Graphics Hardware (2004)
T. Akenine-Möller, M. McCool (Editors)

A Flexible Simulation Framework for Graphics Architectures

J. W. Sheaffer, D. Luebke, and K. Skadron

Department of Computer Science, The University of Virginia

Abstract

In this paper we describe a multipurpose tool for analysis of the performance characteristics of computer graphics hardware and software. We are developing Qsilver*, a highly configurable micro-architectural simulator of the GPU that uses the* Chromium *system's ability to intercept and redirect an OpenGL stream. The simulator produces an annotated trace of graphics commands using Chromium, then runs the trace through a cycle-timer model to evaluate time-dependent behaviors of the various functional units. We demonstrate the use of Qsilver on a simple hypothetical architecture to analyze performance bottlenecks, to explore new GPU microarchitectures, and to model power and leakage properties. One innovation we explore is the use of dynamic voltage scaling across multiple clock domains to achieve significant energy savings at almost negligible performance cost. Finally, we discuss how other architectural features and experiments might be incorporated into the Qsilver framework.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture

1. Introduction

Simulation has long been a vital tool for the study and design of computer architecture, but simulation of graphics architecture presents some unique challenges. Commodity graphics hardware is evolving at a tremendous rate, with each successive generation adding not only performance but fundamentally new functionality. In the time required to build a complex simulation infrastructure, the simulated architecture can easily become obsolete. Furthermore, the architecture of modern graphics processors or *GPUs* are largely secret; vendors in the highly competitive PC graphics arena are reluctant to release architectural details, such as texture cache design, even to their registered game developers. Most GPU simulators today can be broadly classified as cycle-accurate circuit-level simulations, which are extremely costly to run and available only to vendors because of the enormous manpower they require to produce, and functional emulators, which model the user-level interface (for example, a shading assembly language or API) without any underlying architectural structure. This situation, coupled with the sheer complexity and performance of modern GPUs, presents a significant challenge for researchers interested in exploring architectural innovations, modeling fine-grained effects such as intra-frame performance bottlenecks, or simulating power, leakage, and thermal properties of GPUs.

We present a simple, flexible simulation framework for graphics architectures that fills the gap between cycle-accurate VHDL/Verilog models and functional emulators. Our framework builds on the *Chromium* system [7], which intercepts and processes streams of OpenGL calls. We use Chromium to record a trace of graphics instructions during an application and then to instrument the rendering pipeline to accumulate statistics during playback of that trace. The instrumented trace is then run through the simulator itself, which is essentially a queue-driven, cycle-timer model of data and computation flow through the functional stages, decoupled by queues and caches, of the GPU. The resulting simulation framework is fine-grained enough to be useful for a variety of tasks – for example, we demonstrate power modeling of a hypothetical architecture – but efficient enough to analyze long (multi-second) runs of real-world applications. The framework is also flexible, allowing greater or lesser amounts of detail to be modeled and providing a natural structure for adding experimental pipeline stages or functional units via Chromium *stream processing units* or

© The Eurographics Association 2004.

delivered by
EUROGRAPHICS
DIGITAL LIBRARY
www.eg.org diglib.eg.org

SPUs. We call our fast Chromium-based simulation framework *Qsilver*.

To demonstrate the applicability of Qsilver, we present two simple studies. First, we analyze the performance of a simple hypothetical architecture running the recent Splash Damage game "Return to Castle Wolfenstein: Enemy Territory." We show that, as might be expected, different stages of the GPU pipeline act as the bottleneck during different frames and during different segments of a single frame. Next, we build a power model of the hypothetical architecture and use Qsilver to explore power-related architectural optimization.

Note that these studies serve as illustrative examples rather than serious architectural experiments. Our hypothetical GPU pipeline is a prototype, modeled only coarsely, and our power model represents a best-guess estimate of a current graphics processor. We emphasize that our goal here is not to perform conclusive architectural research but to describe a useful tool and demonstrate the sort of interesting studies that vendors and researchers, armed with the architectural details that our prototype model lacks, could easily perform with the Qsilver approach.

2. Related work and motivation

The advent of detailed but flexible, configurable, cycle-accurate CPU simulators in the 1990s for complex, superscalar architectures served as the catalyst for an explosion of quantitative research in the computer architecture community. The most prevalent simulator in academic architectural research is SimpleScalar [3]; other simulators used in specific circumstances include Rsim [6] for multiprocessors, as well as Simics [10] and SimOS [13] for capturing operating-system and multi-programmed behavior.

By describing instruction flow at the granularity of individual steps through the CPU pipeline, these simulators allowed research and design to move beyond simplified, imprecise analytical models or cumbersome, logic-level models. Instead, architects could analyze detailed tradeoffs under realistic workloads and estimate how various microarchitectural choices affected instruction throughput. Examples include the impact of different cache and branch predictor algorithms, the impact of different superscalar out-of-order instruction-issue designs, and the ability to evaluate a host of novel CPU organizations such as hyper-threading.

Subsequent power-modeling capability launched another round of innovation by allowing architects to estimate the energy efficiency of different processor organizations, verify that new microarchitecture innovations are justifiable from an energy-efficiency standpoint, and explore microarchitectural techniques for managing energy efficiency. The dominant power model today is Wattch [4], which uses calibrated analytical models to allow flexible and configurable estimation of power for a variety of structures, structure sizes, organizations, and semiconductor technology generations or "nodes". Other power models that use circuit-extracted data have been described, but they are based on a specific implementation and tend to be inflexible, especially in terms of scaling to future semiconductor technology nodes. These two approaches can be combined, using the circuit-extracted model as calibration for Wattch's analytical models; see for example a recent study of hyper-threading using IBM's circuit-extracted PowerTimer [9].

Our goals with Qsilver are to stimulate the same kind of innovation in the GPU community, to stimulate greater cross-fertilization with the general-purpose CPU architecture community, and to enable new studies in power-aware and eventually thermal-aware design. The purpose of this paper is to report results with a prototype to show the value of this kind of simulation approach.

3. A framework for simulation of graphics architecture

Driving our simulator is a trace of graphics commands, instrumented with additional statistics that describe the behavior of those commands. For example, in our system the graphics command that produces a triangle to be rasterized would be annotated with the total number of fragments generated, number of fragments written to the depth buffer, average number of texels accessed per fragment, and so on. Exactly which statistics must be gathered depends on the level of architectural detail we wish to model. We then feed this instrumented trace into a "cycle-timer" model that simulates the flow of data and computation through each stage in the decoupled microarchitecture of the GPU. From the cycle-timer model we can count the number of operations and estimate the computational load in each unit that we model, which in turn provides a basis for modeling power dissipation. We can also examine high-level behavior such as the migration of performance bottlenecks (vertex processing, fragment processing, memory bandwidth, etc.) between or within frames.

3.1. Capturing the trace

We use Chromium [7] both to capture the original trace and to gather the statistics used to instrument it. Chromium is a system for manipulating streams of OpenGL commands; common uses include splitting rendering across multiple machines for tiled display or implementing sort-first and sort-last parallel graphics architectures on clusters of PCs. The use of Chromium simplifies the task of capturing the behavior of real-world applications, since Chromium can be applied non-invasively to applications for which source code is not available and can store an application's OpenGL stream to disk for playback and reproducible analysis later. Chromium also lets us avoid the development and running-time overhead of implementing simulations of low-level structures, such as rasterization or texture filtering hardware.

© The Eurographics Association 2004.

Instead we use OpenGL itself as a rasterization engine, and instrument the OpenGL stream using mechanisms such as occlusion queries (see http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt) and programmable shaders to gather the essential statistics. For example, we employ a Chromium SPU that renders each triangle twice, once into the destination image buffer and once into a scratch buffer in which the depth test is disabled. Occlusion queries wrapped around both triangles provide a count of fragments drawn and fragments created, which we later use for estimating computational load and memory bandwidth used by the fragment processor and the depth buffer.

Specifically, our prototype simulator implements the following SPUs:

Expand vertex arrays. For efficient rendering, OpenGL applications generally collect object geometry into vertex arrays. To simplify the task of associating fragments with the primitives (vertices and triangles) that produce them, we dereference vertex arrays into immediate-mode `glBegin`, `glVertex`, `glEnd` calls. Note that we can easily retain the vertex array organization in our annotations if, for example, we wish to accurately simulate the post-transform vertex cache (but our current prototype simply uses a statistical model of cache hits).

Triangulate complex geometries. We wish to count total versus occluded fragments generated by each primitive, but complex primitives such as triangle strips can self-occlude. In this SPU we split triangle strips, triangle fans, and complex polygons into triangles so that our occlusion queries are guaranteed to measure only planar polygons. Though our prototype implementation does not do so, this SPU could also generate triangles for other primitives such as thick lines and point sprites.

Unfold display lists. We also store display lists and play them back when referenced in the OpenGL stream. In addition to potentially containing self-occluding geometry, display lists may encapsulate changes to rendering state. These must be exposed to the simulator since we wish to track information such as texture accesses, which are affected by the number of textures bound, MIP-map filtering mode, etc.

Count visible fragments rasterized. The aforementioned SPUs condition the OpenGL stream for annotation; this SPU begins the actual data collection. As each triangle is rendered, we enclose it in an occlusion query, which returns the number of fragments that pass the depth test. We store this information in the annotated trace that will later be used to drive the cycle-timer model.

Count total fragments rasterized. To get an accurate estimate of depth-buffer bandwidth, we must also count total fragments rasterized. We do this by rendering each polygon again, this time into a separate buffer with the depth test disabled. Again, an occlusion query returns the total number of fragments rasterized.

Count average texture accesses per fragment. The number of texture accesses depends on several things: how many textures are bound, what form of texture filtering is enabled, and the per-pixel texture LOD of the fragments. Full trilinear MIP-mapping presents a challenge because the number of texels read for magnification (4) is different from the number read for minification (8). We could use a software rasterizer that directly evaluates per-fragment texture LOD, but this sacrifices much of the simplicity and speed that we gain by using Chromium and OpenGL to perform rasterization and filtering. Instead, we obtain an average texture access count by binding a texture that encodes the MIP level directly, then binding a fragment program that kills fragments based on the hardware-filtered texture value. Another occlusion query counts surviving fragments and uses the result to calculate average texture accesses per fragment.

In practice, the last three SPUs can be combined to increase rendering efficiency. In our experiments with the *Enemy Territory* game, the entire process of annotating a prerecorded OpenGL trace, preparing input for the simulator core, runs at near-interactive rates, roughly 3-15 frames per

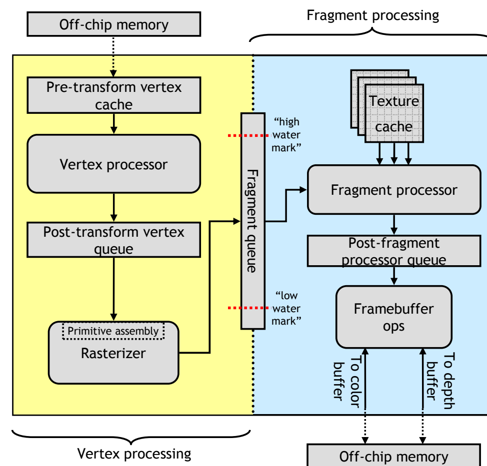

Figure 1: *The functional units and data flow of our hypothetical graphics architecture. The units can be classified as queues, caches, or data processing components. The fragment queue separates the vertex- and pixel-processing components of the GPU; we exploit this decoupling in the multiple-clock-domain experiment of Section 5.2.2.*

3.2. Applying the cycle-timer model

The instrumented trace consists of a stream of geometric primitives annotated with statistics denoting fragments produced and occluded, texture accesses, and potentially other information (such as vertex indices for post-transform cache

© The Eurographics Association 2004.

analysis), as well as the relevant graphics state (such as number of textures, number/kind of lights, and texture filtering state) for each batch of geometry. Once the trace is acquired we next run a cycle-timer simulation that pulls the fragments and vertices, cycle by cycle, through each stage of the GPU pipeline. The pipeline is modeled as a series of functional units connected by queues and caches; Figure 1 shows our prototype model of a hypothetical fixed-function GPU.

In the cycle-timer model, Qsilver repeatedly advances a global time counter by one cycle and advances the simulation stage-by-stage backwards through the pipeline. For example, each cycle a maximum number of fragments may be written to the framebuffer (possibly limited further by memory bandwidth to the color and/or depth buffers). This increments operation counters in the "Framebuffer ops" unit (representing arithmetic operations performed for blending, Z-buffer comparison, etc) and drains the queue between that unit and the fragment processor. If the fragment processor is ready to produce a fragment (actually the fragment processor operates on tiles, which are 2×2 fragments in our model), the processed fragments are added to the queue. The rate of the fragment processor depends on the computation demanded by the rendering state (e.g., how many textures to interpolate), on the annotated trace statistics for that set of fragments (e.g., how many texture value interpolations are necessary due to magnification/minification), on the intrinsic architectural parameters of the fragment processor (how many pipelines, how deeply pipelined), and on the bandwidth to texture memory (accounting for texture cache behavior, which we currently model only statistically). Note that in a given cycle, the fragment processor may stall for several reasons: the subsequent queue may be full (this generally implies the system is framebuffer bound), because the texture fetches cannot be performed (system may be texture bound, or poor texture cache locality), or because the incoming queue is empty (vertex or rasterization bound). As the fragment processor produces fragments we increment operation counters that reflect the arithmetic operations incurred per fragment.

Qsilver follows a similar process through the various stages of the cycle-timer model. The fragment processor drains the "fragment queue", a queue of fragment interpolants (again, organized into tiles) filled by the rasterizer. This queue effectively partitions the GPU into vertex- and pixel-processing components; since a full or empty fragment queue implies that the vertex or fragment processor is stalled, queue occupancy provides a metric for whether the system is currently transform- or fill-rate limited. We therefore use occupancy to drive the multiple-clock domain experiment described in Section 5.2.2.

The rasterizer, which in our prototype incorporates primitive assembly, fills the fragment queue and drains the post-transform cache. Computational operations executed by the rasterizer are derived from the number of fragments generated per primitive (available from our annotated trace) and the number of active interpolants (dictated by current rendering state). The post-transform cache is modeled as a simple FIFO, which accurately reflects current architectures, but we do not currently model cache hits for indexed vertex arrays. Instead, the post-transform cache is treated as a queue, filled by the vertex processor.

The vertex processor (modeled as two separate pipelines in our hypothetical architecture) performs significantly more arithmetic per operation than other units. Again, the details depend on rendering state (e.g., number and kind of lights). To estimate the computational load of transforming and lighting vertices, we use a collection of compiled assembly code for implementing a fixed-function pipeline on programmable GPUs, obtained from a Cg model provided by Nvidia online (http://developer.nvidia.com/object/cg_fixed_function.html). This code includes the various permutations of point, directional, and spot lights, local- versus infinite-viewer lighting, and so on; we simply scale the appropriate operations by the number of lights and count the total operations. The vertex processor drains a pre-transform cache of vertices, which we approximate as a perfect (no-miss) cache in our prototype model.

To summarize, Qsilver runs a cycle-timer model that steps cycle-by-cycle; during each cycle every functional unit either advances in its computation, possibly producing an output for the next stage and incurring the cost of that computation's operations, or stalls as it waits on a queue or cache. By analyzing the activity of the various stages of the cycle-timer model over time, we can study the performance and bottlenecks of the system; by augmenting this information with a power model we can examine the power characteristics of the system. The simulation advances relatively quickly by architectural simulation standards; a simulated frame in our Enemy Territories study requires approximately 10-15 seconds to process on a 1.7GHz AMD Athlon with 512 MB RAM. This enables rapid exploration of design parameters such as the studies presented in Section 5.

4. Modeling power

To model power in Qsilver, we estimate the power for primitive operations like cache accesses, queue accesses, floating-point operations, register accesses, etc. Then, by identifying the primitive operations comprising a functional block in our pipeline, we can construct a power estimate for it.

To estimate the power for each primitive operation, we extract the cost of similar operations from an industrial, architecture-level power model to which we have access. This model is based on circuit-extracted data for a 180 nm high-performance superscalar microprocessor. We then scaled these estimates to match the structure sizes and bit-widths used in Qsilver, and further scaled them to the ap-

© The Eurographics Association 2004.

propriate technology node according to CV^2f (power is proportional to CV^2f, where C is capacitance, V is voltage, and f is frequency. All of these scale with each successive generation of semiconductor technology). In our case, the current Qsilver prototype resembles an Nvidia GeForce 4 (but tuned to drive low framebuffer resolutions–see below), so we have chosen to model a 150 nm implementation running at 1.8 V and 300 MHz. Although there will certainly be differences in circuit-design style between a high-performance CPU and our GPU, the relative power cost among different operations in the high-performance superscalar CPU is likely to be a reasonable indicator of relative cost in the GPU. We also cross-checked the relative power dissipated in each primitive against both Wattch and a second industrial model; of course, these too are models for high-performance CPUs.

Although our current power-modeling approach is admittedly crude, it enables us to construct a prototype that demonstrates the potential benefits of modeling power in the GPU. Even though the prototype power model is clearly imprecise and not suited for detailed modeling, it allows us to explore basic energy-efficiency tradeoffs like the benefits of a more aggressive vertex or fragment engine, the ability of runtime techniques to reduce power in response to time-varying application behavior, and the affect of design tradeoffs on leakage power as it grows in importance with future technology generations.

The structures in the GPU fall into four major categories: cache, queue, register, and arithmetic. For the cache, queue, and register structures, after scaling the power value extracted from the reference model to account for CV^2f, we further scale it to account for the width, height, and number of ports in the GPU. For purposes of our prototype, we scale linearly with each dimension. For example, our reference model uses a direct-mapped, 64 KB, single-ported cache, while our texture cache has four ports. After scaling for technology, we therefore account for the extra ports by multiplying by four.

The data-processing units—for example, the vertex transform-and-light unit—are less straightforward. We deal with these units by assuming they are driven either by microcode or a finite state machine and determine the primitive operations they execute, such as dot products, multiply-adds, etc—for this, we use the collection of compiled assembly code for the model of the fixed-function vertex processor. We then further break these down into basic floating-point multiplies and adds wherever possible (the main exception being transcendental functions). In the case of the vertex transform-and-light unit, we assume that because these operations are so complex, it is not pipelined, but rather it consists of a floating-point multiply-add array and that each microcode operation makes one pass through this hardware. (This assumption is also the basis for our model of how many cycles it takes to traverse the transform-and-light stage.) Our reference power model is also based on a floating-point multiply-add array (double-precision in this case), so after scaling to account for CV^2f, we can simply scale the resulting power value to account for the number of bits involved in each microcoded step. We currently scale linearly with the number of bits in each operation, although this oversimplifies the actual power scaling. In addition to the cost of the arithmetic operations, we also account for the power dissipated in the microcode memory and in reading and writing the register file.

Finally, we augmented the power estimate for each block to include power due to leakage currents. Leakage current arises from the fact that in deep-submicron technologies, transistors are never fully cut off. At 150 nm, leakage already accounts for approximately 10% of total power dissipation (this assumes an operating temperature of approximately 100°C, since leakage is exponentially dependent on temperature). Non-ideal device scaling means that leakage is growing proportionally worse with each successive technology generation; the International Technology Roadmap for Semiconductors (ITRS) [18] projects that it will grow to 50% or more within a few years as we reach the 70 nm node. This is important because it means that units which are clock gated still dissipate substantial power, and so trying to improve energy efficiency by reducing switching activity and/or improving clock gating will be less helpful in future generations, while optimizations that put blocks to sleep (like disconnecting them from their voltage supply using sleep transistors) or reduce the operating voltage (as in the multiple-clock-domain results presented in Section 5.2.2, become more helpful.

Following the approach used in Wattch, we model leakage as a simple fraction of the unconstrained switching power. When a unit is idle and clock gated, instead of dissipating its associated dynamic power that has been derived from our reference model, it dissipates a technology-dependent fraction thereof. The appropriate leakage ratio is taken from ITRS, e.g., 10% for 150nm and 50% for 70nm.

Again, this power model is imprecise and can only serve as a prototype. We have made many guesses or simplifying assumptions about the internal microarchitecture of the GPU, unit latencies and power dissipation, and so forth; and our power estimates are derived from data for a high-performance microprocessor that may use different circuit-design styles than would be employed for a GPU. Just as developing a more detailed and accurate performance model is an important area for future work, so is developing a detailed and accurate power model.

Validating the current prototype model is difficult, because we have no reference against which to compare. We can however perform some sanity checks. First, we verified that power dissipation in each block tracks its activity factor. Next, we looked at the power density in each block. The two blocks dissipating the most power are the vertex engine and texture cache, both dissipating about 12 W at peak. Based

© The Eurographics Association 2004.

on the rough GeForce 4, 12mm × 12mm floorplan in Figure 2 [17], this corresponds to about 0.7 W/mm^2, which is reasonable. Though we might expect each block to have a similar peak power density, the fragment engine only dissipates about 4.5 W. This may seem surprising given that the fragment processor is typically considered a computational "hot spot", but we have tuned our theoretical pipeline for a relatively low-resolution 800×600 display–more appropriate for a game console architecture than the PC graphics chip we are using as a guiding floorplan. The lower resolution leads to a correspondingly lower overall arithmetic intensity, thus the apparent anomaly in power density. Similarly, the rasterizer only dissipates about 2.5 W. This is because our model of rasterizer operations does not include some computations: edge setup, primitive assembly, or clipping. Having verified that the relative power dissipation in each block is reasonable given the actions we are modeling, we also verified that total power dissipation, including blocks and activities we do not yet model, will be within a range of about 50-75 W.

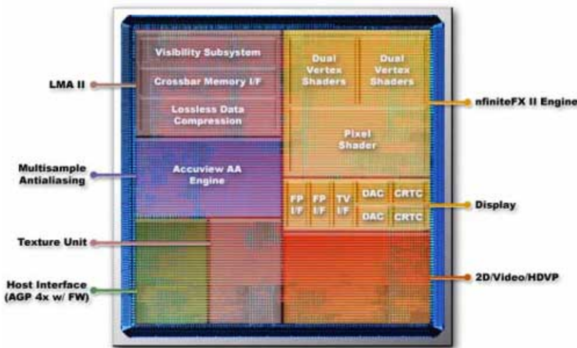

Figure 2: *GeForce 4 die photo from [17]. Note that the functional blocks in this marketing image are likely incorrect.*

5. Experiments and results

5.1. Performance analysis and experiment

One application of Qsilver is to analyze performance of a real application on a particular (perhaps hypothetical) architecture, with very fine granularity, enough to analyze intra- as well as inter-frame phenomena. For example, Figure 6 illustrates the evolution of a typical frame from *Enemy Territory*, along with timelines of some statistics from our model. Analysis of this timeline reveals interesting bottlenecks and behaviors. Images (a-d) are partially rendered frames at a certain cycle count; the ability to easily record such frames is a nice byproduct of using Chromium. In (a), the game is rendering a skybox using multiple passes, some textured and others untextured. As could be expected, the bottleneck is the fragment engine; vertex throughput is quite low (though notice that the textured pass uses additional polygons, perhaps so that fast-reject triangle culling will reduce rasterization load). In (b), the system has turned to rendering small details of trees and buildings; the vertex engine is now plateaued at maximum capacity and the game is transform limited. Shortly afterwards, (c) shows a sudden shift: the game is now rendering a road surface that contains few polygons, occupies much of the screen, and uses a MIP-mapped texture. Finally, in (d) the game is finishing the small head in the lower-left corner, which comprises many very small polygons and is thus heavily vertex-bound.

Of course, Qsilver can also be applied to the traditional task of microarchitectural simulators: performance studies. In the next section we describe a performance and power study based on varying the throughput of the fragment and vertex engines. One could also analyze performance while varying parameters such as the texture cache, queue lengths, memory architecture, and so on.

5.2. Experiment: power-aware graphics architecture

5.2.1. Energy-efficiency tradeoffs

To illustrate the value of Qsilver for power-aware design, we conduct two simple experiments in which we vary the throughput of the fragment and vertex engines to find the highest-performance and most energy-efficient design points (which are typically not the same points). Figure 3 plots the energy-delay-squared (ED^2) product [20], energy (E), and performance (frame rendering time, T) as a function of fragment processing rate. Figure 4 plots the same metrics as a function of vertex processing rate. All three metrics have been normalized to the results obtained with our default configuration: 5 cycles per tile for the fragment engine and 38 cycles per vertex for the vertex engine (no lighting is used in this experiment).

Clearly, the faster the engine, the better the throughput, but throughput reaches a point of diminishing returns at 4 cycles per tile in the fragment engine, at which point the power required to further improve performance is disproportionate to the miniscule marginal benefit. This makes 4 cycles/tile the energy-efficiency optimum, as shown by the ED^2 product. Note that this optimum differs slightly from the design point we initially chose, as might be expected in an early prototype model.

A similar trend is observed for vertex rate, where diminishing returns in terms of performance are not clearly present, but energy begins to rise disproportionately at the most aggressive throughput settings. The ED^2 optimum is therefore at 5-6 cycles/vertex.

These qualitative trends in these results are fairly insensitive to our power modeling assumptions, shifting the curves up or down on an absolute scale but leaving them essentially unchanged once normalized.

ED^2 has been established by the low-power design community as a sound figure of merit for trading off energy

© The Eurographics Association 2004.

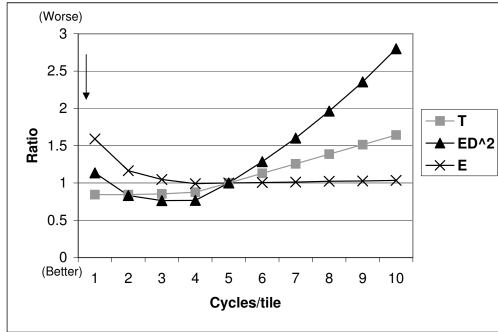

Figure 3: *Performance (T) and energy-efficiency data (ED^2 and E) for different fragment-processing rates. All results are normalized to our base case, which is 5 cycles/tile. Note also that in all cases, a smaller ratio is better.*

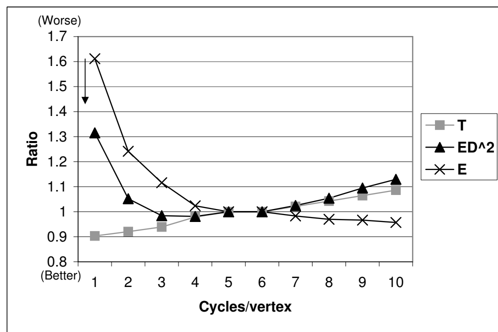

Figure 4: *Performance (T) and energy-efficiency data (ED^2 and E) for different vertex-processing rates. All results are normalized to our base case, which is 18 cycles/vertex.*

and performance. When considering the energy efficiency of different design choices, it is insufficient to simply look at average power, because $E = Pt$, and so if reducing average power comes at a cost of increased execution time, *more* energy may actually be consumed. Energy is also a poor metric when performance matters, since a reduction in energy only means that more power was saved than performance was lost. To cope with this, $E \cdot t$ has been proposed as a heuristic figure of merit, where the execution time t is also referred to as delay (D), hence the term "energy-delay product". This metric simply assumes that energy and performance are equally important. Better yet is the ED^2 product, which is more rigorous because it provides a metric of energy efficiency that is independent of the nominal supply voltage. It is therefore an accurate indicator of the unique contribution of microarchitectural or runtime power management that could not be achieved by simply choosing a different supply voltage at runtime. As an example, consider the the data above: if higher throughput is not needed—that is, the performance need not be any better than 1.0—then 4 cycles/tile is still the best configuration, because the chip's voltage can be reduced by 12%, yielding a 23% energy savings. The results presented here are only an illustration of the kind of tradeoff analysis that an infrastructure like Qsilver permits.

5.2.2. Multiple clock domains (MCD)

The notion of using multiple independent "clock domains" has been broached in both the GPU and CPU communities, e.g. [8, 14, 15] as a way to enhance clock speed by reducing the impact of clock skew and as a way to save power by allowing voltage and frequency to be dynamically reduced in clock domains that do not need to run at peak speed (dynamically changing voltage and frequency is often referred to as dynamic voltage scaling or DVS). In this paper, we focus on the power implications. Although MCD can provide substantial benefits, there are manufacturing and potential performance costs. MCD requires the ability to manufacture chips with independent voltage islands and independent clock domains, increasing testing costs; furthermore, each time the voltage and frequency settings are changed, the chip must stall while the voltage changes and the clock's phase-locked loop resynchronizes. A typical overhead for this is $10\,\mu$s.

In our GPU model, there are two natural clock domains—the vertex and fragment engines—with the fragment queue serving to decouple the two (see Figure 1). Since the GPU tends to alternate between being fragment and vertex bound, one or the other domain is typically stalled and can run at significantly lower voltage and frequency with minimal impact on overall fill rate. When the queue is full, this means the GPU is fragment bound, the vertex engine is effectively stalled, and it should reduce its voltage and frequency. Likewise, when the queue is empty, the GPU is vertex bound, the fragment engine is stalled, and it should reduce its voltage and frequency.

Specifically, we implemented a simple state machine that monitors the fragment queue. To avoid unnecessary changes in DVS setting and the associated overhead, the controller employs hysteresis by requiring the queue to stay within 10% of empty or full for 50,000 cycles before lowering the DVS setting by 10%. Then once the queue passes an appropriate high or low water mark to indicate that execution behavior is changing, the DVS setting is restored to its full value.

Figure 5 compares energy and performance both with and without MCD for three different leakage ratios. Again we use ED^2 as the figure of merit to trade off performance and energy and determine optimal energy efficiency. In fact, regardless of whether E or ED^2 is used, MCD is clearly energy efficient. For a 10% leakage ratio, MCD improves ED^2 by 11% with only a 1.5% performance loss. It achieves this by minimizing power dissipation when one or the other domain is stalled. Because leakage is exponentially dependent

© The Eurographics Association 2004.

on voltage, MCD becomes even more advantageous at future technology nodes where leakage is a greater fraction of total power dissipation. When a domain is stalled but in a low voltage and frequency state, not only is dynamic power reduced, but leakage power is dramatically reduced as well. With the leakage ratio at 50%, the ED^2 improvement reaches 30%. These represent energy savings of 13–30%.

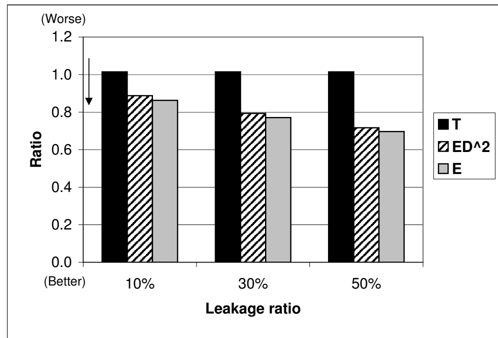

Figure 5: *Performance and energy-efficiency data of MCD for different leakage ratios. All results are normalized to the base case (with no MCD) for the appropriate leakage ratio. Smaller y-axis values are better, representing better performance and better energy efficiency.*

6. Discussion and limitations

Some tradeoffs accompany the decision to use Chromium, rather than writing or modifying a software rendering system such as Mesa [12], to generate the instrumented trace. Chromium's plug-in architecture makes it fast and simple to further annotate the stream by writing additional SPUs, or adding functionality to existing SPUs. We feel that the framework is quite flexible and, with some ingenuity, can incorporate a wide range of architectural experiments. For example, we plan in future work to explore the effects of Z_{min} culling [2] and Z_{max} or *hierarchical Z-buffer* culling [5, 11]. Typical hardware implementations of these ideas organize the depth buffer into 8×8 pixel tiles, and track the maximum or minimum depth, respectively, within the tile. If a given triangle is known to lie further than the maximum depth of a tile (for example, the distance to the triangle's nearest vertex is greater than that maximum depth), rasterization may be skipped in that tile; if the triangle's furthest vertex is closer than the tile's minimum depth, rasterization may proceed without requiring any Z-read operations.

To incorporate such tiled Z-buffer schemes, we propose to add another scratch buffer, with $1/8^{th}$ the linear resolution of the frame buffer, to the Qsilver annotator. This buffer stores at each pixel the maximum and minimum depth of the corresponding 8×8 tile of the framebuffer, and is updated after every primitive by running a min-max fragment program on the pixels of the scratch buffer affected by that primitive (easily accomplished by rasterizing a "fat" version of the primitive, or its bounding box, into the low-resolution scratch buffer). The min-max fragment program simply pools the pixels in the corresponding tile of the full-resolution depth buffer (bound as an input texture) and computes their min and max values. Given this low-resolution buffer, we can easily use an occlusion query and a special-purpose fragment program to render the primitive (at full resolution) and count how many fragments would pass the Z_{min} or Z_{max} tests.

Another motivation for using a hardware-accelerated annotation process via Chromium is speed: the use of hardware rasterization can be considerably faster than using an instrumented software renderer, typically in scenarios (high resolutions, full-screen stencil shadow buffer effects, etc) where the system is heavily fill-bound. Indeed, on our test platform a typical 50-frame trace from Enemy Territories runs 1.8× faster through the Qsilver annotator than through uninstrumented Mesa at 1280x1024, and 2.6× faster at 1600x1200.

It must be noted, however, that incorporating the annotations necessary to evaluate new architectural embellishments can be expensive, since every additional rendering pass and scratch buffer increases the number of pipeline flushes, context switches, and rendering calls. Furthermore, there are fundamental limitations to what data can easily be collected by our hardware-based annotator using fragment programs and occlusion queries. Our annotation approach is well-suited for collecting aggregate information: how many fragments get rasterized, pass the depth test, get textured with magnification versus minification, etc. While such information is sufficient for a wide range of studies, some experiments may require precise information about the positions or values of individual fragments. For example, it might prove difficult to study the effects of a Z-compression scheme in which memory bandwidth depended on the exact contents of the Z-buffer. In such a situation a user would likely wish to revert to instrumenting a software rasterizer, such as in Mesa, to track and collect the necessary data. Conveniently, Mesa implements the OpenGL API and can be incorporated directly into Qsilver's Chromium framework. This would allow the user to use the rest of the Qsilver framework and to re-use existing OpenGL traces, etc. It may even be possible to mix Mesa and hardware-accelerated OpenGL using different SPUs in Qsilver, for example running the slower Mesa rasterizer only for polygons or rendering intervals for which a particular architectural embellishment (e.g. a texture caching or compression scheme) was being evaluated.

7. Conclusions and future work

We have described Qsilver, a framework for power and performance analysis of graphics hardware. We have used Qsilver to develop a prototype power and performance model of a hypothetical GPU, and shown how this model allows:

© The Eurographics Association 2004.

- Fine-grained analysis of rendering behavior and bottlenecks, which can be used to optimize rendering systems.
- Architectural power and performance tuning, such as balancing the fragment-engine throughput to match the vertex engine's throughput.
- Evaluation of new architectural techniques for performance or power, such as Z_{min} culling or the use of DVS across multiple clock domains.

Our goal in developing Qsilver is to create a tool that will not only serve as a useful system for application performance analysis, but also stimulate research on GPU architecture and power-aware design in the same way that similar capabilities in the CPU community have fostered an explosion of research. It is important to note that we have designed Qsilver to be sufficiently flexible to model a wide array of architectures. We specifically wanted to ensure that it would be useful for modeling radical architectures to explore possible GPU architectures for future generations of graphics cards.

The next step in our work is refining our prototype to faithfully model a cutting-edge, programmable GPU. Then Qsilver makes possible a variety of interesting avenues for future work. For example, we would like to experiment with architectural innovations such as Delay Streams [1]. We are also particularly interested in modeling thermal effects. With a faithful performance and power model, and the addition of a more detailed floorplan, Qsilver can be augmented with a thermal model such as HotSpot [19] to dynamically simulate on-chip temperatures. This is an increasingly important area of research as high-performance GPUs become prohibitively costly to cool. For example, the tendency to oscillate between being vertex-bound and fragment-bound suggests that the GPU will in fact oscillate between thermal hotspots in these two regions. By smoothing out these hotspots spatially and temporally, temperature-aware design at the architectural level can significantly reduce cooling costs [16].

References

[1] T. Aila, V. Miettinen, and P. Nordlund. Delay streams for graphics hardware. *ACM Transactions on Graphics*, 22(3):792–800, 2003. 9

[2] T. Akenine-Möller and J. Ström. Graphics for the masses: a hardware rasterization architecture for mobile phones. *ACM Transactions on Graphics*, 22(3):801–808, 2003. 8

[3] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(4):59–67, Feb. 2002. 2

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000. 2

[5] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238. ACM Press, 1993. 8

[6] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ilp processors. *IEEE Computer*, 35(4):40–49, Feb. 2002. 2

[7] G. Humphreys, M. Houston, R. Ng, S. Ahern, R. Frank, P. Kirchner, and J. T. Klosowski. Chromium: A stream processing framework for interactive graphics on clusters of workstations. *ACM Transactions on Graphics*, 21(3):693–702, July 2002. 1, 2

[8] A. Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 158–68, May 2002. 7

[9] Y. Li, K. Skadron, Z. Hu, and D. Brooks. Understanding the energy efficiency of simultaneous multithreading. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, Aug. 2004. To appear. 2

[10] P. S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(4):50–58, Feb. 2002. 2

[11] S. Morein. ATI radeon HyperZ technology. Presentation at Workshop on Graphics Hardware, Hot3D Proceedings, ACM SIGGRAPH/Eurographics, 2000. 8

[12] B. Paul et al. The mesa 3-d graphics library, 1993–2004. http://www.mesa3d.org/. 8

[13] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995. 2

[14] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–67, Nov. 2002. 7

[15] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 29–40, Feb. 2002. 7

[16] J. W. Sheaffer, K. Skadron, and D. P. Luebke. Temperature-aware GPU design. Poster at ACM SIGGRAPH, Aug 2004. 9

[17] A. L. Shimpi. NVIDIA GeForce4 - NV17 and NV25 come to life, Feb. 2002. http://www.anandtech.com/print-article.html?i=1583. 6

[18] SIA. *International Technology Roadmap for Semiconductors*, 2001. 5

[19] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 2–13, Apr. 2003. 9

[20] V. Zyuban. Unified architecture level energy-efficiency metric. In *Proceedings of the 2002 Great Lakes VLSI Symposium*, pages 24–29, Apr. 2002. 6

© The Eurographics Association 2004.

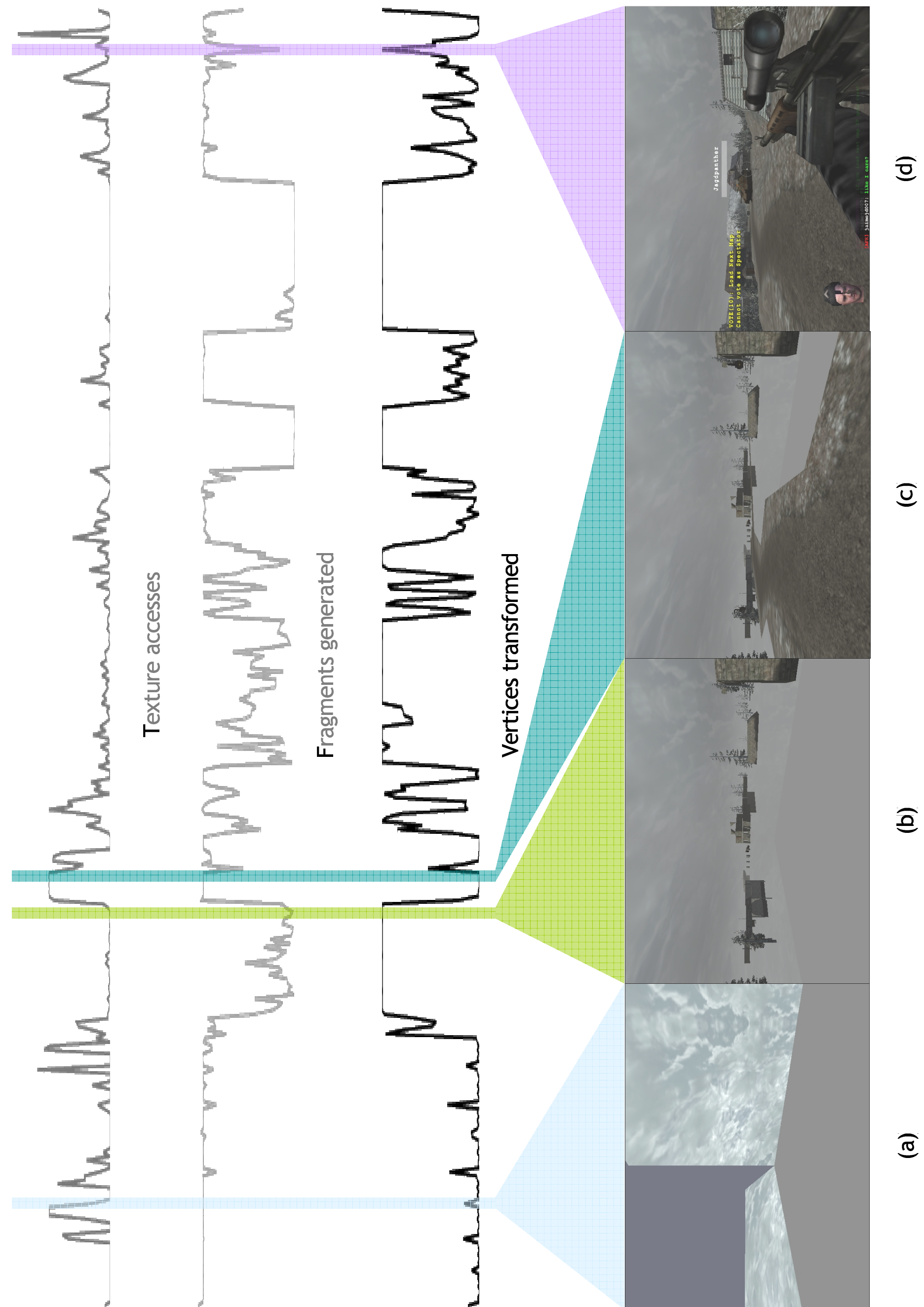

Figure 6: *The evolution of a frame in the* Enemy Territories *game.*

© The Eurographics Association 2004.