for all processor models. Then, we investigate how changes in cycle time affect overall performance. All the processor models are configured with a 64-entry dispatch queue and 128 registers. These numbers are chosen such that the performance of an 8-way issue (2xBase) processor is not constrained too much by the dispatch queue and register file and the processor configuration is within a reasonable range.

The simulation results, assuming no changes in CPU cycle time, are shown in Figure 9. The y-axis is the speedup of the various processor models over the Base configuration. Adding paired-single instructions effectively doubles the floating-point issue width so a Base processor with the paired-single instruction extension can potentially achieve the same floating-point processing capability as 2xFP.

Our results show that Base+Pair performs within 7% of 2xFP for CDRS and DX and it even outperforms 2xFP for Awadvs test1. Besides the advantage of doubling floating-point processing rate, adding paired-single instructions can better utilize the dispatch queue, as mentioned in the previous section. For an application where the dispatch queue is the performance bottleneck, like Awadvs test1, Base+Pair has a performance advantage over 2xFP. An 8-way issue processor using paired-single instructions (2xBase+Pair) can achieve 1.9 speedup over Base for CDRS test1.

5.3.1 Effects on Clock Cycle Time

Previous studies have shown that increasing issue width has significant impact on the processor cycle time [4][21]. Palacharla et al. [21] studies how the instruction dispatch, issue logic and data bypass delay varies with different issue width. Their results show that the issue logic determines the critical path delay in a 0.35um technology for both 4-way and 8-way issue processors (not considering cache and register files) and the wakeup logic delay (part of issue logic) grows linearly with the issue width. Farkas et al. [4] shows that the issue width determines the number of read/write ports to a register file, and thus can have significant impact on the cycle time.

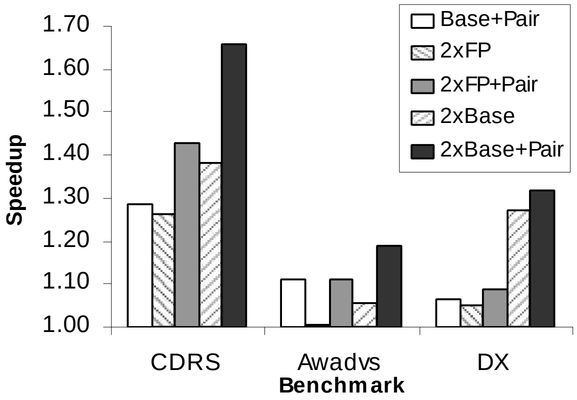


Figure 10: Relative performance of various processor models assuming that the window issue logic determines processor cycle time.

Processor Model	Register File Access Time(%)	Issue Logic Delay (%)
2xFP	0	7%
2xBase	50%	14%

Table 5: Cycle time increase over the Base model assuming 0.35um technology.

The percentage increase of issue logic delay and register file access time over the Base model are summarized in Table 5. We use a modified version of CACTI [8] developed by K. Farkas in [4] to generate the register file access time. Note that the floating-point register file of 2xFP has the same number of read/write ports as the integer register file of Base. Thus, the register file access time of 2xFP is equal to the Base access time. The 2xBase model increases the register file cycle time by 50% in a 0.35mu technology. We use the data provided in [22] to derive the issue logic delay. Linear extrapolation is used to obtain the data for configurations not studied in that paper. The 2xFP model increases the issue logic delay by 7%, and 2xBase by 14%.

We present simulation results in two sets. The first set assumes that issue logic (wakeup+selection) determines the critical path delay (Figure 10) and the second set assumes that register file access does (Figure 11). The y-axis is the speedup of the various processor models over the Base configuration. If the issue logic determines critical path delay, Base+Pair outperforms 2xFP for all the benchmarks. The performance difference is most substantial for Awadvs test1. The 2xBase+Pair processor model achieves 1.6 speedup for CDRS test1. However, if the CPU cycle time is determined by the register file access, increasing the issue width up to 8-way (2xBase) has negative impact on the performance

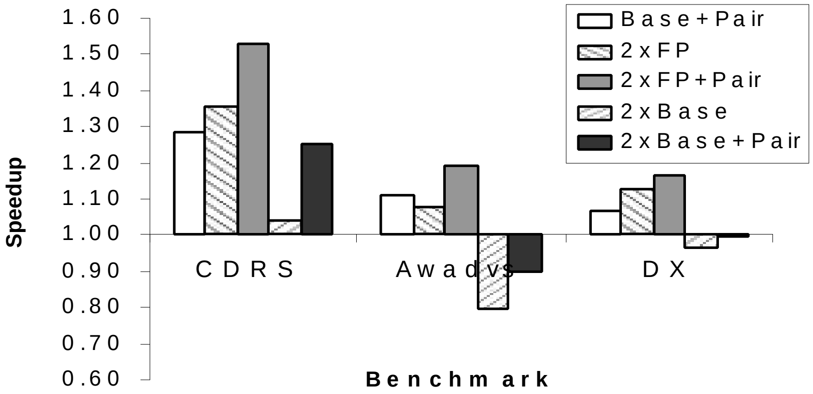


Figure 11: Relative performance of various processor models assuming that the register file access delay determines processor cycle time.

for Awadvs and DX test1 as shown in Figure 11. Thus, 2FP+Pair becomes the best design choice, achieving 1.5 speedup over the Base processor model for CDRS and 1.2 for both Awadvs and DX.

6 Performance Comparison: General Processors v.s. High-End Systems

To get an idea how well a general processor can execute the geometry pipeline, we compare the performance of a general processor with high-end graphics systems in terms of number of frames per second (the high end performance numbers are obtained from [19]). Note that the general processor performance shown here is optimistic because we do not take into account some system components such as TLB, instruction cache, CPU cycle time effects, etc. nor do we scale the technology for the custom systems. Nonetheless, this comparison provides a general idea of the relative performance. This comparison is shown in Figure 12 for test1 from CDRS, Awadvs, and DX. We look at three general processor models, Base, Base+Pair and 2xBase+Pair assuming the same CPU clock cycle time. The high-end systems that we compared to are HP Visualize fx6 for CDRS and SGI Infinite Reality for Awadvs and DX. These two systems achieve the highest benchmark results.

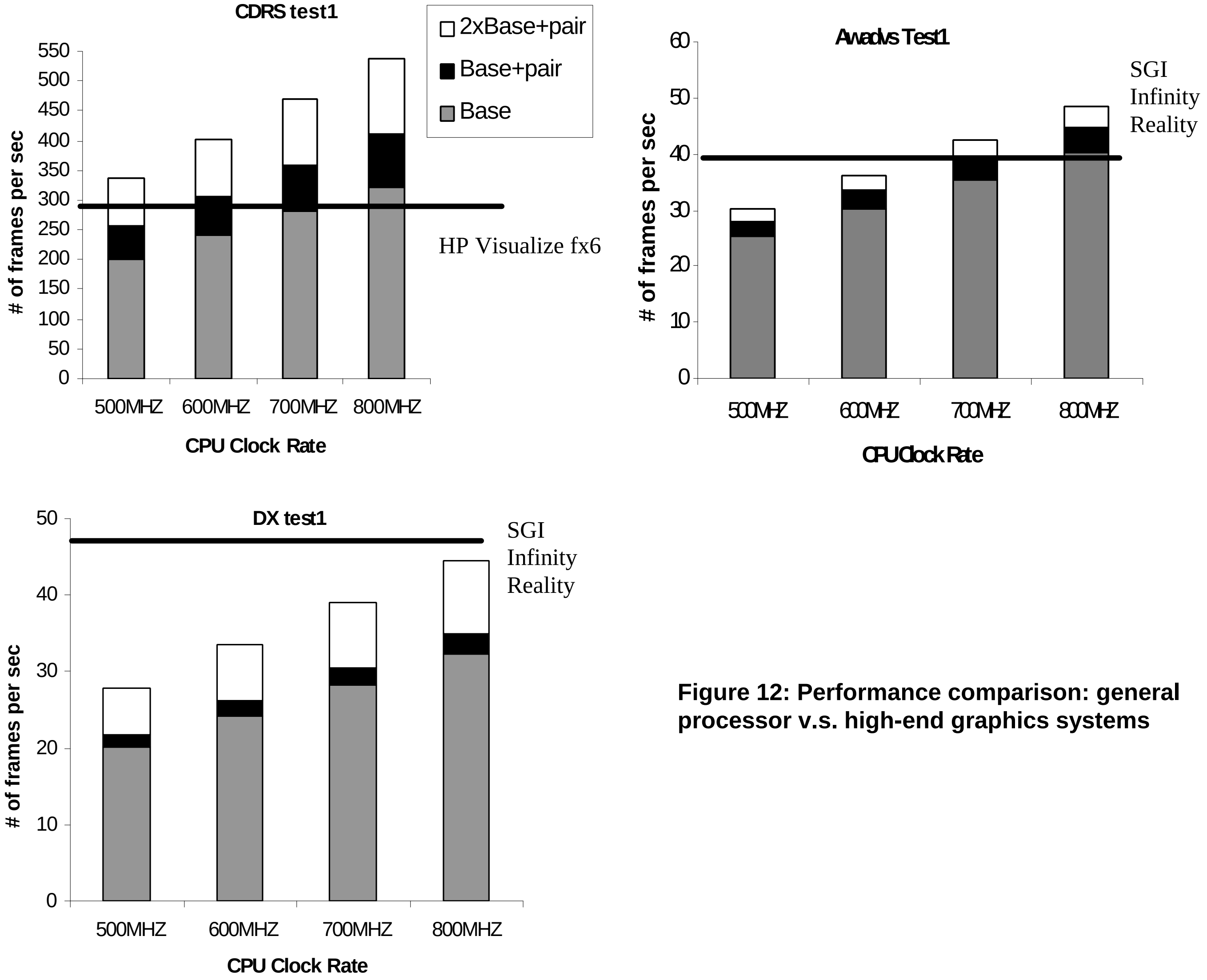


Figure 12: Performance comparison: general processor v.s. high-end graphics systems

Assuming a 500MHZ CPU clock rate, adding paired-single instructions on a 4-way issue processor (Base+Pair) achieves performance close to the high-end system for CDRS test1 (250 v.s. 290 frames per second). On an 8-way issue processor with paired-single instructions (2xBase+Pair), a general processor can even outperform the high-end system. However, a general processor performs less effectively for lighting-intensive applications like DX and Awadvs. For these two benchmarks, a 2xBase+Pair processor performs within 60% to 70% of the high-end system if CPU clock rate is

500MHZ. To predict the future general processor performance as the process technology progresses, we look at different CPU clock rates. For Awadvs test1, a 700MHZ, 2xBase+Pair processor can achieve performance similar to the current high-end system. For DX test1, a 2xBase+Pair processor still performs slightly lower than the current high-end system even the CPU clock rate reaches 800MHZ.

7 Conclusion

The widespread use of multi-media applications presents new design challenges for system designers. In this paper, we examine the performance of geometry computation in three dimensional graphics applications on future superscalar processors. Geometry computation is single-precision (32-bit) floating-point intensive. We investigate the performance of recently proposed instructions that perform two independent 32-bit operations by packing the operands in 64-bit registers and exploiting the existing 64-bit datapath. We use simulation to compare the performance of these new instructions, called paired-single to that achieved by increasing a conventional out-of-order processor's issue-width.

From our simulation results, we found that paired-single instructions improve performance by up to 28% on a 4-issue processor and 20% on an 8-issue. These improvements are comparable to those achieved by doubling only the floating-point issue width (2xFP). Our results reveal that 4xFP performs equal to 2xFP because load instructions that read source operands of floating-point operations become the bottleneck, and hence require a commensurate increase in the integer issue width.

We also found that the average number of vertices processed in each stage of the geometry pipeline (i.e., vertices per glBegin/glEnd) is the primary factor determining performance on superscalar processors. For benchmarks that have a large number of vertices per glBegin/glEnd, the speedup of an 8-way issue processor over a 4-way is 1.6 with a 64-entry dispatch queue and 128 registers. However, for benchmarks that have a small average number of vertices per glBegin/glEnd, the speedup is only 1.2.

Considering the impact of the issue width on the CPU cycle time, we looked at two pipeline stages that can be on the critical timing path. One is the register file access and the other is the issue logic. If the issue logic is on the critical path, an 8-way issue processor with paired-single instructions provides 20% to 65% performance improvement over a 4-way issue. However, if the register file access is on the critical path, the processor cycle time increases by almost 50% going from 4-way to 8-way. Thus doubling only the floating-point issue width of a 4-issue processor (2xFP) with paired-single instructions becomes the best design choice. The improvement over a 4-way issue processor ranges from 20% to 50%.

References

[1] K. Akeley, and T. Jermoluk. High-Performance Polygon Rendering. Computer Graphics, Volume 22, pages 239-249, August 1988.

[2] AMD 3DNow! Technology. http://www.amd.com/product/cpg/k623d/inside3d.html

[3] Digital Unix V4.0 Programmer’s Guide, pages 8-13.

[4] K. Farkas. Memory-system Design Considerations for Dynamically-scheduled Microprocessors. Ph.D. thesis, Department of Electrical and Computer Engineering, University of Toronto, January 1997.

[5] K. Farkas. N. Jouppi, and P. Chow. Register File Design Considerations in Dynamically Scheduled Processors. In Proceedings of the Second International Symposium on High Performance Computer Architecture, pages, 1997.

[6] J. Foley, A. van Dam, S. Feiner, and J. Hughes. Computer Graphics – Principles and Practice. Addison-Wesley, 1996.

[7] Intel MMX2. http://developer.intel.com/drg/news/katmai.html.

[8] N. Jouppi and S. Wilson. An enhanced access and cycle time model for on-chip caches. Technical Report 93.5, DEC Western Research Laboratory, July 1994.

[9] G. Kane. PA-RISC 2.0 Architecture. Prentice Hall PTR, 1996.

[10] L. Kohn, G. Maturana, M, Tremblay, A. Prabhu, and G. Zyner. Visual Instruction Set (VIS) in UltraSPARC™. In Proceedings of COMPCON’95, pages 462-469, March 1995.

[11] R. Lee and M. Smith. Media Processing: A New Design Target. IEEE Micro, pages 6-9, August 1996.

[12] S. McFarling. Combing Branch Predictors. Digital Equipment Corporation Western Research Lab Technical Note TN-36, 1993

[13] MESA library. http://www.ssec.wisc.edu/~brianp/Mesa.html

[14] Microprocessor Forum, October 1997.

[15] MIPS V ISA Extension. http://www.sgi.com/MIPS/arch/ISA5/

[16] MMX™ Technology. Intel Architecture MMX Technology Programmer's Reference Manual. Intel Corporation, March 1996

[17] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In Proceedings of SIGGRAPH '92, pages 231-240, August 1992.

[18] J. Montrym, D. Baum, D. Dignam, and C. Migdal. InfiniteReality: A Real-Time Graphics System. In Proceedings of SIGGRAPH '97, pages 293-302, August 1997.

[19] Motorola AltiVec Technology. http://www.mot.com/SPS/PowerPC/AltiVec

[20] OpenGL Performance Benchmark – Viewperf. http://www.specbench.org/gpc/opc.static/vp50.html

[21] S. Palacharla, N. Jouppi, and J. Smith. Complexity –Effective SuperScalar Processors. In Proceedings of the 24th Annual International Symposium on Computer Architecture, pages 206-218, 1997.

[22] S. Palacharla, N. Jouppi, and J. Smith. Quantifying the Complexity of Superscalar Processors. Technical Report CS-TR-96-1328, University of Wisconsin-Madison, November 1996.

[23] P. Ranganathan, S. Adve, and N. Jouppi. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. In Proceedgins of the 26th Annual International Symposium on Computer Architecture, pages 124-135, 1999.

[24] R. Sites. Alpha Architecture Reference Manual. Digital Press, 1992.

[25] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In Proceedings of the ACM SIGPLAN '94 Conference on Programming Languages, March 1994.

[26] M. Woo, J. Neider, and T. Davis. OpenGL Programming Guide. Addison-Wesley, 1997.

[27] C. Yang, B. Sano, and A. Lebeck. Exploiting Instruction Level Parallelism in Geometry Processing for Three Dimensional Graphics Applications. Technical Report CS-1998-14, Computer Science Department, Duke University, September 1998.