Graphics Hardware (2002), pp. 1–11
Thomas Ertl, Wolfgang Heidrich, and Michael Doggett (Editors)

Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware

Eric Chan Ren Ng Pradeep Sen Kekoa Proudfoot Pat Hanrahan

Stanford University[†]

Abstract

Real-time programmable graphics hardware has resource constraints that prevent complex shaders from rendering in a single pass. One way to virtualize these resources is to partition shading computations into multiple passes, each of which satisfies the given constraints. Many such partitions exist for a shader, but it is important to find one that renders efficiently. We present Recursive Dominator Split (RDS), a polynomial-time algorithm that uses a cost model to find near-optimal partitions of arbitrarily complex shaders. Using a simulator, we analyze partitions for architectures with different resource constraints and show that RDS performs well on different graphics architectures. We also demonstrate that shader partitions computed by RDS can run efficiently on programmable graphics hardware available today.

Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Graphics processors; G.2.2 [Mathematics of Computing]: Graph Algorithms, Trees

Keywords: programmable graphics hardware, multipass rendering, graph partitioning algorithms, shading languages

1. Introduction

Real-time programmable shading using mainstream graphics hardware has been an active area of research in recent years. Earlier generations of graphics hardware provided fixed-function pipelines designed for rendering texture-mapped triangles. In contrast, commodity graphics chips today support complex, user-programmable shading while maintaining high performance. This flexibility has encouraged the development of real-time shading languages that target these chips.

Shading languages evolved from the early work of Cook[5] and Perlin[14]. The RenderMan Shading Language is commonly used today for movie production-quality shading in software rendering systems[8]. Olano and Lastra described pfman, the first shading language that targets graphics hardware for real-time rendering[12]. Peercy et al. proposed a method for mapping shading languages to multiple rendering passes on non-programmable commodity graphics hardware[13]. Proudfoot et al. described a system that maps a shading language to programmable graphics hardware using a retargetable compiler back end[15]. These last three systems are able to render high-quality shaders in real-time.

Graphics chips today provide user-programmable pipelines[11, 2]. These longer pipelines accomodate larger shaders with more sophisticated operations. However, this hardware still has a limited set of resources. Examples of such limits are:

- A fixed memory size for instruction storage, i.e. a maximum number of instructions.
- A fixed number of active textures, texture accesses, and texture dependencies.
- A fixed number of registers for storing temporary values.
- A fixed number of interpolants for storing vertex-to-fragment values.

With shading languages, it is easy to write large shaders that exhaust available resources and cannot be mapped to a single rendering pass. The hardware can be virtualized by partitioning the shader into multiple passes, where each pass is a subset of the entire computation that satisfies all resource constraints. Many such partitions exist, and it is desirable to find the one that renders most efficiently. We call this the Multipass Partitioning Problem (MPP).

Peercy et al. solved this problem for non-programmable graphics hardware by using dynamic programming. Their

[†] {ericchan, renng, psen, kekoa, hanrahan}@graphics.stanford.edu

system interally represents shading computations as directed acyclic graphs (DAGs). They first decompose a DAG into trees, then perform tree-matching to find a minimum-cost set of passes. This approach works well for non-programmable hardware, which supports only a small set of operations per pass. Proudfoot et al. observed, however, that tree-matching techniques are inadequate for mapping DAGs to programmable hardware. To address this issue, they developed a back end for their shading system specifically to target this hardware. However, they did not solve MPP for programmable hardware, so their back end can only handle shaders that map to a single rendering pass.

In this paper, we present Recursive Dominator Split (RDS), an algorithm that solves MPP for programmable graphics hardware. RDS uses dominator trees, a heuristic search, and a greedy merging strategy to approximate minimum-cost partitions. Using a simulator, we show that RDS finds partitions within 5% of optimal for different shaders on architectures with different resource constraints. We also demonstrate how RDS can be used with an existing shading system to partition and render a multipass shader on a programmable graphics card.

2. Overview

We designed RDS in the context of the Stanford Real-Time Programmable Shading System[15]. This system is illustrated in Figure 1. A shader enters the system as source code written in a high-level language. The compiler front end parses this code and generates an intermediate pipeline program split by computation frequency. A compiler back end maps the pipeline program to hardware rendering passes. In this paper we discuss back end modules that map fragment computations to multiple passes. Each back end performs instruction selection on the fragment portion of the pipeline program and builds a DAG of hardware-specific fragment operations. If the back end cannot map the DAG to a single rendering pass, it calls RDS to partition the DAG into multiple passes.

There are many possible partitions, so RDS uses a cost model to evaluate them. The model reflects performance characteristics of the target architecture, such as the cost of operations and per-pass overhead. The goal is to find the optimal solution to MPP, i.e. a partition with the minimum cost.

MPP is related to some well-studied graph partitioning and NP-optimization problems. However, MPP is sufficiently different that, as far as we have been able to determine, existing techniques cannot be easily adapted to solve it. For example, the problem of load balancing parallel applications can be formulated in terms of finding a balanced subdivision of a graph[4], but algorithms to solve this problem are not immediately applicable to MPP. This is because in the load balancing case, setting the number of desired graph cuts is fixed, whereas in MPP it is part of the solution. More fundamentally, graph partitioning algorithms tend to assume that partitions are disjoint, whereas MPP allows partitions with overlapping subregions. In fact, MPP solutions almost always contain overlapping regions, which correspond to recomputed operations.

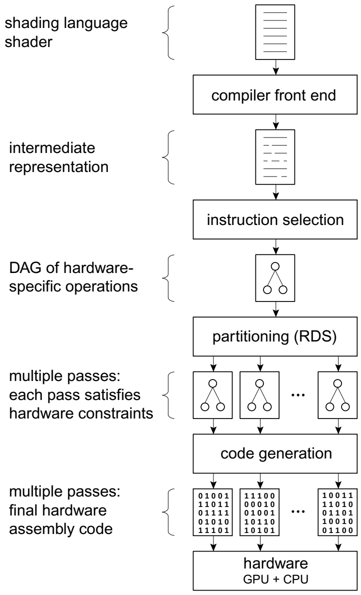

Figure 1: *System block diagram. The compiler front end converts a shader from shading language source code to an intermediate representation. The compiler back end performs instruction selection to build a DAG of hardware-specific operations. If the DAG is too large to be mapped to a single pass, RDS partitions the DAG into multiple passes. The compiler then generates assembly code for each of these passes and sends them to the hardware for rendering.*

Our solution to MPP is based on a number of architectural assumptions. We assume that architectures support only one 4-component output per pass. There may be many outstanding values at a time, but the framebuffer can only store one of them. Hence, intermediate results are usually preserved by copying the framebuffer contents to texture memory. Alternatively, render-to-texture can be used to avoid this expensive framebuffer copy. In either case, we say that intermediate results are *saved*. These values are *restored* in subsequent rendering passes via texture fetches. This save-and-restore technique relies on the following two assumptions:

- Architectures preserve intermediate values properly. For example, if the architecture uses floating point data types, then it must also support floating point buffers and textures to preserve high-precision intermediate results.
- Given b, the branching factor of the DAG, architectures support at least $b + 1$ operations, $b + 1$ texture units, b registers, 1 vertex interpolant, and 1 level of texture dependency per pass. This is the minimum set of resources required to support an arbitrary DAG via multipass rendering. For example, $b = 3$ if hardware instructions can have at most 3 operands.

Note that even with these assumptions, multipass rendering is an imperfect virtualization technique. It does not produce correct results for overlapping transparent geometry. This is a fundamental limitation of multipass rendering that could be overcome with changes to hardware[10].

Multipass rendering creates a per-pass overhead cost. This overhead arises from re-execution of the graphics pipeline (including transformation and rasterization of geometry), saving the results of a pass to texture memory, and restoring these results in later passes. Furthermore, extra passes may require more bounding box-sized or viewport-sized textures for saving intermediate results; these large textures consume GPU texture memory.

For these reasons, RDS attempts primarily to minimize the number of passes. Given a DAG of fragment operations, RDS first identifies nodes that are multiply-referenced. RDS then searches over these nodes and uses a heuristic to decide if the subgraphs rooted at these nodes should be saved in a separate pass or recomputed. It packs the remaining nodes into as few passes as possible using a greedy bottom-up merging algorithm. RDS optimizes both of these steps by using a dominator tree to group sufficiently small regions of the DAG together into a single pass. Minimizing the number of passes in this manner helps to minimize the overall cost.

3. Algorithms

We begin this section by formulating our problem and describing a simple and optimal solution to MPP. However, in practice this algorithm is intractable because it exhaustively searches a large space of possible partitions. We then identify the key issues of MPP by studying the structure of the problem. Understanding these issues helps us closely approximate the minimum cost without having to search over the whole space of partitions. Finally, we propose an $\mathcal{O}(n \cdot g(n))$ algorithm called RDS$_h$ and an $\mathcal{O}(n^2 \cdot g(n))$ algorithm called RDS, where $g(n)$ is the cost of checking if a set of n nodes can be mapped to one pass.

3.1. Preliminaries

We can represent the space of possible partitions by *marking* nodes to indicate pass boundaries. More precisely, we mark a node if and only if the node is the root of a pass. The number of marked nodes equals the number of passes in the partition. An example is shown in Figure 2.

The *subregion* of a node N is the set of nodes including N and recursively all unmarked children of N. For example, the subregion of node F in Figure 2b is $\{F, G, I, K, L\}$. A subregion is *valid* if it can be mapped to one pass; otherwise it is *invalid*.

We now describe a simple and optimal solution to MPP: examine the entire space of partitions, evaluate the cost of each partition, and keep the one with the lowest cost. Suppose there are n nodes in the DAG. Since each node may be marked or unmarked, there are 2^n unique ways to mark the nodes; each of these yields a possible partition. This exhaustive algorithm is clearly intractable because the partition space grows exponentially with the size of the DAG.

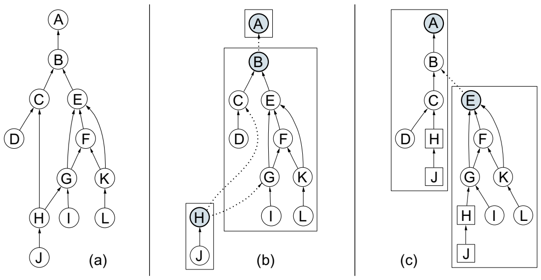

Figure 2: *A DAG with root A is shown in (a). We* mark *nodes to indicate the tops of passes. For example, by marking nodes A, B, and H, we split the DAG into three passes as shown in (b). On the other hand, by marking nodes A and E, we split the DAG into two passes as shown in (c), which causes nodes H and J to be recomputed. Marked nodes are shaded, and recomputed nodes are squares.*

3.2. Greedy Bottom-Up Merging

Since searching exhaustively is intractable, we propose a greedy bottom-up algorithm that merges nodes into as few passes as possible. We make a postorder traversal of the DAG that applies the following steps at each node N:

```
Merge(node $N$)
$k \leftarrow$ the number of kids of $N$
for $d \leftarrow k$ down to 0
do for each subset $S$ of $N$'s kids with $d$ kids
    do try to merge $N$ with all subregions of the kids of $S$
        if exactly one subset can be merged with $N$
            then pick that subset and stop
            else  if two or more subsets can be merged with $N$
                    then use MERGE heuristic to pick one
```

The algorithm is greedy in the sense that it starts from the largest possible merge and only considers progressively smaller subsets when necessary. Sometimes, there is more than one subset of a given size that can be merged. For example, suppose a node has two children and that it can be merged with either the left child or the right child, but not both. We then use a hardware-specific heuristic called MERGE to choose one of two possible merges. In principle, MERGE should pick the subset of children that uses the fewest resources, since this leaves the most room for additional nodes to be merged with this pass. Sometimes it is unclear which pass consumes the fewest resources. This can occur, for example, if one pass uses 5 interpolants and 3 textures, but another pass uses 3 interpolants and 5 textures. Our implementation of MERGE breaks these ties arbitrarily.

3.3. Save vs. Recompute

Some nodes in the DAG are referenced more than once; we call these multiply-referenced (MR) nodes. Subregions of these nodes may be saved or recomputed. For example, the subregion of MR node H is saved in Figure 2b but recomputed in Figure 2c. Always saving is undesirable because each save creates an additional pass. However, always recomputing is also undesirable, since it could lead to an explosion in the number of recomputed operations. It is unclear which choice will lead eventually to the best partition. Intuitively, we should save if the subregion is "full" and recompute if the subregion is "empty" relative to the architecture's available resources.

Both saving and recomputing involve a cost, but if we can map all of the references to a MR node to a single pass, then we can avoid these costs. We can detect these cases by identifying the immediate dominators of MR nodes. Intuitively, the immediate dominator B of MR node A is the node "closest" to A such that all the references of A go through B[1]. If subregion(B) and subregion(A) can be mapped together to a single pass, then we can avoid the save vs. recompute decision for node A.

Since we are interested in MR nodes and their immediate dominators, we would like to store them in a convenient data structure. We use a data structure that we call a *partial dominator tree* (PDT), which in turn is constructed from the dominator tree of a DAG. In a dominator tree, the parent of each node is its immediate dominator. This structure is a tree as opposed to a DAG because each node except the root has a unique immediate dominator[1]. A PDT is obtained from a dominator tree by discarding all nodes except MR nodes, their immediate dominators, and the root. This construction is illustrated in Figure 3.

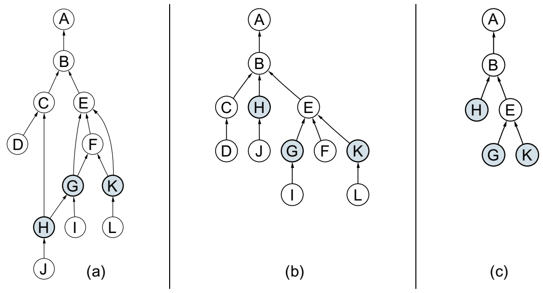

Figure 3: *A DAG is shown in (a) with multiply-referenced nodes drawn shaded. The dominator tree for this DAG is shown in (b). Node B is the parent and therefore the immediate dominator of H. Similarly, E is the immediate dominator of both G and K. The partial dominator tree in (c) is obtained from the tree in (b) by keeping only the multiply-referenced nodes, their immediate dominators, and the root.*

3.4. RDS$_h$

In this section, we describe an algorithm called RDS$_h$. We apply the ideas of the previous section by mapping multiply-referenced subregions and the subregions of their immediate dominators to the same pass. Since this isn't always possible, we break the problem down by recursively subdividing the DAG into smaller regions using the PDT. When necessary, we perform greedy bottom-up merging within these regions and evaluate save vs. recompute decisions using a heuristic. Pseudocode for the algorithm is shown below.

```
RDS$_h$ (DAG $D$)
    $P' \leftarrow$ the root node of the PDT of $D$
    Subdivide ($P'$)

Subdivide(PDT node $T'$)
1  if subregion($T$) is invalid
2    then $L \leftarrow$ the list of children of $T'$ ordered to maintain
3              DAG dependencies
4         for each element $K'$ of $L$
5         do Subdivide($K'$)
6            if $K$ is a MR node
7              then use RECOMPUTE heuristic to decide
8                     if $K$ should be saved or recomputed
9         apply greeding merging to subregion($T$)
```

In this pseudocode, we use the following convention to describe the relationship between nodes in the DAG D and its PDT P. If $N \in D$ and $N \in P$, we refer to the node in D as N and refer to the node in P as N'. In other words, N and N' represent the same node, but in different structures. This distinction is subtle but important. For example, the children of N are nodes in D, whereas the children of N' are nodes in P.

The RDS$_h$ algorithm takes an unmarked DAG and calls Subdivide to partition it. The Subdivide procedure marks nodes to indicate pass boundaries as described in Section 3.1. The algorithm first checks at each node if its subregion is small enough to map to one pass (line 1). If this check fails, the problem is broken down into recursive subdivisions of each child (line 5). We subdivide children in an order that maintains DAG dependencies, which is the same as the order given by a postorder traversal of the DAG. For example, in the PDT shown in Figure 3c, nodes H and E are both children of B. Since H always appears before E in a postorder traversal of the DAG shown in Figure 3a, we subdivide H first, then E.

After subdividing each child that is multiply-referenced, we use a heuristic called RECOMPUTE to decide if that child's subregion should be saved or recomputed (lines 6-8); saved children are marked. In principle, subregions that use few resources should be recomputed, whereas those that consume most of the available resources should be saved. We implement RECOMPUTE by choosing to recompute a set of nodes if and only if the consumption of each resource is less than one-half the maximum allowed. However, this heuristic can be replaced with one that is more specific to a given architecture.

Finally, after all children have been subdivided, we apply the greedy merging algorithm described in Section 3.2 to the current subregion (line 9). During this step, nodes that cannot be merged with their parents are marked.

The Subdivide procedure makes only one traversal through the PDT, but at each node it checks the validity of its subregion. If $g(n)$ is the cost of this check, then the overall running time of RDS$_h$ is $\mathcal{O}(n \cdot g(n))$, where n is the size of the DAG. In our implementation, the target architecture's compiler makes each validity check by generating code and allocating resources for the subregion. This can be done in linear time, so our implementation of RDS$_h$ runs in $\mathcal{O}(n^2)$ time.

3.5. RDS

RDS$_h$ uses a simple heuristic to make save vs. recompute decisions. However, correct decisions are difficult to make because they are interdependent. For example, suppose we have two MR nodes A and B such that A depends on B. If we save B to a separate pass, then the subregion of A becomes smaller and may be worth recomputing. On the other hand, if we recompute B, then the subregion of A becomes larger and may be worth saving. It is difficult for a simple heuristic to predict which of these two choices will eventually lead to the minimum-cost partition.

One way to address this problem is to search over the MR nodes exhaustively and try all possible save/recompute configurations. However, this increases the overall running time by a factor of 2^k, where k is the number of MR nodes in the DAG. Instead, we propose a less expensive alternative called RDS that combines a limited search with the existing RECOMPUTE heuristic.

The pseudocode for RDS is shown below.

```
RDS (DAG $D$)
    $P' \leftarrow$ the root node of the PDT of $D$
    Search($D$, $P'$)

Search(DAG $D$, PDT node $P'$)
 1  $L \leftarrow$ list of the MR nodes of $D$ ordered to maintain
 2       DAG dependencies
 3  for each node $M$ in $L$
 4  do unmark all nodes of $D$
 5     fix $M$ as marked    # save subregion($M$)
 6     $p_s \leftarrow$ the partition computed by Subdivide($P'$)
 7     $c_s \leftarrow$ COST($p_s$)
 8
 9     unmark all nodes of $D$
10     fix $M$ as unmarked    # recompute subregion($M$)
11     $p_r \leftarrow$ the partition computed by Subdivide($P'$)
12     $c_r \leftarrow$ COST($p_r$)
13     if $c_s < c_r$ then fix $M$ as marked
```

We also replace lines 6–8 of the Subdivide procedure with:

```
14  if $K$ is a MR node
15  then if $K$ is fixed as marked, then mark $K$
16       else if $K$ is fixed as unmarked, then unmark $K$;
17       else use RECOMPUTE heuristic to decide if
18            $K$ should be saved or recomputed;
```

In the Search procedure, all MR nodes are initially *unfixed*. Each iteration of the loop makes a save vs. recompute decision at just one MR node. At each node M, we use the Subdivide algorithm to produce two partitions: one that results when subregion(M) is saved (line 6), and another that results when it is recomputed (line 11). In other words, in each case we have already determined whether subregion(M) will be saved or recomputed before subdivision begins. The code represents this by *fixing* M as marked or unmarked (lines 5 and 9). In both cases, we use the RECOMPUTE heuristic described above to make save vs. recompute decisions at the remaining unfixed MR nodes (lines 17–18). We then evaluate both partitions using a given cost model (lines 7 and 12) and decide to save or recompute M based on which partition has the lower cost (line 13).

The RDS algorithm wraps a search around the Subdivide procedure. The search calls Subdivide $2k$ times, where k is the number of MR nodes. By the analysis in the previous section, subdivision has complexity $\mathcal{O}(n \cdot g(n))$. Since k is typically proportional to n, the overall running time of RDS is $\mathcal{O}(n^2 \cdot g(n))$.

3.6. Analysis

We have described two versions of an algorithm. The first one, RDS$_h$, uses only heuristics to make merging and recompute decisions and has complexity $\mathcal{O}(n \cdot g(n))$, where $g(n)$ is the cost of the validity check on a subregion of size n. In our experiments, we found that a simple heuristic is inadequate for making save vs. recompute decisions. Thus we described a second version, RDS, that uses a limited search to make these decisions but has complexity $\mathcal{O}(n^2 \cdot g(n))$.

In our implementation, a validity check involves generating code and allocating resources for the subregion. The check has complexity $g(n) = \mathcal{O}(n)$, so RDS$_h$ and RDS have complexity $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively. The actual running time depends on the resource consumption of the given shader and the resource constraints of the target architecture. When resources are extremely limited, the validity check in line 1 of the Subdivide procedure fails most of the time, which leads to further traversal of the PDT. On the other hand, when resources are less constrained, the validity check usually succeeds and terminates the recursion.

4. Implementation

In the rest of this paper, we focus on RDS. We implemented RDS and integrated it with the Stanford Real-Time Programmable Shading System[15]. The integrated system is shown in Figure 1.

We evaluated this system by developing two fragment compiler back ends. The first one targets the ATI Radeon 8500 architecture[2]. This architecture exposes a custom set of OpenGL extensions. We queried the software driver using these extensions to determine the hardware's resource constraints. The hardware is limited to 16 operations, 6 registers, 6 texture units, 6 interpolants, and 1 level of texture dependency. Since the hardware provides one output value per pass, we use render-to-texture to spill intermediate values to texture memory. However, floating point buffers and textures are unsupported, so these values are not preserved at full precision. To circumvent this issue, we limited our tests on this architecture to shaders whose intermediate values are already clamped to the range $[0, 1]$.

The Radeon 8500 is one example of a programmable fragment architecture. To evaluate RDS on architectures with different resource constraints, we wrote a second back end that compiles shaders to a generic programmable fragment pipeline. The maximum number of per-pass operations, registers, active texture units, and interpolants can be configured; we call each setup a *pipeline configuration* (PC). Our simulated architecture supports 4-component floating-point vectors and one output value. It uses the NVIDIA vertex program instruction set[9], with the addition of texture operations. The architecture places no limits on the number of texture dependencies within a fragment program. In short, each PC provides the basic data types and instruction set necessary to support fragment shaders, but imposes four per-pass resource constraints.

Both compiler back ends incorporate the following three elements to support RDS:

1. Before partitioning, the compilers perform instruction selection to build a DAG of fragment operations.
2. To support partitioning, the compilers provide a common interface to RDS that exposes a cost model and hardware-specific resource constraints.
3. After partitioning, the compilers order the passes and assign them to textures.

To perform instruction selection, both of our back ends use a modified version of *lburg*[6]. We extended lburg to support operators with arbitrary arity and wrote covering rules to map the shading system's intermediate representation directly to hardware operations. Note that unlike Peercy and Proudfoot, we only used lburg's tree-matching capabilities for selecting operations, not for mapping operations to rendering passes.

The interface between RDS and the back ends consists of four callback functions:

1. VALID, a function that determines if a given set of nodes can be mapped to a single pass. This is needed to ensure that each pass satisfies the hardware's resource constraints. Our back ends implement this check by generating code and allocating resources as needed. The check fails if any part of the resource allocation fails.
2. COST, a function that computes the cost of a given partition using a cost model. It is called during the search algorithm described in Section 3.5. The cost of a partition depends on several factors, including the number of passes p, the number of texture accesses t, and the number of non-texture fragment instructions i. We use a simple linear cost model:

$$\text{cost} = c_p p + c_t t + c_i i,$$

   Note that the costs c_t and c_i are charged on a per-fragment basis, so the total cost of texture accesses and non-texture instructions is proportional to the number of rendered fragments. On the other hand, c_p is the overhead of an entire pass, so we can think of c_p as the per-pass cost amortized over all the fragments. For our ATI back end, the multipass renderer preserves intermediate results by saving the entire framebuffer with render-to-texture. Saves can also be implemented using copy-to-texture, in which case c_p depends on the size of the viewport.
3. RECOMPUTE, a heuristic function that decides whether or not to recompute a set of nodes. This is called during the search algorithm described in Section 3.5. In our back ends, we choose to recompute a set of nodes if and only if the consumption of each resource is less than one-half the maximum allowed.
4. MERGE, a heuristic function that, given a set of passes, picks the one that consumes the fewest resources. This is called during the merging algorithm described in Section 3.2. Sometimes it is unclear which pass consumes the fewest resources; our back ends break these ties arbitrarily.

RDS uses these callback functions to query a compiler back end for hardware-specific information. This design allows RDS to target any architecture whose compiler back end implements these callbacks. Furthermore, it requires minimal changes to our existing compiler infrastructure.

After partitioning is complete, compiler back ends must order the passes and assign textures to store intermediate results. Ideally, these textures should be assigned in a way that minimizes the number of textures needed. Assigning textures is similar to register allocation, so we applied graph coloring techniques as described by Chaitin[3].

5. Results

5.1. Example and System Demonstration

We now demonstrate how our ATI Radeon 8500 compiler uses RDS to partition a shader into multiple passes so that it can be rendered in real-time. Our shader is a version of the RenderMan bowling pin surface shader combined with five light shaders: one point light source and four animated textured lights. Figure 4 shows the source code for this shader written in the Stanford Real-Time Shading Language. The compiler front end maps this source code to a hardware-independent intermediate representation. Next, our Radeon 8500 back end performs instruction selection and builds the DAG in Figure 5a. Since this DAG is too large to map to a single pass, the compiler calls RDS to partition the DAG into multiple passes. Figure 5b shows the resulting partition, which contains 7 passes, 12 texture fetches, and 30 non-texture operations. We ran the partitioned shader at a resolution of 512×512 on a 1.4 GHz Pentium 4 system with an ATI Radeon 8500. The system renders the shader at 30 frames/sec and produces the image shown in Figure 7 (see color plates).

5.2. Testing Methodology

In this section, we discuss the techniques that we used to evaluate the efficiency of RDS. We chose three shaders, seven pipeline configurations of our simulated architecture, and five cost models.

```
surface shader float4
bowling_pin (texref pinbase, texref bruns, texref circle,
               texref coated, texref marks, float4 uv)
{
   float4 uv_wrap = { uv[0], 10 * Pobj[1], 0, 1 };
   float4 uv_label = { 10 * Pobj[0], 10 * Pobj[1], 0, 1 };
   matrix4 t_base = invert(translate(0, -7.5, 0) * scale(0.667, 15, 1));
   matrix4 t_bruns = invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
   matrix4 t_circle = invert(translate(-0.8, -1.15, 0) * scale(1.4, 1.4, 1));
   matrix4 t_coated = invert(translate(2.6, -2.8, 0) * scale(-5.2, 5.2, 1));
   matrix4 t_marks = invert(translate(2.0, 7.5, 0) * scale (4, -15, 1));
   float front = select(Pobj[2] >= 0, 1, 0);
   float back = select(Pobj[2] <= 0, 1, 0);
   float4 Base = texture(pinbase, t_base * uv_wrap);
   float4 Bruns = front * texture(bruns, t_bruns * uv_label);
   float4 Circle = front * texture(circle, t_circle * uv_label);
   float4 Coated = back * texture(coated, t_coated * uv_label);
   float4 Marks = texture(marks, t_marks * uv_wrap);
   float4 Cd = lightmodel_diffuse({ 0.4, 0.4, 0.4, 1 }, { 0.5, 0.5, 0.5, 1 });
   float4 Cs = lightmodel_specular({ 0.35, 0.35, 0.35, 1 }, Zero, 20);
   return (Circle over (Bruns over (Coated over Base))) * (Marks*Cd) + Cs;
}
```

Figure 4: *RTSL source code for the bowling pin surface shader.*

Data for the shaders are listed in Table 1. The first shader (RBP) is the version of the RenderMan bowling pin described above. RBP has modest computation but requires several vertex interpolants and texture units. The second shader (BMBP) applies a bump map to the bowling pin surface and uses only one point light source. BMBP uses a more balanced set of resources than RBP. The third shader (Wood) procedurally generates a wood texture[7]. It is computationally intensive and requires many dependent texture lookups, but uses other resources modestly. We chose these three shaders for testing because they stress different resources.

Eight pipeline configurations are listed in Table 2. PCs 1–4 are limited in only one resource; these allow us to show that RDS finds good partitions on architectures constrained in different ways. PCs 5–7 have more balanced constraints; they represent evolving architectures that provide more and more resources to fragment pipelines. We use PCs 5–7 to show that RDS performs well when multiple resources are constrained. PC 8 has unlimited resources and is useful for comparing the cost of a single-pass partition to the cost of multiple passes.

For convenience, we will use the notation $O_o/R_r/T_t/V_v$ to refer to architectures' resource constraints, where O is the number of operations, R is the number of registers, T is the number of texture units, and V is the number of interpolants. For instance, PC 5 has constraints $6_o/4_r/4_t/4_v$.

We chose a simple, linear cost function $c_p p + c_t t + c_i i$, as discussed in Section 4. For the ATI Radeon 8500, we estimated the coefficients for each term by profiling the hardware as follows. We performed all our measurements by rendering a screen-aligned square into a 512×512 window. First, we compared the rendering times of shaders that differed only in the number of non-texture fragment instructions. We then computed $c_i = \Delta t_i / f \Delta i$, where f is the number of fragments, Δi is the controlled change in the number of non-texture instructions, and Δt_i is the measured change in rendering time. Our measurements of c_i remained constant when we varied f by resizing the square, as expected. We computed c_t similarly. To estimate c_p, we reduced the square to one pixel in size to make the cost of fragment operations negligible. After measuring the three coefficients, we normalized them by c_i to obtain:

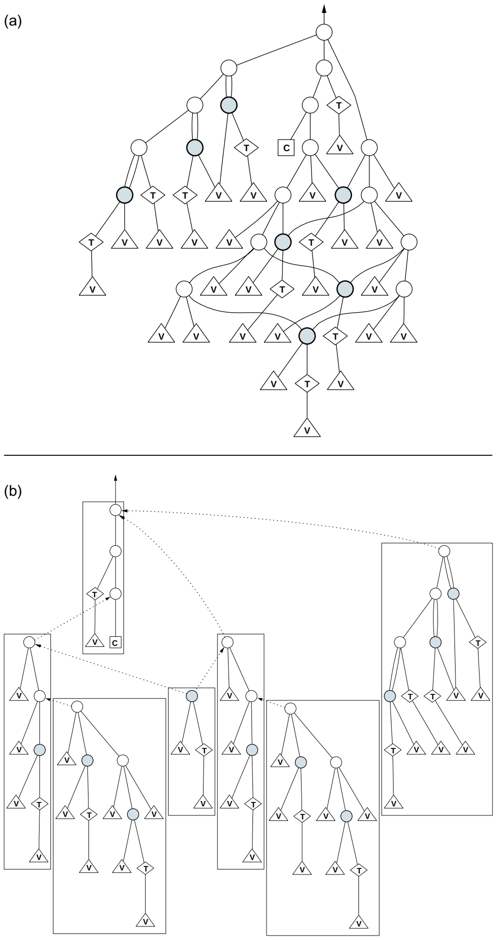

Figure 5: *(a) DAG of the RBP shader, after instruction selection. (b) A partition of the shader with 7 passes, computed by RDS for the ATI Radeon 8500 architecture. Texture fetches (T), vertex interpolants (V), and constants (C) are shown as diamonds, triangles and squares, respectively. Circles represent other fragment operations, and multiply-referenced nodes are shaded. Dotted edges indicate dependencies between passes.*

$$\text{cost} = 15.7p + 1.36t + i$$

While the result is a rough estimate, it is clear that per-pass overhead is high relative to individual fragment operations. This supports our assumption that per-pass overhead dominates the overall cost.

For our simulated architecture, we also tested RDS with the following cost models:

Shader	RBP	BMBP	Wood
# total nodes (n)	68	72	**475**
# MR nodes (k)	11	14	**72**
Non-texture operations	27	33	**175**
Texture operations	9	3	**33**
Total operations	36	36	**208**
Registers	7	5	**16**
Texture units	**9**	3	6
Interpolants	**24**	6	5

Table 1: *Shaders. Resource consumption on our simulated architecture is listed for each shader. The shader with the largest consumption in each category is shown in boldface. The last four rows contain the primary resource constraints. For example, in order for RBP to compile to a single pass, our pipeline configuration must support at least 36 operations, 7 registers, 9 texture units, and 24 interpolants.*

Architecture	o	r	t	v
PC 1 (operation-limited)	6	∞	∞	∞
PC 2 (register-limited)	∞	4	∞	∞
PC 3 (texture-limited)	∞	∞	4	∞
PC 4 (interpolant-limited)	∞	∞	∞	4
PC 5	6	4	4	4
PC 6	24	8	8	8
PC 7	128	12	16	12
PC 8 (unlimited)	∞	∞	∞	∞

Table 2: *Architectures. Four resource constraints are given for each architecture: operations (o), registers (r), texture units (t), and vertex interpolants (v).*

$$\begin{aligned} 15p + 5t + i \\ 5p + 3t + i \\ 3p + 2t + i \\ p + t + i \\ t + i \end{aligned}$$

We picked different coefficients to show that RDS performs well for architectures with different performance characteristics. Note that the last cost model $t + i$ completely ignores per-pass overhead.

5.3. Efficiency

For comparison, we implemented the algorithm in Section 3.1 that uses exhaustive search to find the optimal partition. It is interesting to note that in all of our tests, RDS always found a partition with the same number of passes as the optimal partition. However, there is a difference in cost because RDS picked different pass boundaries, which can affect the total number of restore operations needed.

Table 3 compares the partitions computed by RDS using the cost model $15p+5t+i$ to the optimal partitions. In some cases, the search space was too large for the exhaustive algorithm to finish. We measure the efficiency of RDS by computing the percentage increase in cost of the RDS-generated partition over the optimal partition. There are 17 complete test cases not including PC 8. RDS found a minimum-cost partition in 14 of these cases, and was within 5% of optimal cost in each of the remaining cases.

Results for the other cost models are similar. RDS found an optimal partition in two-thirds of all the test cases. For the remaining cases, RDS was within 5% of optimal on average and within 15% in the worst case. The worst cases occurred when using the cost models $p+t+i$ and $t+i$. This is not surprising, since RDS was designed under the assumption that per-pass overhead dominates the overall cost. Nonetheless, these results indicate that RDS performs consistently well across different cost models.

5.4. Speed vs. Quality Tradeoff

In practice, one can imagine a knob that allows users to trade off speed for partition quality. As an example, we compare RDS to RDS$_h$ in Table 4. We measure the efficiency of RDS$_h$ as a percentage increase in cost over RDS. Over the 21 cases, the average cost of RDS$_h$ is 10.5% higher. Since RDS$_h$ uses only heuristics and RDS performs a limited search, it is not surprising that RDS$_h$ runs faster than RDS at the expense of quality.

Both versions of the algorithm may be useful in practice. For example, a fast partitioning scheme such as RDS$_h$ can be used during iterative development of new shaders. Once the shader is complete, a slower but more thorough algorithm such as RDS is used to produce efficient partitions, which in turn will improve rendering performance.

5.5. Cost Analysis

In this section, we study the cost of saves and restores relative to the total cost of a partition. Table 5 compares two partitions of the RBP shader. The 11-pass partition contains 53 operations, of which 11 are restores. In contrast, the 3-pass partition contains 44 operations, of which only 2 are restores. Thus all 9 of the extra operations in the former partition are due to restores. This overhead occurs because the partition simply requires more intermediate values.

As the number of passes increases, the additional costs come primarily from save and restore overhead. Figure 6 shows the cost breakdown for seven partitions of the RBP shader, each generated by RDS. These partitions are the same as the ones listed in Table 3, but they are now arranged in order of increasing number of passes. The cost of non-texture operations and non-restore texture fetches remain nearly constant across the graph; the slight variations arise from recomputation. In contrast, the cost of saves and restores rises steadily. This suggests that rendering performance could be improved by changes to hardware that use more efficient methods for saving and restoring intermediate results.

6. Discussion

In this section we discuss limitations of RDS and future work.

Architecture	RBP					BMBP					Wood				
	Optimal		RDS		Eff.	Optimal		RDS		Eff.	Optimal		RDS		Eff.
	c	p	c	p	+%	c	p	c	p	+%	c	p	c	p	+%
PC 1: $6_o/\infty_r/\infty_t/\infty_v$	309	11	309	11	0.0	310	12	310	12	0.0	–	–	2685	91	–
PC 2: $\infty_o/4_r/\infty_t/\infty_v$	125	2	131	2	4.8	109	3	109	3	0.0	–	–	1462	32	–
PC 3: $\infty_o/\infty_r/4_t/\infty_v$	171	4	171	4	0.0	63	1	63	1	0.0	378	2	378	2	0.0
PC 4: $\infty_o/\infty_r/\infty_t/4_v$	269	9	269	9	0.0	84	2	84	2	0.0	374	2	374	2	0.0
PC 5: $6_o/4_r/4_t/4_v$	310	11	310	11	0.0	310	12	310	12	0.0	–	–	2681	92	–
PC 6: $24_o/8_r/8_t/8_v$	179	5	184	5	2.8	111	3	116	3	4.5	–	–	937	17	–
PC 7: $128_o/12_r/16_t/12_v$	145	3	145	3	0.0	63	1	63	1	0.0	396	3	396	3	0.0
PC 8: $\infty_o/\infty_r/\infty_t/\infty_v$	87	1	87	1	0.0	63	1	63	1	0.0	355	1	355	1	0.0

Table 3: *Efficiency comparison. For each case, results for both the optimal partition and the RDS-computed partition are given. All partitions were generated using the cost model $15p + 5t + i$. Column c shows the cost of the partition, and column p shows the number of passes in that partition. The efficiency of RDS is measured as the percentage increase in cost over the optimal cost. – indicates the search space for that configuration was too large to find an optimal partition.*

Architecture	RBP			BMBP			Wood		
	RDS	RDS$_h$		RDS	RDS$_h$		RDS	RDS$_h$	
	Time	Time	Eff. (+%)	Time	Time	Eff. (+%)	Time	Time	Eff. (+%)
PC 1: $6_o/\infty_r/\infty_t/\infty_v$	0.32	0.03	7.1	0.32	0.06	1.9	55.58	0.69	5.7
PC 2: $\infty_o/4_r/\infty_t/\infty_v$	0.15	0.02	0.0	0.07	0.04	0.0	53.27	0.78	16.1
PC 3: $\infty_o/\infty_r/4_t/\infty_v$	0.12	0.02	0.0	0.01	0.01	0.0	0.61	0.13	0.0
PC 4: $\infty_o/\infty_r/\infty_t/4_v$	0.21	0.02	8.2	0.05	0.02	52.4	0.88	0.31	0.0
PC 5: $6_o/4_r/4_t/4_v$	0.18	0.03	6.8	0.21	0.04	1.9	31.20	0.37	6.8
PC 6: $24_o/8_r/8_t/8_v$	0.09	0.02	3.8	0.17	0.03	10.3	39.32	0.58	0.7
PC 7: $128_o/12_r/16_t/12_v$	0.12	0.02	0.0	0.01	0.01	0.0	6.32	1.30	99.0
PC 8: $\infty_o/\infty_r/\infty_t/\infty_v$	< 0.01	< 0.01	0.0	0.01	0.01	0.0	0.07	0.07	0.0

Table 4: *RDS vs. RDS$_h$. Running times for both versions of the algorithm were measured on a 1.4 GHz Pentium 4 system running Windows 2000 SP2. All times are reported in seconds. The efficiency of RDS$_h$ is computed as a percentage increase in cost over RDS.*

Pass	PC 5: $6_o/4_r/4_t/4_v$					PC 7: $128_o/12_r/16_t/12_v$				
#	o	r	t	v	R	o	r	t	v	R
1	5	3	2	3	0	10	2	3	10	0
2	5	3	2	2	1	10	2	3	10	0
3	5	3	2	2	1	24	4	8	10	2
4	4	1	1	4	0	not applicable				
5	2	1	1	2	0					
6	6	2	3	4	2					
7	5	2	2	3	1					
8	5	3	3	1	2					
9	4	1	1	4	0					
10	6	2	3	4	2					
11	6	3	3	3	2					

Table 5: *Per-pass resource consumption for two partitions of the RBP shader. Column R gives the number of restores. The 11-pass and 3-pass partitions target PCs 5 and 7, respectively. Both partitions were computed by RDS.*

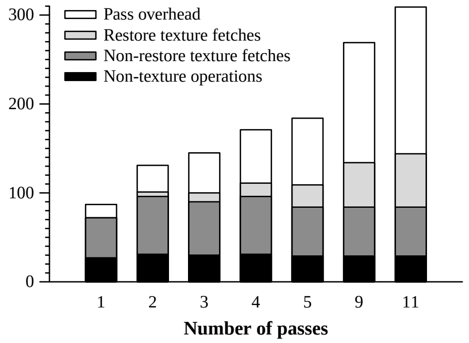

Figure 6: *Cost of partitions, broken down by type. These partitions were computed by RDS for the RBP shader with the cost model $15p + 5t + i$.*

In designing RDS, we have assumed that a shader can be represented as a single DAG, which is equivalent to one basic block of hardware assembly code. However, programmable graphics hardware will likely support loops and conditionals in the future. Since this requires flowgraphs with multiple basic blocks, we need additional techniques to handle branching correctly and efficiently.

We have assumed that programmable graphics hardware allows only one output value per pass. However, future hardware may support multiple outputs, so a single pass can compute several results. These computations may be unrelated and correspond to disjoint regions of the DAG. Efficient partitioning becomes more difficult because it requires examining many disjoint regions simultaneously. Our current algorithm cannot handle multiple outputs because it considers only one connected region at a time.

Accurate cost models are needed to enable RDS to find partitions that render efficiently. For that reason, RDS allows any cost model to be plugged in. However, we make some simplifying assumptions in our cost model. Some of these assumptions are necessary because we don't have all the relevant cost information at compile time. For example, the per-pass cost may depend on the viewport size, and the total cost of instructions depends on the number of rendered fragments. In our shading system, however, neither the viewport size nor the number of fragments are known at compile time, so we simply treat the coefficients c_p, c_t, and c_i as constants. On the other hand, some limitations can be addressed with more sophisticated cost models. For example, our current model ignores the cost of vertex shaders. If rendering performance is limited by vertex computations, then it is desirable to find a partition that minimizes recomputation of regions containing vertex inputs. This could be done by using a cost model in which the per-pass cost is proportional to the number of vertex operations needed by that pass.

We showed that RDS and RDS$_h$ are $\mathcal{O}(n^2 \cdot g(n))$ and $\mathcal{O}(n \cdot g(n))$ algorithms, but it may be possible to eliminate the $g(n)$ term. For simplicity and ease of implementation, we check validity by calling existing compiler subroutines. However, this requires that subregions be compiled from scratch every time, so $g(n) = \mathcal{O}(n)$. The check could be made less expensive by using incremental techniques, since a subregion depends only on previously checked subregions. One approach would be to use vectors to keep track of a subregion's resource consumption. Resource vectors from different subregions could be added to determine quickly if the subregions can be merged. However, overlapping subregions and peephole optimizations are tricky and must be treated carefully. Using incremental approaches, it may be possible to reduce the cost of the validity check to constant time when amortized over the traversal of the entire DAG. This would reduce the complexity of RDS and RDS$_h$ by a factor of n.

7. Conclusion

We have described RDS, an algorithm for partitioning fragment shaders into multiple passes. Our algorithm finds near-optimal partitions for a number of shaders on architectures with different limitations. Furthermore, RDS performs consistently well across a range of different cost models. We have integrated RDS with an existing programmable shading system and demonstrated how this system can be used to partition and render a large shader in real-time. Since RDS depends only on a flexible cost model and a set of resource constraints, it can be readily applied to future programmable graphics architectures.

8. Acknowledgments

We would like to thank everyone in the Stanford graphics architecture group for contributing ideas to this work. We had helpful discussions with Bill Mark and Monica Lam during our early brainstorming sessions. Evan Hart and Jeff Royle from ATI provided hardware and driver assistance. This work was done as part of the Stanford real-time programmable shading project, which is sponsored by DARPA (contracts DABT63-95-C-0085 and MDA904-98-C-A933), ATI, NVIDIA, Sony, and Sun.

References

1. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986, p. 602.
2. ATI. RADEON 8500 product web site, 2001. *http://www.ati.com/products/pc/radeon8500128/index.html*.
3. CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. W. Register allocation via coloring. In *Computer Languages* (1981), vol. 6, pp. 47–57.
4. CHAMBERLAIN, B. L. Graph partitioning algorithms for distributing workloads of parallel computations. Tech. Rep. TR-98-10-03, 1998. *http://www.cs.washington.edu/homes/brad/cv/pubs/degree/generals.pdf*.
5. COOK, R. L. Shade trees. In *Proceedings of ACM SIGGRAPH* (1984), pp. 223–231.
6. FRASER, C., AND HANSON, D. R. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995, pp. 373–406.
7. GRITZ, L. Renderman repository web site, 1998. *http://www.renderman.org/RMR/Shaders/BMRTShaders/wood2.sl*.
8. HANRAHAN, P., AND LAWSON, J. A language for shading and lighting calculations. In *Proceedings of ACM SIGGRAPH* (1990), pp. 289–298.
9. LINDHOLM, E., KLIGARD, M. J., AND MORETON, H. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH* (2001), pp. 149–158.
10. MARK, W. R., AND PROUDFOOT, K. The F-buffer: a rasterization-order FIFO buffer for multi-pass rendering. In *Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2001), pp. 57–64.
11. NVIDIA. GeForce3 Ti family product overview, 2001. *http://www.nvidia.com/docs/lo/1050/SUPP/gf3ti_overview.pdf*.
12. OLANO, M., AND LASTRA, A. A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of ACM SIGGRAPH* (1998), pp. 159–168.
13. PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH* (2000), pp. 425–432.
14. PERLIN, K. An image synthesizer. In *Proceedings of ACM SIGGRAPH* (1985), pp. 287–296.
15. PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH* (2001), pp. 159–170.

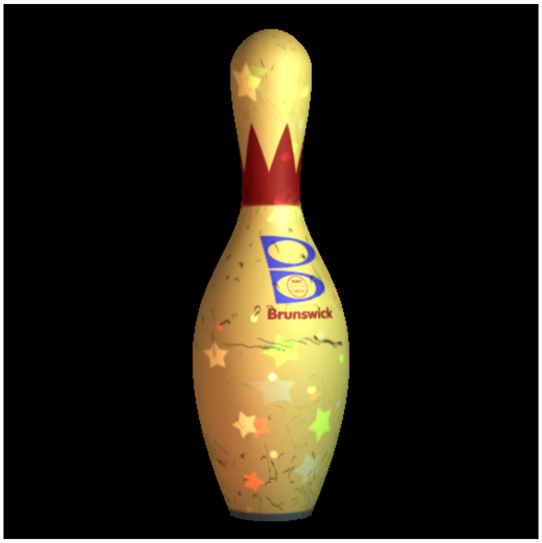

Figure 7: *RenderMan bowling pin with 1 point light source and 4 animated textured lights. Since the shader is too large to map to a single pass, our RDS algorithm splits it into multiple passes. The pin renders at 30 frames/sec on a 1.4 GHz Pentium 4 system with an ATI Radeon 8500 graphics card.*