PYSTREAM: PYTHON SHADERS RUNNING ON THE GPU

BY

NICHOLAS C. BRAY

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Abstract

Graphics processing units (GPUs) have tremendous computing power, but are hard to program. Most GPU programming languages are extremely low level; no one has run a general-purpose high-level language on a GPU. This dissertation shows how to run Python shaders on a GPU for real-time rendering. Shaders written in Python run 30,000 - 300,000 times faster than on a CPU. This is partly because GPUs are designed to run shaders and partly because of the design of PyStream, a Python compiler for the GPU. PyStream advances the state-of-the-art in pointer analysis for Python, eliminates abstraction overhead for Python, maps a language with references onto an architecture that does not support references, and uses a single code base to program both the CPU and GPU. PyStream points the way for running high-level languages on specialized architectures.

Contents

Chapter 1

Introduction

This dissertation explores running a modern high-level dynamic language on a restricted, specialized high-performance architecture. More specifically, we are focused on the means to write Python shaders for real-time rendering systems and to run these shaders on a graphics processing unit (GPU). This was accomplished by using a subset of the Python language that was compiled onto the GPU, with the implication that a rendering system could be entirely written in Python. The platform created for this research is referred to as *PyStream*, which is both a compiler and a language.

While the investigated topic is real-time rendering shaders, the means used in this investigation are themselves independent topics of research. The PyStream platform shows how modern high-level dynamic languages such as Python can be run directly on the GPU, which is a restricted architecture. This research demonstrates potential means by which an existing high-level dynamic language can be run on architectures for which the language was not originally designed. The PyStream compiler also demonstrates how pointer analysis can be performed for Python. This research demonstrates a potential means for analyzing the entire Python language. The research further demonstrates how programs with references can be translated onto architectures that do not support references, or more specifically, means by which Python's semantics for references can be emulated on the GPU.

The PyStream compiler translates a subset of Python into the OpenGL Shading Language (GLSL), effectively allowing Python to run on GPU architecture, something that was thought to be an intractable problem. Running Python shaders in GLSL is actually harder than just running Python on a GPU because GPU architecture is more robust than exposed by GLSL. The PyStream platform contains a number of features that allow Python to run on the GPU. One feature of the PyStream platform is that it integrates Python's interpreter functions into the program allowing the interpreter to be compiled and optimized in tandem with the program. The resulting program can be run without an interpreter. PyStream encodes the semantics of Python in libraries and then analyzes the libraries instead of having Python's semantics hard-coded in the analysis, making PyStream simpler. Use of libraries in this manner means that PyStream can both compile Python's complicated semantics and offer hope of coping with the full Python language.

Present programming techniques generally use low-level languages to program restricted high-performance architectures. Low-level languages allow greater control over how a program is implemented, but low-level languages also *require* greater control over how a program is implemented. As programs become more complex, this required control, inherent in low-level languages, becomes an increasing programming burden. For large-scale systems this required control demands that every programming decision be made correctly or program performance will be diminished. A platform like PyStream shifts many implementation issues onto the compiler. PyStream does this by using a high-level language to program a restricted high-performance architecture and by performing domain-specific optimizations.

Using current techniques, programming real-time rendering systems is difficult. A significant part of this difficulty stems from the use of GPUs to accelerate the rendering. GPUs are data-parallel coprocessors and can provide outstanding performance on tasks that fit their architecture. However, GPUs are burdened by common factors associated with using coprocessors. In most applications, using coprocessors requires two separate code bases, one for the processor and one for the coprocessor. Significant glue code — code that moves and reformats data from a source to a destination — is often required to transfer data between a processor and a coprocessor. In addition, GPUs must be explicitly managed by their host application, commonly through APIs such as DirectX, OpenGL, OpenCL, or CUDA. This explicit management increases programming time and code complexity. The increased system complexity resulting from these factors is the cost paid for achieving improved performance.

While the usual reason to adopt parallel architectures is to achieve greater performance, high-level dynamic languages such as Python are not designed for efficient execution. Some high-performance parallel architectures, such as GPUs, do not have or do not make available many of the features commonly used to implement dynamic languages. Algorithms developed for this research allow Python, a dynamic language, to be run on the GPU architecture without needing these features. The PyStream platform makes Python run fast and efficiently on a GPU.

An example real-time rendering system, written entirely in Python, runs at least 146,000 times faster when its shaders are compiled onto the GPU. Using a GPU instead of a CPU can improve performance for numeric applications by one to two orders of magnitude. Real-time rendering is a numeric application that GPUs were designed to accelerate. Further, eliminating Python's overhead can improve performance by additional orders of magnitude.

There are two implications from this research into running Python shaders on the GPU. First, the research identifies a new generation of tools to reduce the complexity of rendering systems and make writing them faster and simpler. Running Python on a GPU allows rendering systems to be written in a single language with a single code base. Having a single code base gives multiple engineering advantages, including eliminating the need to write glue code between a processor and a coprocessor. Instead, the glue code can be automatically generated, enabling the use of future improvements in an API without requiring changes to the code base. The second implication is that this research demonstrates the feasibility of running a high-level language on a restricted, specialized architecture and opens up new areas of research in language translation and high-performance computing.

The PyStream platform translates a subset of Python into GLSL. Translating Python into GLSL requires the development of new compiler algorithms and the extension of existing compiler algorithms. The compiler algorithms discussed in this dissertation can be divided into two categories: those compiling Python and those translating Python's semantics into GLSL.

Algorithms that compile Python are necessary to run Python on GPU architecture. Python was designed to be interpreted, not compiled, and GPUs were not designed to run interpreters. At present, there has been little work done on compiling Python and little discussion in the literature of compilation techniques for Python. This dissertation identifies impediments encountered when compiling Python and provides techniques for overcoming them.

The second category of compiler algorithms addresses translating code that, like Python, has references to objects into code that, like GLSL, holds objects by value. This difference creates a significant semantic gap that interferes with code translation. Python may have lexical similarities to GLSL, but the underlying meaning of these languages is fundamentally different. This dissertation details transformations for bridging this gap.

1.1 Goal Statement and Research Questions

Our goal is to *reduce the burden of writing a GPU-accelerated real-time rendering system.* Specifically, we want to run an existing general-purpose high-level language on the GPU. We hypothesize that a useful subset of Python can be run directly on the GPU, allowing an entire rendering system to be written in Python. Our thesis is that *it is possible to program a GPU in a high-level language like Python.* There are a number of questions that must be answered.

How can Python be analyzed in an effective manner? Previous attempts to analyze Python have been hampered by the complexity of the Python language. Translating a general-purpose high-level dynamic language onto a specialized architecture requires a clear, precise understanding of a program's semantics. Effective analysis of Python shader programs is necessary to enable their transformation.

Can Python's abstraction overhead be eliminated with static compilation? It is not enough to simply run Python on a GPU; it must run fast. Running Python on a GPU would be a technical feat in of itself, but performance is the motivating factor. Python was not designed for efficient execution, and its abstraction overhead must be eliminated if it is to be directly used in a high-performance application. Eliminating abstraction overhead also simplifies a program, which should make it easier to translate the program to a specialized architecture.

Can Python, a modern high-level dynamic language designed to run on a CPU, be translated onto a specialized high-performance architecture on which it was not designed to run? Due to technical limitations of current GPU APIs, an even harder question must also be answered: Can Python run on an architecture that does not support hold-by-reference semantics? Translating Python shaders into GLSL shaders appears to be the best way to run Python shaders on the GPU, but GLSL does not support references or memory allocation. Techniques for bridging this semantic gap have not been previously explored.

1.2 Contributions

This dissertation is an exploration of how a modern, high-level dynamic language can be run on a specialized high-performance architecture, and how it can be run fast. This exploration breaks new ground, and improves the state-of-the-art in a number of areas:

- Python shaders running on the GPU. This dissertation shows how Python shaders can be run on the GPU, allowing an entire rendering system to be written in Python, Python's abstractions to be used on the GPU, and code to be seamlessly shared between CPU and GPU. Existing shader languages require that a rendering system be split into two code bases, do not allow sharing between the code bases, and offer few abstractions for shader programming.

- Low-level analysis for Python. This dissertation shows how the problem of analyzing Python, a high-level language, can be simplified by analyzing it at a slightly lower level. Prior attempts to analyze Python have been frustrated by the complexity of Python's semantics.

- Pointer analysis for Python. This dissertation shows how pointer analysis can be performed for Python, an area that has not been previously explored. Issues that can impede pointer analysis of Python programs are identified, and solutions are formulated. Low-level analysis also allows pointer analysis for Python to be linked to the larger body of work on Pointer analysis.

- Translating Python into GLSL. This dissertation shows how a useful subset of Python can be translated into GLSL, and by implication onto GPU architecture. This translation bridges a significant semantic gap, and allows Python to be run on an architecture it was not designed to run on.

- Domain-specific optimization. This dissertation shows that a domain-aware compiler can improve the performance of real-time rendering systems.

1.3 Organization

The chapters of this dissertation are structured in sequence to provide the reader with information about a number of topics to help understand the background and importance of this research into the PyStream compiler. Chapters 7, 10, and 11 discuss the contributions of the PyStream compiler related to pointer analysis for Python, translating Python into GLSL, and the results of writing and running an entire rendering system in Python. Chapter 5 discusses the overall structure of the compiler. Chapters 8 and 9 discuss the features of the PyStream compiler that eliminate Python's abstraction overhead and eliminate memory operations from Python shaders.

Chapter 2

Engineering Real-Time Rendering Systems

> “Interestingly enough, we’ve come to the point where the body of shader code alone has grown larger than the entire codebase for games from the early stages of the games industry.” - StarCraft II: Effects and Techniques

Real-time rendering systems have a number of software engineering problems that are not adequately addressed by current techniques. The PyStream platform is designed to address these software engineering problems. This chapter discusses the general nature of these software engineering problems and how the PyStream platform addresses them.

Real-time rendering is a computation process where a computer synthesizes images fast enough that a user can interact with them. This pushes a processor to its limits because of the large amount of computation required. As more computational power has become available over time, image fidelity in real-time rendering has been improved by increasing the amount of computation per image. The goal of real-time rendering is a moving target. Real-time rendering systems require all the performance that can be achieved from a processor, making efficiency an important aspect of programming these systems. Real-time rendering systems thus have favored the GPU because of the computational power that it provides, allowing images to be synthesized faster and with higher fidelity than on a CPU. As image fidelity has improved, the complexity of real-time rendering systems has correspondingly increased. Managing increasingly complex rendering systems presents a number of software engineering challenges.

This chapter provides background information on real-time rendering and discusses why it was the focus of this research. This chapter also details how GPUs are used as coprocessors for real-time rendering and discusses why reducing the barriers between the CPU and the GPU can reduce the overall complexity of a real-time rendering system.

2.1 Historical Emergence of Real-Time Rendering

Real-time rendering has changed at an astonishing rate. The first consumer-level 3d accelerator, an early form of the GPU, was released roughly 10 years ago. Since then GPUs have become faster and more flexible every year. The earliest GPUs had fixed, hardware-defined functionality. Since then, GPU programmability has increased along with GPU performance. Intel has a GPU under development [1] that is architecturally similar to a multicore x86 CPU. The programming interfaces for GPUs have gone through a corresponding evolution, starting with fixed functionality, moving to programmable graphics pipelines, and now to general-purpose GPU computation [2, 3, 4]. There has recently been a push to use general-purpose GPU computation for real-time rendering [5].

2.2 Rendering Heuristics

A real-time rendering system is a set of heuristic algorithms that are developed through trial and error to provide a convincing approximation of how light interacts with objects. Even in photo-realistic rendering systems, heuristics are used to approximate the generally intractable light transport equation. One commonly used heuristic, for example, is the Lambert model for diffuse reflectance. Diffuse reflectance describes how light is scattered when it reflects off a surface. The Lambert model defines diffuse reflectance to be proportional to `max(dot(n, l), 0)`, where `n` is the surface normal (the direction perpendicular to the surface) and `l` is the direction to the light source. Although this model is not physically or perceptually realistic, it is very easy for real-time rendering systems to compute.

Another commonly used heuristic is to ignore the interreflection of light between surfaces and compute only the effect of light directly striking a given surface. This significantly improves the tractability of the light transport equation at the cost of eliminating many subtle visual cues.

The choice of heuristics has aesthetic implications because changes to the heuristics can affect the appearance of the image. For example, rendering everything in the image using a Phong reflectance model can cause everything to look like plastic due to how it models shiny surfaces. Another example is that the lack of interreflection can cause a rendered image to look harsh, or stark. Choosing a specific set of heuristic algorithms can give a real-time rendering system a specific set of qualities to varying degrees, including performance, style, image fidelity, and scope, among others.

Different heuristics can be chosen to trade one quality for another. The usual trade-off is image fidelity for performance. For example, lighting calculations can be performed at each vertex of a triangle and interpolated across the triangle rather than performing the calculations at each pixel of the rendered triangle. Per-vertex lighting will be faster than per-pixel lighting, but it will also produce lower quality results. Similarly, reflectance functions that are easy to compute can be substituted for reflectance functions that are better looking. Controlling these trade-offs is critical to creating a rendering system that meets the requirements of a given application. The PyStream platform makes it easier to control these trade-offs because it simplifies the creation and modification of real-time rendering systems.

2.3 Real-Time Rendering System Life Cycle

Modifiability is an important quality of rendering systems. Initially, rendering systems rapidly evolve in a prototyping phase. Later in their life cycle, they need to be modified in response to changing requirements or to fix bugs. For graphically demanding applications, such as video games, it is typical for an existing rendering system to be used and extensively modified to meet the specific needs of the game.

Despite the importance of modifiability, real-time rendering systems are often difficult to modify. Generally, the need for efficiency results in higher-level rendering algorithms being coupled with lower-level implementation decisions. More specifically, using a GPU as a coprocessor requires that rendering algorithms be written to accommodate the programming interface of the GPU. GPU programming interfaces expose low-level functionality for maximum flexibility and performance. Using these programming interfaces is analogous to writing a program in assembly language for an extremely complex instruction set computer (CISC) architecture. As with assembly language, there are often multiple ways to achieve the same functionality with a GPU API, each way having different performance characteristics. For example, it may be possible to render an image in several small steps, or in a single large step. Rendering the image in several smaller steps will consume more memory bandwidth because of the need to read and write intermediate values. At the same time, several small steps may run faster if computation can be reused and the individual steps have simpler control flow. Although both of these approaches can produce identical results, they will have distinctly different implementations and different performance characteristics.

Many rendering systems query the GPU on what features it supports, and then select different code paths depending on the features provided by the GPU. Many of these paths are functionally equivalent, but have different performance characteristics. This coupling of low-level implementation issues with high-level rendering algorithms obscures the rendering algorithm. Modifying the rendering algorithm therefore requires simultaneously modifying the implementation. Ideally, a real-time rendering system would be a high-level expression of the heuristic rendering algorithms, and implementation issues would be delegated to the compiler. The PyStream platform is a step toward this ideal.

Real-time rendering systems can have short lifetimes when compared to other types of software. Higher performance GPUs and CPUs can make new rendering techniques viable. New GPU features can create new ways to implement old techniques. These trends, combined with the inevitable coupling of implementation and algorithm in real-time rendering systems, can result in significant architectural shifts between generations of rendering systems. For example, introduction of a GPU that could be programed with shaders fundamentally changed real-time rendering system software architecture. Supporting legacy code is therefore not a critical issue for most real-time rendering systems. Moving implementation issues into a compiler, as the PyStream platform does, can potentially increase the longevity of a rendering system by allowing the system to evolve along with changes in the hardware without having to be rewritten.

2.4 GPU Architecture

Graphics processing units (GPUs) are specialized coprocessors designed to accelerate real-time rendering. GPUs are massively parallel and have high memory bandwidth. To realize all performance offered by GPUs, it is necessary to program them in ways that reduce the impact of memory latency and maximize the use of memory bandwidth. Memory will otherwise be a bottleneck to keeping a GPU's functional units fed. A widely used model for programming GPUs is stream programming (Section 2.8.3). Stream programming uses "kernel" functions that operate on streams of similar elements. Kernel functions are restricted so that they have no side effects. Kernels can be executed in parallel on multiple elements in the stream. Kernel functions also do not define how the stream is iterated over: rather, they only define how an individual element is processed. As a result, a compiler has latitude to optimize memory usage patterns. *Shaders* are a restricted form of kernel function with additional rendering-specific functionality.

Modern GPUs are programmable, but they impose a fairly rigid structure on the overall rendering pipeline. GPUs structure the rendering pipeline as a series of three user-specified shader kernels with fixed functionality before and after each shader. The first shader in the pipeline is called the *vertex shader*. It processes the individual vertices of a rendered object. Fixed functionality then assembles the vertices into small groups that represent geometric shapes, such as lines and triangles. These shapes are then processed by the second shader, called the *geometry shader*. The geometry shader can modify the vertices in a shape, create new shapes, or even discard the shape it is given. Fixed functionality then rasterizes the geometric shapes into pixels. More precisely, the rasterizer creates multiple *fragments* that are later blended together to create a single pixel. The fragments are processed by a third shader, called the *fragment shader*. The fragment shader computes the color of each fragment. Fixed functionality then blends these fragments into a two-dimensional array called a *frame buffer*. The rasterizer determines where each fragment will be stored in the frame buffer. Together, shaders used in the rendering pipeline are called a *shader program*. To fully utilize a GPU as a coprocessor for real-time rendering, rendering algorithms must fit within the model provided by the GPU.

2.5 GPU Application Programming Interfaces (APIs)

GPUs are coprocessors, and an application programming interface (API) is used to control their execution. Real-time rendering systems often contain many calls to the API to control the GPU. GPU APIs are rapidly evolving.

One way GPU APIs evolve is by adding new features. For example, the types of shaders supported by the rendering pipeline have increased as APIs have evolved. Currently vertex shaders, fragment shaders and geometry shaders are supported, and three kinds of tessellation shader will soon be supported.

A second type of evolution adds alternative API calls that provide better performance than existing API calls. Typically, this type of evolution consolidates into a single call operations that previously required multiple calls. For example, OpenGL originally required that each component of each vertex be specified with an API call. Subsequent versions of the OpenGL API allow arrays of attributes to be specified with a single call. The current version of the OpenGL API now allows multiple arrays to be specified with a single call. Reducing the number of API calls can provide significant performance benefits on systems with protected memory, as API calls require switching protection modes. It is not uncommon for rendering algorithms to be bound by API call overhead.

A third type of evolution reduces the need to synchronize the CPU and GPU. GPUs are deeply pipelined, and operations that transfer data back to the CPU can stall until the data is ready. If the CPU is stalls, it stops feeding data to the GPU and as a consequence the GPU pipeline can be flushed. Synchronization has a performance overhead for both the CPU and GPU. API improvements have allowed asynchronous data transfer between the GPU and CPU. For example, an addition to the OpenGL API allows pixel data to be read back to the CPU asynchronously. API improvements have also allowed some dependent rendering operations to be executed entirely on the GPU, eliminating the need for data transfer. For example, an addition to the OpenGL API allows API calls to be conditionally executed if a previous call resulted in pixels being drawn. This allows API calls to be issued before the result of the previous call is known.

The latter two types of API evolution improve efficient management of execution on a GPU. These types of evolution in the API do not change the rendering pipeline even though they may require radically different API calls. The PyStream compiler can target new versions of an API without requiring change in the rendering system.

2.6 Software-Defined Rendering Pipelines

A software-defined rendering pipeline can be compared in function to a GPU hardware-defined rendering pipeline. A software-defined rendering pipeline allows functionality to be specified in software which is currently defined in hardware. Rendering algorithms are beginning to break away from the hardware-defined rendering pipeline. GPUs that provide a software-defined rendering pipeline seem to be the likely future. The flexibility of a software-defined rendering pipeline would allow a much richer set of algorithms to be used for real-time rendering, but it also has the potential to exacerbate software engineering challenges. A software-defined rendering pipeline would give greater control when writing programs. The research design was mindful of the likelihood that software-defined rendering pipelines may become the standard within the decade. At the same time, the research design was mindful that the PyStream platform must run on today's hardware.

2.7 Shader Languages

A single rendering system cannot support every possible rendering algorithm. Rendering systems are typically designed to be extended, allowing a program to specify shaders that are executed at well-defined points in the rendering pipeline. GPUs commonly in use support three categories of shaders: vertex shaders, geometry shaders, and fragment shaders. Each of these shaders is designed to process a different kind of data. Future real-time rendering systems will support more categories of shaders.

Most shaders are performance sensitive. This is because a single shader can be executed millions of times while rendering an image. Specialized languages are often used for programming shaders to address performance issues. These languages have restricted sets of semantics to ensure good performance. Two of the most widely used real-time shading languages are GLSL and DirectX's High-Level Shading Language (HLSL). These languages are similar. Their syntax is based on the C language. Semantically neither language supports references to objects or memory allocation.

Previous work has demonstrated the use of interpreted Python shaders for non-real-time, CPU-based ray tracing [6, 7]. A simple Python shader interpreted on a CPU was observed to run at approximately one thirtieth the speed of a native equivalent written in C and run on the CPU. Time spent ray tracing dominated the time spent in all of the shaders, so the overall system was half as fast when Python shaders were used. In contrast, the PyStream platform compiles Python shaders onto the GPU, resulting in less overhead and better performance than interpreting them on the CPU.

2.7.1 Lack of Abstraction in Shader Languages

Historically, GPU shaders were written in assembly language while currently, "high-level" shading languages like GLSL, HLSL, and Cg are commonly used. These languages are easier to use than the prior GPU shader assembly languages, but they still lack a level of abstraction found in most general-purpose languages. Real-time rendering systems are commonly structured to compensate for this lack of abstraction in shading languages. For example, currently available shader languages do not easily support polymorphism. To work around this absent feature, it is common practice to use a shader preprocessor and create a suite of variant shaders by recompiling a single shader multiple times with different preprocessor definitions [8]. A quasi-static control flow statement can be used to achieve the same effect, but it requires more sophistication from the compiler. The Cg shading language allows the specification of polymorphic interfaces, but using these interfaces is cumbersome because each shader program must be manually specialized for every combination of concrete objects that will occur at run time. Similar interfaces will be included in DirectX 11. The PyStream platform allows shader programs to be written in an object-oriented language that cleanly supports expressions of polymorphism.

2.7.2 Barriers Between the CPU and GPU

In real-time rendering systems that use a GPU as a coprocessors, refactoring code across the renderer/shader interface can be complicated and time-consuming. Refactoring reorganizes code but maintains its behavior. Unless there is an extremely strong reason to refactor, the division of functionality between the renderer and the shaders tends to stay fixed in a rendering system.

Unlike the PyStream platform, most rendering systems must separate application code from shader code. Arbitrary functions in the shader code cannot be called by the application code, nor can the shader code call functions in the application code. While this may appear to be a consequence of shaders and the application being written in different languages, inter-language calling conventions are available in other situations, but not for shaders.

Shader code and application code are not integrated because of a design assumption that shader code will only be executed on the GPU in the rendering pipeline. With the exception of the top-level function of each shader, there is little reason the shader code could not be reused in other contexts. The way to currently execute arbitrary shader functions on the CPU is to rewrite them in the application's language. This results in at least two definitions of the same function, which in turn complicates maintenance of a rendering system. The PyStream platform allows all shader code to be directly called by the application code.

In addition to not sharing code, existing host applications and GPUs do not share data structures. To move data between the processor and the coprocessor, glue code must be written which pulls primitive data types out of the application's data structures and writes them into the shader's data structures. This functionality is similar to serialization. Three things in a rendering system must be manually synchronized: the application's data structures, the shader's data structures, and the glue code. Not sharing data structures between the CPU and the GPU complicates modifying a rendering system, particularly at the renderer/shader interface. The PyStream platform shares data structures between the application and the shaders, automatically generating glue code when needed. This feature of the PyStream platform increases the modifiability of a rendering system.

Many of these shader language issues are actually a consequence of a deliberate design decision to keep the shader language independent from the host application language, making it simpler to implement a shading language. It also simplifies binding a GPU API to a variety of host languages. Keeping a shader language independent from a host application language makes it difficult to program a GPU. PyStream forgoes the benefits of keeping a shading language and a host application language separate from each other in order to facilitate simple programming of the GPU.

2.7.3 Prior Work to Simplify Shader Programming

The PyStream platform is not first to recognize that shader programming must be simplified. Various means have been used to simplify the writing of shaders, including shader authoring tools, collaborative shader systems, and shader metaprogramming, among others.

Shader authoring tools are programs that can be used to author individual shader programs and create *shader effects* composed of several sequentially executed shader programs [9, 10]. Authoring tools are useful for prototyping rendering algorithms, but rendering algorithms must ultimately be written for their usage context.

Collaborative shader systems [11, 12, 13] are another approach used to simplify the creation of shaders. This is done by allowing graphical editing of a portion of the shader and then injecting the resulting code into a template. For example, the collaborative shader system in Unreal Engine 3 allows the editing of surface properties and light properties, but does not allow the modification of how surfaces and lights interact. This type of restriction is typically imposed by the underlying rendering algorithms, and the collaborative shader system is created to obey these restrictions. In effect, a collaborative shader system is a render-aware shader authoring tool with a visual data-flow programming language. Data flow programming is quite effective for describing many shading algorithms [14], but has difficulty when complex control flow is required. For example, the parallax occlusion mapping algorithm [15] is difficult to express in a strict data flow language due to its use of loops. Some collaborative shader systems provide a hybrid approach where the user can create custom nodes in the dataflow graph that contain procedural code. A collaborative shader system could be built on the PyStream platform.

Sh [16, 17] is shader metaprogramming library for the C++ language. Sh dynamically constructs shaders when the library is used in a C++ program. The shaders that are constructed are semantically equivalent to GLSL or HLSL. Sh does not offer new semantics for shader programs but it does offer two software engineering benefits. Sh can use the abstraction mechanisms of the C++ language when generating shaders, making it easier to generate specialized, polymorphic variants of a shader. These variants must be manually generated. Sh can also generate glue code in tandem with generating the shader. Sh does not entirely eliminate the need for glue code, as data must still be passed from the C++ language to Sh.

2.8 Computation on the GPU

This section is about general-purpose computing on the GPU as it relates to real-time rendering. Real-time rendering systems are beginning to use general-purpose computation techniques to implement rendering algorithms.

The joining of Python with a GPU has been recognized as a potentially fruitful combination. Python is designed for rapid application development and GPUs are designed for high-performance data-parallel computation. Combining the best aspects of both potentially allows rapid development of high-performance data-parallel applications. Previous efforts to join Python and the GPU have focused on using Python as a tool to generate other code that runs on the GPU. The PyStream platform focuses on actually running Python on the GPU.

2.8.1 PyGPU

PyGPU [18] is a GPU-accelerated domain-specific language for image processing that superficially resembles PyStream since it uses Python syntax and runs on the GPU. However, PyGPU is closer in nature to Sh because Python syntax is only used to generate GPU programs in another language. PyGPU does not support Python semantics on the GPU.

2.8.2 Data Parallel Array Libraries

Data-parallel array libraries [19] allow a program running on a CPU to use a GPU as a data-parallel co-processor for numeric applications. Data-parallel array libraries consist of array types that can contain uniform numerical data and functions that can operate on the arrays in parallel. This type of library allows a given language to use a GPU as a data-parallel coprocessor, but only for a specific set of numerical operations. (At the time this dissertation was written, we are aware of at least one unreported GPU-accelerated data-parallel array library in development for Python.)

Data-parallel expression libraries significantly improve the programmability of GPUs, but these libraries only allow for a restricted set of semantics and data types. The host language can perform data-parallel computations on the GPU, but the host language cannot actually run on the GPU.

2.8.3 Stream Programming Languages

Stream programming is another method that can perform computation on the GPU. Shading languages are a form of Stream language. Stream languages have issues similar to those previously discussed in the context of shading languages (Section 2.7). Many stream programming languages are syntactically similar to the C language [2] or the Java language [20]. Stream languages restrict kernel side-effects like shader languages. Most dynamism at the language level is eliminated in Stream languages to ensure performance. Stream languages are designed to operate on large sets of similar data, and they are not well suited for computational tasks such as loading data from disk or routing data between kernels, tasks for which they require a host application. This requirement of stream languages creates two code bases and requires manually developed glue code.

The PyStream platform can potentially support high-performance stream programming in Python. The PyStream platform has the potential to benefit stream programming as it does shader programming, including providing a single code base, eliminating glue code, and minimizing explicit management. Combining abstractions from the Python language with stream programming using the PyStream platform offers the potential to improve programmability while maintaining performance.

2.8.4 Compute Unified Device Architecture (CUDA)

CUDA [3] is a low-level language for data-parallel programming on the GPU. CUDA is lexically similar to the C language. Despite similarities with stream languages, CUDA is not a stream language. CUDA does not abstract memory access. In fact, CUDA explicitly exposes the memory hierarchy of the GPU to allow a user to control where data is stored in the GPU's memory. This design allows high performance parallel programs to be written at the cost of portability and understandability, and with reduced latitude available to an optimizing compiler. CUDA allows references to data, global writes, and arbitrary gather/scatter operations. CUDA's features indicate that a GPU has a more liberal execution model than is usually exposed through GPU APIs designed for real-time rendering. Real-time rendering APIs are currently the only way to access rendering-specific hardware on the GPU, however, so CUDA could not be used for this research.

There is a library for Python that allows CUDA kernels to be called from a Python program. Confusingly, this library is also named PyStream. This library is a wrapper for calling foreign functions, and is nothing like the platform described in this dissertation.

2.9 Example Rendering System

Chapter 11 details the example rendering system used in this dissertation to evaluate Python shaders running on the GPU. The results presented in this dissertation have been obtained from compiling the Python shaders of the real-time rendering system described in Chapter 11.

Chapter 3

Python

Python is a widely used, interpreted, modern high-level dynamic language, designed for rapid application development. The PyStream platform created for this research was developed to integrate with Python. The PyStream platform includes a language, called the PyStream language, which is a subset of the Python language. The PyStream language can be compiled by the PyStream compiler onto the GPU. The PyStream language is discussed in Chapter 4. This chapter discusses the characteristics of Python that make it both a powerful tool and difficult to compile.

3.1 An Introduction to Python

Python is an object-oriented language and shares common concepts with other object-oriented languages but also has many differences, both subtle and obvious. Figure 3.1 shows a Python function that interpolates between two inputs based on a third input. A lexical feature of Python is that it defines blocks of code using indentation, in contrast to most other languages that denote blocks of code using special characters ({ and } in the case of the C language). In Python, the indentation level of each statement determines what block of code the statement

```
def linearStep(x, y, amt):
    if amt < 0.0:
        return x
    elif amt > 1.0:
        return y
    else:
        a = x*(1.0-amt)
        b = y*amt
        return a+b
```

Figure 3.1: Python Function for Blending Values

belongs to.

3.1.1 Dynamic Types

A feature of Python illustrated by Figure 3.1 is that there are no annotations indicating the type of a variable. Python is a dynamically typed language, which means that type annotations are not used, and the correctness of the program is determined at run time. A function may be called with any type of argument, as long as the arguments support the operations performed inside the function. In the Figure 3.1, for example, the function `linearStep` could be called with floating-point values for all of its arguments and it would return a floating point value. Alternatively, three-dimensional vectors of floating point numbers could be passed to `x` and to `y`, and the function would then return a three-dimensional vector of floating point numbers. If strings of text were passed to `x` and to `y`, then a run time error would occur during the execution of the function because a string does not support multiplication with a floating-point number. Although Python does not have type annotations, it still has the notion of type errors. Type errors occur at run time when an invalid operation is executed.

3.1.2 Everything is an Object

Python is an object-oriented (OO) language. In addition, everything in a Python program is an object. For example, when adding two numbers (`1+2`), each of the numbers is an object, as is the result of the addition. How addition is performed in general is defined by Python's semantics, and how addition is specifically performed is defined by specially named methods attached to the objects (in this case the methods are named `__add__` and `__radd__`). Numbers, Boolean values, and strings are all objects, and they are also immutable value objects. Once a value object is allocated, its value does not change. Consequently, arithmetic operations on numbers produce new objects. If an arithmetic operation on two integers overflows, a long integer is produced. Long integers contain more bits than are natively supported on most architectures and are effectively immune to overflow at the cost of additional computation. Floating point numbers are always 64 bits long in Python.

3.1.3 Function Objects

In Python, functions are objects. Declaring a function creates a new function object and attaches byte code to it. A function declaration also stores the function object into a global dictionary, based on the declared name of the function. In Figure 3.1, the declared function is stored in the global variable `linearStep`. This function can be retrieved by reading the global variable. Global variables are mutable, however, so a function declaration can be overwritten by another function declaration, or by any sort of expression that sets a global variable. Unlike languages such as C, declarations in Python do not permanently bind the declaration to a given name.

Function calls are resolved dynamically. A lexical function call, such as `f()`, is resolved in two steps. First, the global variable `f` is read. Second, the result of the lookup is called. The fact that an object is called does not mean that it must be a function. If an object has a method named `__call__`, the object can be called as if it were a function, and the call will be delegated to the method. What appears to be a lexical function call may have arbitrary semantics. The precise semantics of a call cannot be resolved without additional information about the object being called. Which object is called depends on the contents of a mutable global dictionary. In general, no assumptions can be made about any call site when analyzing a Python program. This means that the call graph of a Python program must be dynamically discovered.

Function objects contain dictionary objects that hold the global variables that a function can access. Function objects also contain an object that holds the default arguments of a function. Unlike other languages, default arguments in Python are not inherent properties of the underlying executable code, nor is the global name space. These features of a function are part of the object structure. When used in this dissertation, the term *function* refers to the underlying executable code, and not to the function object.

3.1.4 Class Objects

Figure 3.2 shows a class declaration in Python. Class declarations create class objects and store them into a global dictionary, similar to function objects. Calling a class object creates a new instance of the class. There are no special operators for allocating objects. In Python, each object has a pointer to its class object. This

```
class Example(object):
    def say(self, message):
        print message

# Instantiate the class
e = Example()

# Call a method
e.say("Hello, world")
```

Figure 3.2: Example Python Class

is similar to a vtable in the C++ language, except for the fact that a Python type pointer points to a full-fledged object, and not an opaque data structure. There is no concept of “casting” the type of a Python object; an object’s type is solely determined by its type pointer. Type pointers are mutable, which allows the type of an object to change, but this is rarely done in practice. For all intents and purposes, the type of a Python object is fixed when it is allocated.

A method is a function that is declared inside of a class. When a method is invoked on an object, the method name is looked up on the object’s class object. If the result is a function, it is treated as if it were a method. In most other OO languages (C++, Java, Smalltalk, and others) the parameter containing the object on which the method is invoked is an implicit parameter of the method, typically named `this` or `self`. In Python, since methods are functions, this parameter is explicitly declared as the first parameter of the function. By convention, most Python programs name this parameter `self`.

Methods invocations are resolved in two steps. First, the method name is resolved as if it were one of an object’s attributes. This process returns a *bound method object* that contains a reference to the original object and to the function implementing the method. Calling a bound method object in turn results in calling the function, but with the original object inserted as the first parameter.

3.1.5 Built-in Container Types

Python has three main types of built-in container objects: `tuple`, `list`, and `dict`.

Tuple. Tuple objects are immutable arrays of objects. Tuples are immutable both in their contents and in their length. Tuples are commonly used in Python programs to temporarily group objects together and to return multiple objects from

```
def interpreter_add(a, b):
    result = a.__add__(b)

    if result is not NotImplemented:
        # The call succeeded.
        return result

    # adding a and b did not work, fallback
    # to the reverse addition method
    result = b.__radd__(a)

    if result is not NotImplemented:
        return result

    # cannot add the objects, raise an exception
    raise NotImplementedError, ...
```

Figure 3.3: Simplified Semantics for Adding Two Objects

a function.

List. List objects are arrays which are mutable, both in terms of their contents and length. Lists are commonly used wherever arrays of data are needed.

Dict. Dictionary objects are associative dictionaries that map keys to values. Dictionaries are used extensively in the Python run time. For example, global variables are held in dictionaries that associate the global variable name with the value of the global variable. Dictionaries are also used to store the attributes specified in a class definition. These dictionaries are referred to as *class dictionaries*.

3.1.6 Python Semantics are Complex

The semantics of Python are not simple. Figure 3.3 shows pseudocode for the logic that a Python interpreter uses to add two objects together. The interpreter attempts to call a method on the first object. If that fails, then the interpreter attempts to call a method on the second object. If that fails, then the interpreter raises an exception. Other operations, such as getting an attribute from an object, have even more complicated semantics. When adding objects, the interpreter attempts to delegate the operation to methods on the objects being operated on. In general, operations in Python delegate functionality to a method, although the logic of determining which method is delegated to is unusually complicated. Ignoring the complexities of how the method is selected, the fact that Python operations are delegated allows an object to override how an operation is implemented by overriding a method. For example, an object could print itself when the addition operation is invoked on it, despite the fact that this is completely inconsistent with what would be expected when adding two objects. It is difficult to predict what an operation in Python will actually do without additional information about the argument types.

Modern high-level languages generally eschew an elegant implementation in favor of adding features to the language. Modern high-level languages are sometimes called “kitchen sink” languages, since almost everything short of the proverbial kitchen sink is thrown into their design. Although adding features to a language may make it simpler to use, each feature has the potential to make the language harder to analyze. Adding features adds special cases to the semantics of the language. Special cases do not impact writing a program as strongly as they impact compiling a program. A programmer can use trial-and-error to reason out the approximate behavior of a program, deemphasizing the special cases. This approximate reasoning process does not work for a compiler because it will likely result in the compiler making unsound transformations, changing the fundamental meaning of the program.

3.1.7 Calling Conventions

Python offers richer calling conventions than are available in the C language. A basic form of Python’s calling conventions can resemble those in the C language. Call sites can specify arguments in a particular order, and the arguments are bound to function parameters in the same order. These are known as positional arguments and positional parameters. In Figure 3.4a, `a0` is a positional argument. In Figure 3.4b, `p0` and `p1` are positional parameters. If the number of positional arguments exceeds the number of positional parameters, then the excess arguments are written into a variable length parameter if it exists. *Variable length parameter* (*vparam*) refers to a function parameter that contains a tuple of excess positional arguments. In Figure 3.4b, `p2` is a vparam. Positional arguments can also be read from a special argument, the *varg*, containing an object that is iterated over. In Figure 3.4a, `a2` is a varg. If a call site does not provide enough arguments for the target function, then a default argument is bound in place of a missing argument, if the function specifies a default for that particular parameter. In Figure 3.4b, `p1`

```
call(a0, foo=a1, *a2, **a3)
```

(a)

```
def func(p0, p1=None, *p2, **p3):
    ...
```

(b)

Figure 3.4: Calling Convention Examples

```
def method_call(self, *v, **k):
    func = self.im_func
    inst = self.im_self
    return func(inst, *v, **k)
```

Figure 3.5: Method Call Pseudocode

will default to `None` if no argument otherwise binds to it.

In Python, arguments are not always bound to parameters by position; they may be bound by name, in which case they are referred to as *keyword arguments*. In Figure 3.4a, `a1` is a keyword argument that binds to a parameter named `foo`. Similar to the vparam, excess keyword arguments can be written into a dictionary held by a *kparam*. In Figure 3.4b, `p3` is a kparam. Similar to the varg, additional keyword arguments can be specified by passing a dictionary to the *karg*. In Figure 3.4a, `a3` is a karg.

These calling conventions are widely used in the Python run time. Using these calling conventions makes it easy to create generic functions. For example, Figure 3.5 shows a pseudocode version of the function that is executed when a method object is called. The use of Python's calling conventions allows a method object to wrap a function with arbitrary parameters. Using these calling conventions makes it harder to analyze function calls, because the data transfer between caller and callee is complicated and tends to obfuscate how an argument gets bound to a parameter.

3.1.8 Metaprogramming

The fact that everything is an object adds uniformity to a Python program. A reference to an instance object is the same as a reference to a class object is the same as a reference to a function object. There are two disadvantages to everything being an object. First, there is no static declaration of a program. Class and function declarations are stored in mutable global variables, and the definition of a Python program can change as it runs. Function objects and class objects are themselves mutable. Second, critical information for resolving the behavior of a Python program is stored in heap memory. Analysis algorithms are notoriously bad at precisely analyzing heap memory, in turn making it difficult to precisely analyze a Python program.

Everything being an object has a significant upside: it makes metaprogramming simple. Metaprogramming is a technique where code modifies other code in a program. For example, metaprogramming can create new classes and functions and can modify existing classes and functions. In languages such as C, a preprocessor is used in place of metaprogramming. Python contains a rich set of metaprogramming features and can use the full power of the Python language to modify a program. In some cases, metaprogramming can greatly simplify the specification of a program. For example, if a program requires many variations of a single function, metaprogramming can be used to generate those variations. Explicitly specifying the variations would otherwise be tedious, and if modifications were later required, then every variation would need to be modified. On the other hand, metaprogramming can make it difficult for a compiler to reason out the structure of a program because the structure of the program is less explicit.

As part of its metaprogramming tool set, Python contains an `eval` statement that interprets arbitrary strings as Python code. This statement allows any Python program to generate and interpret Python code and it is the primary method for dynamically creating functions.

3.1.9 Integrating Python With Other Languages

When Python code alone does not provide the desired performance, then critical sections of the code can be rewritten in another language, such as C. The capacity to integrate with other languages allows Python to be used for rapid application development in domains where performance is important, such as for scientific computing [21]. While Python has been used in many high performance applications, it is largely used to connect the parts of an application and provide configurability in nonperformance-critical sections. Python's performance is generally insufficient for it to be used in performance-critical sections of high performance applications. Mixing Python with other languages improves performance. Most Python interpreters optimize performance-critical sections of the Python run time by writing them in another language. For example, the CPython interpreter uses C to implement the default method for getting an object attribute. This complicates compiling Python, as a compiler must find a way to deal with these foreign functions.

3.2 Compiling Python

Python is designed to be interpreted, and not compiled. Many interpreters have been created for the Python language. The specific interpreter used determines the run time performance of a Python program. Regardless of the interpreter used, an interpreted Python program often runs many times slower than a program implementing the same algorithms but written in C.

Instead of being interpreted as designed, Python can actually be compiled, but only with great difficulty. Features of Python that make the language difficult to compile are the following: Python lacks a static program declaration, Python stores most information critical to resolving the behavior of a program in heap memory, and Python has complex semantics. Python's lack of a static program declaration makes it nearly impossible to say anything about any program without first running it or doing extensive analysis. Heap memory is notoriously difficult for a compiler to analyze, so Python's design, which stores critical information in the heap, makes Python difficult to analyze. A compiler is only capable of compiling Python if the compiler can support Python's complex semantics. Supporting Python's complex semantics causes the compiler and its associated algorithms to become complex. A search of the literature did not reveal any reports of successful static compilation of the entire Python language, although there are several compilers that can handle subsets of the language.

Psyco [22] is a just-in-time (JIT) compiler that hooks into the Python interpreter and compiles Python code at run time. JIT compilation used by the Psyco compiler works well with Python because it requires no prior knowledge about a program before program execution. Psyco uses information discovered as the program runs to optimize the program. The performance improvement offered by Psyco varies widely depending on the application. In some cases it offers no improvement, and in a synthetic integer benchmark it offers a 109x speedup. In our experience, Psyco offers a ~2x speedup in most applications.

Pyrex [23] is a static compiler that translates specially annotated Python code into C. The translated code runs faster than if it were interpreted. The C code generated by Pyrex largely consists of bytecode-equivalent API calls to the interpreter. Although this removes the overhead of decoding byte codes, it does not remove the abstraction overhead in Python's object model. This offers 1.1x–2.1x speedup, depending on the application. Some of the object model's abstraction overhead can be removed by Pyrex by unboxing value objects such as integers. Unboxing can improve performance for arithmetic operations, because the generated C code can perform the arithmetic operations without using Python's object model. Unboxing allows a 288x speedup on a synthetic integer benchmark. Pyrex does not perform significant analysis on the Python code that it compiles but relies on programmer-supplied annotations to obtain an understanding of the program during compilation.

PyPy [24] is a Python interpreter written in Python. Python cannot be directly executed on a CPU and must be interpreted. There is a circular dependency between PyPy's interpreter and the Python language. PyPy breaks this dependency by compiling and translating its interpreter into a language other than Python. PyPy's interpreter is written in a restricted subset of Python called RPython [25] which is restricted to simplify translation. PyPy's static compiler can only handle this subset of Python. PyPy's compiler is designed to translate the interpreter, rather than compile a general Python program. PyPy's compiler analyzes what code it compiles. PyPy performs type inference to determine the type of every variable in a program, formulating the inference as dataflow analysis, which allows the compiler to infer type information that is not present in Python's type system, such as if an integer is strictly positive.

Starkiller [26] is a Python-to-C++ translator that was designed to optimize Python programs and run them directly on the CPU. Starkiller offers a 62x speedup on a synthetic integer benchmark, roughly 2.5x better than Psyco on the same benchmark. Starkiller's analysis is an improved version of the Cartesian product type inference algorithm [27] that has been extended for Python. Starkiller can only analyze a subset of Python; for instance, it cannot analyze exceptions, iterators, dynamic code generation, or tuple unpacking. Starkiller translates an even smaller subset of Python, restricted to functions, control flow, and arithmetic.

Shed Skin [28] is a Python-to-C++ translator, similar to Starkiller. Very little has been published about Shed Skin.

The PyStream platform, which was built for this research, translates and optimizes a subset of the Python language so it can run GPU. None of the previously discussed compilers target the GPU. The PyStream platform contains a static compiler that analyzes the code it compiles. The PyStream compiler formulates its analysis as pointer analysis, unlike PyPy, Starkiller, and Shed Skin, which formulate their analysis as type inference. PyStream's pointer analysis for Python is discussed in Chapter 7.

PyStream is similar to Pyrex in that it is designed to compile performance-critical sections of a larger program. PyPy, Starkiller, and Shed Skin are all designed to compile an entire program. PyStream is similar to Starkiller and Shed Skin in that each is designed to analyze a Python program and compile the program so that it can be run without an interpreter. PyPy compiles code in a similar manner to these other three compilers, but PyPy compiles code with the objective of creating an interpreter. PyStream is the only Python compiler that compiles both the interpreter and the program in tandem to eliminate the interpreter.

Chapter 4

The PyStream Language

The PyStream language is a subset of Python created for writing real-time rendering shaders, a performance-sensitive application. PyStream meets these performance needs because it is designed to be run directly on GPU hardware. Instead of being interpreted, the PyStream compiler translates the PyStream language into GLSL and runs it directly on the GPU. This makes the PyStream language a practical high-performance language for real-time rendering, unlike interpreted Python. The PyStream language is both lexically and semantically compatible with Python.

4.1 Restrictions on Python in PyStream

A computer language cannot do everything. Computer language designs are usually explained in terms of their features, but a language's restrictions are just as important. The design of the PyStream language starts with Python and adds restrictions to make it compilable onto the GPU by removing features from Python. This section discusses PyStream's restrictions on Python.

PyStream's restrictions on Python make it easier to compile. These restrictions can be divided into three groups. The first group improves the precision and speed of subsequent analysis algorithms. The second group allows PyStream to be mapped onto GLSL. The third group resulted from engineering decisions to simplify the construction of the compiler. This third group of restrictions may be lifted later to improve compatibility with Python.

4.1.1 Compilation Restrictions

The first group of restrictions improve analysis speed and precision in the PyStream compiler. There are three of these restrictions:

Immutable Type Pointers. In Python it is possible to change the type pointer of an object and thereby change its type at run time. Immutable type pointers allow an analysis to permanently label an object with its type and to assume that the object label always matches the type of the object. Changing the type of an object exposes interpreter-specific issues of how the fields of one type are mapped to the fields of another type. Immutable type pointers do not pose a significant restriction on Python because programs rarely mutate type pointers.

Long Integers. Long integers are not allowed in the PyStream language, a restriction that ensures performance of integer arithmetic in shaders. In Python, arithmetic operations on integers can produce long integers if the number that is calculated is outside the range that can be represented with a standard integer. Because analysis algorithms generally do not track the value of integers, they must assume that every arithmetic operation on integers can produce either a standard integer or a long integer. This feature is known to cause performance problems when statically compiling dynamic languages that contain it [29]. This restriction is not significant for Python shaders since arithmetic overflows are rare in shaders.

Dynamic Code Creation. PyStream does not permit new code to be created at run time. There are two features in Python which PyStream disallows by this restriction: an `eval` statement that can interpret any string as if it were Python code, and the capability to dynamically import modules, which effectively creates new code at run time. However, most Python programs do not use these features and PyStream does not need these features for shader programming, so this is not a substantial restriction.

No Stack Frames. PyStream does not allow a program to access interpreter stack frames. Stack frames are a data structure that the interpreter uses to track the execution of a program, similar to the stack in C. Python interpreters typically allow accesses to this data structure from within the program being executed, but it does not make sense in PyStream because PyStream's compilation process eliminates the interpreter. Emulating stack frames would add run time overhead and complicate the transformations applied by the compiler without providing significant benefits.

4.1.2 GLSL Restrictions

The second group of restrictions simplify the mapping of PyStream onto GLSL. There are four of these restrictions:

Recursive Function Calls. Any GLSL code generated by PyStream must not contain recursive function calls because recursive calls are not supported in GLSL. PyStream disallows recursive function calls *after* compilation, but they are allowed in the Python shader as long as they can be eliminated by the compiler. Because of Python's heavy use of parametric polymorphism, Python programs typically contain recursive calls to generic functions, particularly in the language run time. In PyStream, what may first appear to be a recursive call may be later eliminated by the compiler with function specialization (function cloning). Recursion can be eliminated in cases where a generic function is being used in distinctly different manners every time it is called. To allow for this, PyStream restricts recursive functions calls *after* compilation. Applying the restriction before compilation would make most Python programs invalid PyStream programs. Tail recursion elimination could also be used to eliminate recursive calls, but it is not implemented in PyStream.

Recursive Data Structures. GLSL disallows recursive data structures. All data structures in GLSL must have a known, finite size. Any GLSL code that the PyStream compiler generates must therefore have no recursive data structures. PyStream makes an exception to this restriction when all objects in a data structure are unique as identified during compilation and when the number of unique objects is finite. Another way of stating this restriction is that PyStream does not allow data structures such as linked lists. This restriction is minor because most recursive data structures are manipulated using recursive functions, which are also disallowed in PyStream.

Standard Libraries. Python has a "batteries included" philosophy and contains a large number of standard libraries so that Python can be used for most tasks without needing to install any additional libraries. Some of Python's standard libraries are not supported by PyStream. This restriction arises from some Python standard libraries calling native code that cannot be emulated in GLSL. For instance, GLSL blocks use of some Python standard libraries on the GPU because GLSL does not support file input/output (IO). PyStream's non-support of libraries containing file IO is not a serious restriction because shaders could not use libraries with file IO effectively. Python standard libraries can be used in PyStream if the library is either a pure Python library contained in the language subset used by PyStream or the library is not a pure Python library but contains functionality that can be emulated on the GPU, such as Python's math library.

Floating-point Numbers. Floating-point numbers are assumed to be 32 bits long instead of the 64 bits used in Python. This is because GLSL does not support 64 bit floating point numbers, although this feature of GLSL may change.

4.1.3 Engineering Restrictions

The third group of restrictions simplify the PyStream compiler. There are five of these restrictions:

Exceptions. In Python, exceptions cause non-local control flow. The PyStream language assumes that exceptions do not occur, which allows simpler algorithms to be used in the PyStream compiler. For instance, accessing an attribute of an object in Python can cause an exception if the attribute is not defined. Proving that an attribute is defined would require the flow-sensitive analysis of heap objects by the PyStream compiler. PyStream assumes that exceptions will not occur, reducing the complexity of the compiler.

Keyword Arguments. Another feature of Python not included in the PyStream language are some Python calling conventions, including keyword arguments, karg arguments, and kparam parameters. These calling conventions all relate to passing arguments by keyword. Supporting keyword calling conventions in PyStream increases compiler complexity without providing significant benefits to shader program development. The use of keyword arguments is largely a matter of coding style and many Python programs do not use it. Some Python programs do use keyword arguments extensively. PyStream's exclusion of keyword argument support in the PyStream language did not cause an issue in the research because a common and simpler coding style was used in the example shaders.

Arguments to Numerical Function Have a Single Type. In the PyStream language, calls to numerical functions must have unambiguously typed arguments. Unlike Python, GLSL is a statically typed language and calls to GLSL built-in numeric functions must have arguments of a known single type. PyStream would not need to impose this restriction if the PyStream compiler inserted switches to select between different built-in functions in the generated code. This change would make the compiler more complex. Using the GLSL restriction made the compiler simpler. In practice, this means that arguments to numeric functions must be well-typed in PyStream. This is not a major restriction since most programs are well-typed, even in a dynamically typed language such as Python.

Class Dictionaries are Immutable. PyStream requires that class dictionaries be immutable. Immutability allows class dictionaries to be flattened down the class hierarchy in a preprocessing step that simplifies the analysis algorithms used in the PyStream compiler, as discussed in Section 7.5.1. This restriction is redundant because all objects passed into a shader are immutable, which is discussed in Section 4.2. Class dictionaries were specifically singled out in this restriction because the PyStream compiler takes advantage of the immutability of class dictionaries.

Portions of the Python Language Not Supported. As an engineering decision, the PyStream language does not support some features of the Python language, such as closures and generators. These features were not needed in the research. There are no engineering, technical, or research reasons that would prevent these features from being supported.

4.1.4 Translation Restrictions

All of the listed restrictions affect the entire compilation process in one way or another. The final step in the PyStream compiler translates Python shaders into GLSL shaders. There are some troublesome issues in this translation process that create de facto restrictions on the Python shaders. These restrictions are discussed in Section 10.7 because they are purely artifacts of the translation process and do not affect the rest of the compiler. At a high level, the restrictions imposed by the translation process are threefold. First, complex, mutable data structures cannot be created inside loops. Second, variable-sized container objects are not supported on the GPU, although common uses could be supported with additional engineering. Third, some built-in Python data structures, such as strings, are not supported on the GPU because GLSL does not provide an easy way to emulate them. In many cases, these scenarios are dealt with by the compiler before they become issues for the translation process. These restrictions are not considered part of the design of PyStream, rather they are topics for future research.

4.1.5 Post-compilation Language Restrictions

The PyStream language formulates some of its restrictions to apply after compilation, not before. Applying restrictions after compilation allows the PyStream platform to support a wider range of programs because in many cases a compiler can eliminate undesirable features from a program. For instance, the use of generic functions in the Python interpreter causes most Python shaders to be recursive. A compiler can eliminate this recursion using function cloning. Pre-compilation restrictions on recursion would result in most Python shaders being immediately rejected as invalid. Post-compilation restrictions cause fewer shaders to be rejected.

PyStream's use of restrictions after compilation is an important difference from the general approach of imposing restrictions before compilation. Most languages define their restrictions in terms of the original program, which allows a compiler to provide a clear, concise description of what caused a restriction to be violated. Imposing restrictions after compilation is a central aspect of the PyStream platform's design that makes it possible to run Python shaders directly on the GPU. If only pre-compilation restrictions were used, almost all Python shaders would be invalid. In addition, Python's use of metaprogramming means there is no static definition of a program, which makes it difficult to link pre-compilation restrictions back to the original program. When using post-compilation restrictions, a programmer must be more attentive to the transformations occurring during compilation. This was not an issue in this research because PyStream is a research compiler and the research required a full understanding of all transformations. An open question is how to assist a programmer to understand more clearly how post-compilation restrictions in the PyStream language map onto the original program.

4.2 Python Shader Programs in PyStream

The restrictions of the PyStream language allow it to be compiled and GLSL code generated for the GPU. These restrictions alone are not sufficient to allow PyStream to take full advantage of rendering-specific hardware on the GPU, however. Python shader programs must be written in a specific format so that Python shader programs can use rendering specific hardware, such as the rasterizer and framebuffer blending. PyStream has a specific format for writing these Python shader

```
# A shader program
class SimpleShader(object):

    # Run per-vertex
    def shadeVertex(self, context, pos, color):
        # Project position onto the screen
        cameraPos = self.worldToCamera*pos
        projected = self.projection*cameraPos

        # Output the projected position
        context.position = projected

        # Output an attribute to be interpolated
        return (color,)

    # Run per-fragment
    def shadeFragment(self, context, color):
        # Output the interpolated color
        context.colors = (vec4(color, 1.0),)
```

Figure 4.1: Example Python Shader Program in PyStream

programs which allows Python shaders to use this hardware. PyStream's shader format is syntactically and semantically compatible with Python.

Figure 4.1 shows a simple shader program written in PyStream's shader format. A shader program contains several shader functions that are used to parametrize different sections of the rendering pipeline running on the GPU. In this case, the shader program contains a vertex shader and a fragment shader. This shader program takes vertices composed of a position and a color. The vertex shader (the part of the shader program that processes each vertex of a 3D shape) transforms and projects the position onto the screen and then passes this position to the rasterizer. The vertex shader also passes the color to the rasterizer, to be interpolated across the triangle. The fragment shader writes the interpolated color into the frame buffer. PyStream's shader format has several necessary features permitting Python shaders to use rendering specific hardware on a GPU. Each Python shader program is implemented in the PyStream language as a Python class. Different shaders in a shader program are implemented as specially named methods of the Python class. For example, the vertex shader of a shader program is implemented in a method called `shadeVertex`. PyStream's shader format was designed so that object-oriented concepts in Python are used to create a format similar to that in GLSL, but with better structure. Different parameters of a shader method are defined as corresponding to different features of a GLSL shader program.

The first parameter of each Python shader method holds a data structure that corresponds to *uniform* variables in GLSL. In the example, this is the `self` parameter. Uniform variables are shared between GLSL shaders. In the Python shader, the first parameter of any shader method naturally corresponds to an instance object of the Python shader program because of Python's method dispatch mechanism. Python data stored in an instance object of a shader program is shared among all shaders and naturally maps to uniform variables in GLSL.

The second parameter of each Python shader method holds a data structure (context object) that corresponds to built-in variables provided by GLSL. In the example, this is the `context` parameter. These built-in variables are typically used to pass data to rendering-specific hardware. Each Python shader method is passed a different type of context object depending on the type of the shader. For example, a vertex shader's context object contains a position field that is used by the rasterizer. The context object held in the second parameter can also expose shader-specific functions as methods, such as derivatives in a fragment shader. In PyStream, the output of each shader is written to a specific field on the context object.

The remaining parameters of each Python shader method correspond to *varying* variables in GLSL. In the example vertex shader, these are the `pos` and `color` parameters. In the example fragment shader, this is the `color` parameter. This type of variable contains data that is streamed into the shader. In GLSL, data transferred between shaders must be a flat list of values with intrinsic types. In PyStream, data transferred between shaders can have structure so long as the structure is consistent. An inconsistent structure cannot be handled by the rendering-specific hardware on the GPU, so PyStream disallows inconsistent structures. For instance, the rasterizer cannot interpolate between a three-dimensional vector and a two-dimensional vector. Similar restrictions are applied to the output of each shader, as the output of one shader becomes the input of the next. Figure 4.2 shows a vertex shader that would be disallowed for having an inconsistent output structure.

4.3 GLSL Intrinsic Functions and Types in PyStream

PyStream reimplements GLSL intrinsic functions and types in Python. For instance, in PyStream there is a Python implementation of the GLSL `vec3` type.

```
def shadeVertex(self, context, pos, condition):
    cameraPos = self.worldToCamera*pos
    projected = self.projection*cameraPos
    context.position = projected

    if condition:
        color = vec3(1.0, 0.0, 1.0)
    else:
        color = vec2(0.0, 1.0)

    return color
```

Figure 4.2: Vertex Shader with Inconsistent Output Structure

Functions that operate on the `vec3` type are mostly implemented as methods in PyStream. Reimplementing functions and types in Python ensures that every Python shader is a valid Python program in its own right. Reimplementation also makes it simpler to analyze a Python shader because GLSL functions and types can be analyzed as Python equivalents. When translating Python shaders into GLSL, the reimplemented functions and types can be replaced with their original equivalents in GLSL.

Two features of GLSL that do not have direct analogues in Python are function overloading and swizzle attributes. Function overloading is where multiple versions of a function are provided, each with different argument types. Python does not have argument types, so function overloading is emulated in PyStream by using "if" statements to explicitly decode the argument types and select the correct implementation of the function. Swizzle attributes are attribute accesses that simultaneously read or write multiple attributes. For example, `t = v.zyx` is equivalent to `t = vec3(v.z, v.y, v.x)`. PyStream emulates swizzle attributes by generating an attribute descriptor for each possible swizzle.

4.4 Advantages of Using PyStream to Write Shaders

Rendering systems can be written in a single code base with the PyStream platform. This simplifies the use of a GPU as a coprocessor. With the PyStream platform, shader code can be written using Python's abstractions. For instance, polymorphism can be used to encapsulate points of variation in a shader with PyStream. In existing shader languages, points of variation are handled by explicitly generating multiple versions of a shader. Because polymorphisms can be mapped onto the GPU with PyStream, points of variation in a shader can then be expressed in a simpler manner. A Python shader program can be written so that the functionality of the shader is subdivided using object composition, which is the defining of program structure with data structures rather than with function calls. In comparison to PyStream, Sh allows the use of object composition to define shaders but Sh does not support run time polymorphism like PyStream.

Object sharing can be used to define how a rendering system controls shaders, and it occurs when multiple shaders have references to the same object. When the shared object is modified, all shaders that use the object incorporate the modified result. For example, shaders often have the concept of a "camera" that spatially defines what portion of the image is being drawn. To draw a coherent image, every shader must agree on where the camera is positioned. When directly using GLSL code, the program must explicitly define a policy on what data is shared between shaders and where it comes from. PyStream allows a simpler expression of this policy because it uses Python data structures and can share objects between data structures.

PyStream has a single code base and shares functions, methods, and data structures between the CPU and the GPU. In existing rendering systems, separate code bases for each processor require that any shared code be duplicated, rewritten, and maintained. Easily sharing code from a single code base between the CPU and GPU is useful. For example, in real-time rendering, it is desirable to have the color of the fog match the color of the background. This allows an object to fade into the background as it fades into the fog. The background color is specified by a CPU and the color of what is drawn is specified by shaders running on a GPU. Complex shaders often process colors in a way such that the color specified for the fog may not match the color that is actually drawn. In this situation, setting the background color based on the fog color results in an image as shown in Figure 4.3a. By sharing code, PyStream allows the CPU to process the fog color in the same way as the GPU shader, allowing the background color to match the fog. The result of applying the same color processing on both the CPU and GPU is shown in Figure 4.3b.

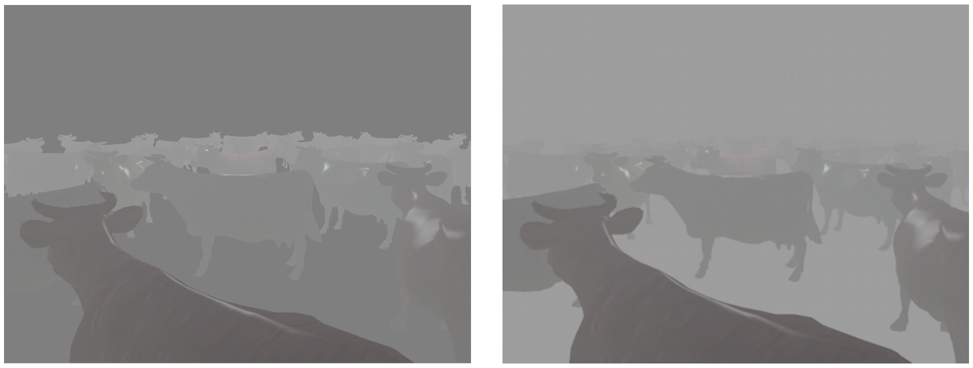

(a) (b)

Figure 4.3: Fixing Fog Color Mismatch

Chapter 5

PyStream's Compiler Architecture

The PyStream compiler is a static compiler, rather than a dynamic compiler as has become fashionable for dynamic languages. GPUs currently do not provide the architectural features that would make dynamic compilation viable for the PyStream platform.

The PyStream compiler turns Python shader programs into ASTs that are used throughout the compilation process. A Python shader program is compiled with a series of passes through the program. Each compiler pass analyzes, transforms, or does both actions on the program. Each pass is implemented as a "type dispatcher" that traverses the ASTs. PyStream uses a number of special AST nodes to represent low-level operations not directly available in Python. PyStream annotates ASTs with the analysis information that it generates. As compilation progresses, PyStream's transformations refine the ASTs until they are refined enough to be transformed into GLSL and run on the GPU. The PyStream compiler essentially consists of a program representation (AST nodes and annotations) and a set of type dispatchers.

Static compilation transforms code to run on a target architecture and does not modify the code after it is initially translated. In contrast, dynamic compilation uses information gained at run time to rewrite the code. GPU APIs do not incorporate an opportunity to halt or inspect execution under their current design. Absence of this feature in the APIs prevents Python from being run on the GPU with current dynamic compilation techniques. Compiling dynamic languages, like Python, has historically been based on dynamic compilation techniques, such as type feedback [22].

The PyStream compiler is written in Python. While a compiler for Python could be written in many languages, PyStream was written in Python for two main reasons. First, writing the compiler in Python for the purposes of this research allowed for rapid development of the compiler because Python itself is a rapid development language. Second, writing the PyStream compiler in Python allows the Python interpreter to be used during compilation. Python exposes the mechanisms to interpret the Python language to the programs being interpreted. PyStream uses these mechanisms to preprocess code and fold constant expressions. The basic architectural features of PyStream are detailed in the following sections.

5.1 The PyStream Compiler Pipeline

The PyStream compiler pipeline is a sequence of steps that are designed to translate Python shader programs into GLSL. The PyStream compiler pipeline has three main logical grouping of steps. First, the compiler front end reads in and preprocesses all of the shader programs that are being compiled. Second, all of the shader programs are analyzed and optimized together simultaneously to remove Python's abstraction overhead. (The shader programs are compiled together because they often share functions, and it is more economical to compile them all at once.) Third, each shader program is then optimized and transformed, isolated from the other shader programs, to complete the translation into GLSL.

Compiler Front End. The steps used by the compiler front end to read and preprocess shader programs are described in Chapter 6.

Analysis and Optimization. The compiler analyzes and optimizes shader programs with the following steps:

1. Initial pointer analysis. Initial pointer analysis discovers the call graph, type information, and pointer information for all of the shaders. This information is not explicitly contained in any shader, but it is needed for subsequent steps in the pipeline. The initial pointer analysis is designed to provide enough information to eliminate both polymorphism and abstraction overhead. Pointer analysis is performed again, later in the pipeline, with greater precision.

2. Method fusion. Python method calls are implemented in two operations. Method fusion transforms method calls into a single operation. Method fusion is done early in the pipeline to make subsequent transformations easier.

3. Escape analysis. A simple escape analysis is performed to bound the lifetime of each object and reduce the amount of read/modify information that

is propagated interprocedurally. Pointer analysis in PyStream is simple by design and it does not bound the lifetime of objects, so escape analysis is done as a separate step.

4. Constant folding and dead code elimination. Pointer analysis discovers that some operations in the shader programs are spurious, and that other operations can be statically resolved. Constant folding and dead code elimination get rid of these operations. Eliminating these operations simplifies the shader programs.

5. Function cloning. Function cloning duplicates functions according to how the functions are used in the shader programs. This duplication splits polymorphic functions into a number of monomorphic functions.

6. Constant folding and dead code elimination. Once functions have been cloned, many of the resulting monomorphic functions can be significantly simplified.

7. Argument normalization. Argument normalization simplifies calls and function declarations wherever possible. Python's full calling conventions are largely used to implement generic, polymorphic functions. After function cloning, the use of these calling conventions usually becomes unnecessary and the calling conventions can be simplified.

8. Exhaustive inlining. Exhaustive inlining consolidates all of the functions called by a shader into a single function. The PyStream compiler exhaustively inlines all functions into the root function of each shader except for functions that correspond to a GLSL built-in function. This prepares the shaders to be translated into GLSL.

9. Constant folding and dead code elimination. Exhaustive inlining exposes new opportunities for simplification.

10. Load/store elimination. Memory operations in a shader program can be eliminated by replacing load operations with local variables that were previously stored or the result of other load operations. Exhaustive inlining exposes many opportunities to do so.

11. Reapplication of pointer analysis. Pointer analysis is performed again to refresh pointer information. Inlining can cause some pointer information to

become outdated. The compiler also increases heap sensitivity during this step so that subsequent optimizations can be more aggressive.

12. Escape analysis. A simple escape analysis is again performed to bound the lifetime of each object and reduce the amount of read/modify information that is propagated interprocedurally.

13. Constant folding and dead code elimination. A final simplification pass is performed to take advantage of information discovered by the reapplication of pointer analysis.

14. Load/store elimination. A final pass is performed, again to take advantage of information discovered by the reapplication of pointer analysis.

Transformation and translation into GLSL. Each shader program is optimized and transformed in isolation to complete the translation into GLSL:

1. Tree transform and reapplication of pointer analysis. Each shader is separated and pointer analysis is re-applied with assumptions that the input data structures are tree shaped and do not alias. These assumptions make it easier to subsequently map the shader into GLSL.

2. Output flattening. The output of each shader is transformed into a tree shape, duplicating objects as necessary.

3. Load/store elimination. Output flattening generates load operations while transforming a shader, and load/store elimination immediately removes the loads from the shader while leaving the transformation intact.

4. Field transform. Unique fields, both in the input and in dynamically allocated data structures, are transformed into local variables. This transformation is analogous to allocating objects on the stack, but is more powerful.

5. Load/store elimination. After the field transform, it may be possible to eliminate some of the remaining memory operations from a shader.

6. Emulating Python reference semantics and translating into GLSL. As a final step, the Python shaders are translated into GLSL shaders. This step is involved and is detailed in Chapter 10.

```
class BinaryOp(Expression):
    __fields__ = 'left:Expression op:str right:Expression'
```

Figure 5.1: An AST Node Declaration

All steps in the PyStream compiler pipeline share a common representation of the program being processed. This program representation is an annotated Abstract Syntax Tree (AST) augmented with special PyStream-specific nodes. Most steps in the compiler pipeline process programs using a mechanism called a *type dispatcher*.

5.2 Abstraction Syntax Trees (ASTs)

Abstract syntax trees (ASTs) are a representation of code used internally by most compilers. ASTs are tree structures where each node in the tree represents a feature in the code. For example, an AST could contain a node representing the addition of two numbers and the two numbers would be children of that node. PyStream uses ASTs to represent code throughout the compilation process, unlike many compilers which transform their ASTs into control flow blocks early in the compilation process. PyStream keeps using AST nodes because it uses term rewriting [30] for code transformation. PyStream contains ASTs for both Python and GLSL which are both based on the same technology.

5.2.1 Specification

PyStream uses metaprogramming to specify the structure of each AST node. Both the field names and the field types of each AST node are declared in a string and transformed into a Python class for use by the compiler.

In Figure 5.1, an AST node for binary arithmetic operations is declared. The AST node has three fields: left, op, and right. The left and right fields contain subexpressions. The op field contains text representing the precise operation.

5.2.2 Generic Traversal

Generic traversal of AST nodes is supported in PyStream through the use of metaprogramming. Generic traversal is a feature that allows arbitrary operations to be applied to the children of an AST node. Methods for performing generic traversal are generated from a AST node's specification string. Generic traversal is used both for rewriting ASTs and analyzing ASTs. In situations where a particular rewrite or analysis does not concern a particular set of AST nodes, then all the nodes can be traversed using the same code.

5.2.3 Symbol Rewriting

Symbol rewriting is the process whereby symbols in an AST are replaced with other AST nodes. Symbols can be inserted into an AST to indicate where information will be filled in later. PyStream uses symbol rewriting when generating code. For example, when a Python shader is compiled into GLSL, a corresponding Python class is also generated that contains glue code to bind the GLSL shader to the GPU. Both the class and the glue code are generated by rewriting AST templates parametrized with symbols.

5.2.4 Shared Nodes

Most AST nodes are uniquely owned by the node that contains them. In some cases, a node is shared by multiple parents. For example, nodes representing local variables are shared between expressions. This makes it simpler to determine that multiple expressions refer to the same variable. On the other hand, this makes term rewriting more difficult because this sharing must be maintained by the rewrite. Another type of shared AST node is the node representing a function. This allows functions to directly refer to each other.

5.3 Special AST Nodes

In most cases, PyStream's AST nodes correspond to Python's AST nodes. PyStream contains a group of AST nodes that that do not follow Python's AST node pattern. These nodes were added because there are features in PyStream's compilation process that do not correspond to features in Python. While transformations are occurring the PyStream compiler uses some intermediate forms that are not available in Python, but are useful to complete the transformation into GLSL. The compilation process is made easier with these special AST nodes.

5.3.1 Existing Object Nodes

Existing object nodes are a generalized form of constant values. An existing object node can point to any object that exists before a Python shader program is run. For instance, existing object nodes can refer to constant strings, type objects, and global dictionaries. Before transformation, an AST only contains existing object nodes that point to constant value objects. Subsequent optimizations of the shader program can create new existing object nodes that reference other types of objects. The PyStream compiler uses existing object nodes for caching the result of predictable memory access patterns. For example, loading a global variable requires that it be looked up in a global dictionary. If this lookup will always return the same object, the compiler replaces the lookup with an existing object node.

5.3.2 Direct Call Nodes

A direct call node is an extended Python call node that contains a reference to the function being called. The call nodes normally contained in Python ASTs may invoke any function, depending on the object being called. This means what function a call node invokes may not be apparent when the compiler first inspects the AST. Direct call nodes explicitly specify what function is invoked. In cases where call nodes only call a single function, the compiler transforms them into direct call nodes. Later in the compilation process, functions may be cloned. In this instance, direct calls nodes can invoke functions that are different from the function associated with the original object. Direct calls make Python code more comparable to C code, since the target of most calls is specified in the AST. Direct calls are also used to transform object model operations, such as adding two objects, into calls to functions that implement these operations. This is further discussed in Section 6.3.

5.3.3 Method Call Nodes

A method call is an AST node that looks up an attribute on an object and then calls the object that was returned by the lookup. Although Python supports method calls, it implements them indirectly and, as a result, one method call spans two operations. PyStream identifies these two operations in an AST and fuses a method call into a single operation using method fusion (Section 8.2).

5.3.4 Type Switch Nodes

A type switch is an AST node that selects between several code blocks depending on the concrete type of an argument that is passed to it. Type switches are similar to Python's "if" statements, except that they select between multiple blocks based on the type of the conditional. PyStream uses type switches to inline method call nodes. A method call may invoke several different functions. A type switch allows a single method call to be turned into several direct calls, and in turn the direct calls may be inlined. A method call cannot be directly inlined if it invokes multiple functions. Similar functionality to type switches can be implemented in an AST using multiple "if" statements with type checks as conditionals. However, representing a type switch as a single AST node allows analysis algorithms to infer the type of the conditional when inside the code blocks contained by the type switch. This can improve analysis precision in many cases.

5.3.5 Low Level Memory Operation Nodes

Python operations do not modify memory directly. An assignment to an attribute results in a chain of function calls causing memory to be modified at the end of the chain. The function that actually modifies memory is implemented by the interpreter and it is not a Python function. If the PyStream compiler cannot directly perceive these memory operations, it is difficult to optimize and translate them; therefore, PyStream lifts these memory operations into Python code so that the compiler can process these exposed memory operations as Python code. To accomplish this, PyStream adds low-level memory operation nodes to the AST. These low-level memory operations include loading, storing, and checking for the existence of fields on objects, and also allocating objects. One would expect to find all of these memory operations in low-level languages, except for checking the existence of a field. The check operation in PyStream returns a Boolean value if the object in question contains the specified field. Python interpreters perform similar checks internally, although they do so indirectly. Doing the check directly improves analysis precision. PyStream does not perform check operations at run time; they are only done during analysis. PyStream check operations are eliminated by the compiler after analysis with function specialization and constant folding.

5.4 Annotations

Annotations are information produced by PyStream's analysis algorithms that cannot be directly stored in AST nodes. This analysis information is stored in an annotation object that is attached to the AST node it concerns. PyStream requires that all AST nodes and all annotations are immutable. By assuming immutability, the compiler can use simpler algorithms when rewriting an AST. An exception to PyStream's rule of immutability is that AST nodes may be reannotated, which allows new analysis information to be added to the AST without completely rewriting it.

5.5 Intermediate Representation

An intermediate representation (IR) is a data structure in which a compiler holds code while it compiles the code. PyStream's intermediate representation is a restricted AST. The restrictions used in PyStream's IR are similar to those used in most IRs. This type of restriction makes compilation easier. The most important restriction used in the PyStream IR is that expression nodes cannot take other expressions as arguments; all arguments must be references. This restriction adopted by PyStream simplifies compilation because the compiler does not need to process nestled expressions.

PyStream uses an AST for its standard IR, rather than the more commonly used control flow graph (CFG). Using an AST complicates certain transformations compared to a CFG. The reason PyStream uses an AST as its standard IR is because the PyStream compiler does source-to-source translation. Using an AST as an IR allows control flow structure to be explicitly represented and preserved.

Since GLSL only allows structured control flow, preserving this structure suits ASTs for the compilation process.

In a few specific circumstances, the PyStream compiler uses a CFG as an IR. The first circumstance is when Python bytecode is being decompiled. The control-flow structure of bytecode is not immediately apparent, so an AST cannot be used. A CFG is used during the intermediate steps of decompilation until an AST can be created. The second circumstance is when performing a static single assignment transformation. A CFG is well suited for this transformation process, although an AST must be reconstructed from the CFG IR.

5.6 Type Dispatcher

A type dispatcher is a specific implementation of the visitor pattern created during research for the PyStream compiler. The visitor pattern, combined with generic traversal, forms the core of most passes in the PyStream compiler. Transformation passes can be implemented by recursively rewriting AST nodes with a visitor. Analysis passes can be implemented in a similar manner, but by traversing the nodes instead of rewriting them. The visitor pattern is integral to the compiler, and any flaws in the implementation of the pattern quickly become apparent.

Python has a standard idiom for implementing the visitor pattern. The idiomatic Python visitor concatenates the string “visit” with the name of the type of the object being visited. It then calls the method with this name attached to the visitor. For example, visiting a “Local” AST node would invoke the “visitLocal” method on the visitor. This idiom has two serious problems that are addressed by using a type dispatcher. First, classes with the same name must have the same visitor method. This is an issue for the PyStream compiler, because the Python and the GLSL ASTs have nodes with the same name. This issue is resolved by using a type dispatcher. The second problem is that types with different names must have different visitor methods. A given visitor may treat a group of types in the same way, such as ignoring them, or invoking a generic traversal. In this situation, a wrapper visitor method must be created for each type within a group. Using a type dispatcher eliminates the need for wrapper methods.

A type dispatcher maps types to methods by explicitly annotating each method with the types being handled. The type dispatcher uses the type object to control dispatch, rather than using the name of the type object. Different types with the

```
class Fold(TypeDispacher):
    @dispatch(ast.BinaryOp)
    def visitBinaryOp(self, node):
        # Process children
        node = node.rewriteChildren(self)
        # Process the OP
        return fold(node)

    @defaultdispatch
    def default(self, node):
        # This node will not be folded,
        # but the children might...
        return node.rewriteChildren(self)

    @dispatch(ast.leafNodes)
    def visitLeaf(self, node):
        # A string constant, or similar.
        # No traversal is possible
        return node
```

Figure 5.2: A Simple Type Dispatcher

same name can therefore be handled separately without complication. With a type dispatcher, multiple types can be associated with a single method which avoids the need for wrappers. A simple type dispatcher for folding binary arithmetic operations is shown in Figure 5.2.

Chapter 6

PyStream's Compiler Front End

The PyStream compiler meets the many challenges of compiling Python. One challenge to compiling Python is that a Python program lacks a static program declaration. Other languages, such as C, have static program declarations. As a consequence of Python's design, a Python compiler must acquire information about a Python program differently from how a C compiler would do so for a C program. The lack of a static program declaration was addressed partly by the front end of PyStream's compiler. The front end is the part of the compiler that acquires information about a program. PyStream's front end uses both new concepts and concepts developed by other efforts to statically compile Python.

PyPy established patterns for a Python compiler front end. The PyPy compiler front end does not directly parse source code. Parsing source code is defined as analyzing the code against a formal grammar and creating an internal representation to use in the compilation process. PyPy uses a Python interpreter during compilation to indirectly parse and initialize a program before extracting the program from the interpreter's memory. PyStream uses a similar method during compilation and adds two unique features to its compiler front end. One feature deals with foreign functions. The other feature "lowers" a Python program so the compiler does not need to deal with the full extend of the Python language.

6.1 Code Preprocessing

PyStream uses a Python interpreter to preprocess code. Python does not have a preprocessor but achieves the same effect with metaprogramming. Metaprogramming is useful when defining or processing a program, but programs that use metaprogramming are hard to analyze. Metaprogramming prevents a closed world assumption from being made since new functions and new classes can be created at run time. PyStream allows metaprogramming to be performed before the program is compiled and then disallows metaprogramming after compilation has begun. This approach allows a closed world assumption to be made by the compiler, but also allows the use of metaprogramming to construct the initial program. Completely disallowing metaprogramming in a Python program would not be practical since class declarations in Python are actually syntactic sugar for constructing a class through metaprogramming. Supporting class declarations requires supporting metaprogramming. A similar approach was used by PyPy.

The result of preprocessing the program is read out of the interpreter's memory using reflection. It is assumed that preprocessing is deterministic and will produce the same result every time it is performed. Each time a PyStream program is executed, its initial memory image is set to the memory image produced by preprocessing the program.

6.2 Decompilation

PyStream produces its ASTs by decompiling Python bytecode rather than parsing Python source code. PyStream's preprocessing step can dynamically construct functions which have no readily available source code. Source code may not be available for some Python modules loaded from disk. Since decompilation is necessary for some functions, PyStream decompiles all functions for the sake of uniformity.

Decompilation is performed in three steps. First, the Python bytecode is split into a CFG. Second, the byte codes are translated from operating on a stack to operating on local variables. Third, structural analysis is performed to reconstruct the AST of the function.

An important feature of PyStream's decompilation is that it is done in a lazy manner: functions are decompiled only as they are requested by the compiler. The initial memory image may contain many extra functions, module imports often import more functions than are needed by an application, and code used during the preprocessing step may not be used during execution. For these reasons, eager decompilation can result in an extremely large number of functions being decompiled compared to the number of functions that are used when the program is executed.

6.3 Operation Lowering

The PyStream compiler front end transforms most operations in the Python language into direct function calls. This process is referred to as *operation lowering*. Python supports a large variety of operations, such as attribute accesses, binary operators, and similar operations. Explicitly supporting all of these operations would increase the complexity of the PyStream compiler. PyStream transforms these operations into direct function calls to reduce the number of operations the compiler must explicitly support. The replacement functions implement the semantics of the original operations. For example, `a+b` is converted into `interpreter_add(a, b)`.

Operation lowering also makes implicit operations explicit. For example, when evaluating an “if” statement, the conditional of the statement is implicitly converted to a Boolean value. Converting an object to a Boolean value can have side effects since in some cases the conversion may call a user-defined method. Operation lowering inserts a direct call before each “if” statement to explicitly convert the conditional. This simplifies the semantics of “if” statements. Similar transformations are made when evaluating loop conditions and when coercing vargs into tuples.

The simplicity of operation lowering belies its importance. Without operation lowering, PyStream's compiler would be significantly more complicated. In effect, PyStream transforms semantics originally embedded in the interpreter into a Python library so that the interpreter can be compiled in tandem with a Python program. This allows the compiled program to run without an interpreter, simplifies optimizing the program, and helps reduce the overall complexity of the compiler.

6.4 Static Single Assignment (SSA) Transformation

PyStream transforms all of the code it processes into the static single assignment (SSA) form [31, 32, 33], which is a straightforward process for Python code. The SSA transform renames variables inside of a function so that each variable is only defined once. For example, Figure 6.1a shows a situation where the variable `a` is defined twice. Although it should be possible to determine that the second definition of `a` is equal to 2, the fact that `a` has two definitions complicates associating a

```
a = 1                  a0 = 1
a = a + 1              a1 = a0 + 1
```
(a) (b)

Figure 6.1: SSA Transformation

value with the variable. Figure 6.1b shows the same figure after the SSA transformation has been applied. The variable `a` has been renamed into two variables: `a0` and `a1`. Having two names makes it simpler to associate values with the variable.

The SSA transformation is complicated if local variables can be modified indirectly. Indirect modification makes it difficult, if not impossible, to determine where a variable is defined. Python does not allow the address of local variables to be taken, as can be done in C. Allowing the address of local variables to be taken allows the local variables to be modified indirectly. Without indirect modification in Python, all uses and definitions of a local variable in Python are explicit and contained within a single function.

It is possible to indirectly affect local variables by gaining access to a Python interpreter's stack frame and modifying the local variables held in the frame. This would make the SSA transformation much more difficult. PyStream's compilation process eliminates interpreter stack frames because emulating them would impose a large performance overhead. Access to stack frames is therefore disallowed in PyStream and indirect modification of local variables is impossible.

6.5 Foreign Functions

PyStream cannot decompile functions unless functions are written in Python. Some functions in the Python run time are written in other languages. For example, in the CPython interpreter, this language is C. Functionality implemented in the Python interpreter, such as the algorithm for adding two objects together, is also not written in Python. The problems introduced by these foreign functions are twofold: the compiler cannot analyze the functions, and once the program is translated, these functions cannot be called from GLSL.

In some cases, foreign functions could have been implemented in Python, but instead they were implemented in another language for performance reasons. For example, the algorithm for adding two objects in Python could be implemented in Python code, but it is performed often enough that interpreters implement it in native code. PyStream replaces these foreign functions with equivalent Python implementations. This type of replacement function is used extensively by operation lowering (Section 6.3).

In some situations, foreign functions could have been implemented in Python if Python supported low-level memory operations. For example, the process of setting a field on an object must eventually call a foreign function to perform the operation. PyStream replaces these foreign functions with equivalent "low-level" Python functions. Low-level Python functions are syntactically identical to Python, but a transformation pass replaces calls to specially named functions, such as "loadAttribute" or "storeArray", with low-level memory operations. PyStream has special AST nodes for low-level memory operations, as mentioned in Section 5.3.5.

Some foreign functions in Python have equivalents in GLSL. For example, Python has a function for taking the square root of a number, as does GLSL. When this occurs the foreign function is replaced with a specially written stub that describes the foreign function's side effects. When the stub is translated onto the GPU, it is replaced with its GLSL equivalent.

Some Python functions correspond to built-in GLSL functions. The compiler does not have a problem analyzing these functions since they are written in Python. Special care must be taken that these functions survive until the final translation step, however. In the final translation step these functions are replaced with their GLSL equivalents. For instance, PyStream contains a pure Python implementation of a three-dimensional floating point vector which is semantically equivalent to GLSL's `vec3` type. The methods of the Python implementation are marked by PyStream to prevent them from being inlined during compilation. In the final translation step, the Python methods are replaced with their built-in GLSL equivalents.

There are foreign functions that are not emulated by the PyStream compiler, including functions that perform file IO, because in general these functions are not supported by PyStream, as mentioned in Section 4.1.2.

Chapter 7

Pointer Analysis for Python

This chapter discusses how PyStream performs pointer analysis. PyStream requires pointer analysis because compiling Python shaders into GLSL shaders requires precise pointer information. Pointer analysis for Python has been previously demonstrated only in limited forms and is usually identified as type inference instead of pointer analysis. Generally, attempts to analyze Python have been impeded by the complexity of Python's semantics, and existing analysis algorithms are only capable of handling a subset of Python.

In contrast, PyStream addresses Python's complex semantics by formulating pointer analysis in a manner that treats Python's complex semantics as code. This approach, developed for analyzing Python shaders in PyStream, shows the potential for handling the full semantics of Python. PyStream's formulation also presents the opportunity to adapt for use in Python pointer analysis the techniques developed for other languages. Once PyStream shifts Python's semantics into code, the resulting pointer analysis algorithm reflects those used for other languages.

Previous work on pointer analysis has focused on languages more widely used than Python, such as C and Java. Pointer analysis for Python is similar to pointer analysis for these languages, but different because Python has a number of idiosyncratic features that must be supported by the analysis. The description of PyStream's pointer analysis algorithm in this chapter focuses on how it differs from pointer analysis algorithms used for other languages.

Pointer analysis determines what objects a reference can point to and what objects a memory operation can access. Information provided by pointer analysis is necessary when performing important transformations in most languages. Figure 7.1 shows an example where pointer analysis would be straightforward. In this example, a pointer analysis algorithm would determine that `a` can point to `obj1`, `b` can point to `obj2`, and `c` can point to either `obj1` or `obj2`.

Starkiller is the only system that thus far is known to have performed pointer

```
a = obj1
b = obj2

if cond:
        c = a
else:
        c = b
```

Figure 7.1: Pointer Analysis Example

analysis on Python code. Starkiller's translation process uses a version of the Cartesian product type inference algorithm [27] that has been extended for Python. The Cartesian product algorithm (CPA) can be thought of as a pointer analysis algorithm that is used to infer types, although it is not usually identified as such. Type inference algorithms infer a single, generalized type for each variable. CPA performs "concrete" type inference by determining the set of objects a variable may point to and thus can be considered pointer analysis. Starkiller's type inference can also be considered pointer analysis for the same reasons. Inferring type information is the goal of both CPA and Starkiller and the precision of pointer information is only a concern insofar as it affects the precision of type information. PyStream must distinguish between two objects of the same type to perform many of its transformations, which requires greater precision than is provided by type inference. For this reason, PyStream's pointer analysis is explicitly formulated as a pointer analysis, and not type inference.

7.1 Topics in Pointer Analysis

A pointer analysis algorithm infers from a program what variables point to what objects and what objects are accessed by memory operations. A pointer analysis algorithm does this without executing the program. If two variables simultaneously point to the same object, the variables are said to *alias*, meaning that there are multiple names that can refer to the same object. Determining if variables may be aliased is important because it indicates that a memory operation using one variable may depend on a memory operation using the other variable. This dependency between memory operations occurs because the operations can affect the same object. Without pointer analysis, the compiler must assume that all

```
x.f = a
y.f = b
c = x.f
```

(a)

```
y.f = b
x.f = a
c = x.f
```

(b)

Figure 7.2: Memory Operation Dependencies

variables alias and all memory operations are interdependent. If a compiler erroneously assumes that two memory operations are independent of each other, it may transform them in a way that is unsound; that is, the transformation may cause an unintended change in the program's behavior. For example, in Figure 7.2a, if x and y alias, then c will contain the value held by b. If the operations are reordered as seen in Figure 7.2b, then c will contain the value held by a.

Information produced by pointer analysis may only partially correspond to what happens when the program is executed. In general, the only way to determine exactly what a program does is to execute it. Pointer analysis must therefore approximate a program's expected behavior when analyzing it. These approximations must not cause the compiler to make unsound transformations. To avoid causing unsound transformations, pointer analysis errs on the side of overestimating what objects a variable can point to. As long as pointer analysis determines all of the objects a variable can point to at run time, all of the potential dependencies between memory operations will be discovered. Overestimating the number of objects that a variable can point to often results in an overestimation of the dependencies between memory operations. Preserving a non-existent dependency between memory operations can prevent certain optimizations but ensures that transformations are sound.

7.1.1 Pointer Analysis Precision

Precision refers to how closely the information produced by a pointer analysis matches the program's behavior when it is executed. Pointer information is guaranteed to include all possible objects that a reference can point to, but it may include additional objects that the reference cannot point to when the program is executed. Different pointer analysis algorithms may infer a different number of additional, unrealizable objects for any reference. Pointer information is termed less precise as the number of unrealizable objects that it indicates a reference can point to increases. Pointer information that is more precise allows a wider range of sound transformations. Directly assessing precision is difficult, particularly for large programs. Determining what aliases can occur in practice would require manually inspecting a program. The result of such an inspection would be hard to quantify since certain spurious dependencies between memory operations would impact compilation more than others.

As a proxy for direct evaluation of precision, a variety of metrics have been used to evaluate pointer analysis algorithms. This dissertation uses the access-per-object (ApO) metric to evaluate the precision of PyStream's pointer analysis. ApO counts how many expressions appear to affect a given object and averages this count across all objects. A higher ApO is correlated with less precision.

7.1.2 Pointer Analysis Concepts and Terms

Pointer analysis algorithms can be described with a common set of concepts and terms. Those relevant to the present research are discussed here.

Assignment Modeling. Assignment modeling determines how a pointer analysis algorithm operates when one variable is assigned to another. There are two primary methods of assignment modeling. Steensgaard-style assignment modeling merges the information about variables when one is assigned to another. Andersen-style assignment modeling keeps the variables separate and copies the information from one to the other. Steensgaard-style analysis can provide better performance, but with less precision than Andersen-style analysis. Andersen-style analysis can be significantly more accurate and it can also be made to run fast enough for most applications [34]. PyStream uses Andersen-style assignment modeling because it offers the needed precision.

Context Sensitivity. Context sensitivity refers to an analysis algorithm's ability to keep dataflow along different call paths separate. For example, if functions A and B both call function C, data passed from A to C would not flow back into B when the program was executed. When a pointer analysis approximates a program's expected behavior, an unrealizable data flow can occur, such as information from A flowing into B via C. Context sensitivity can be improved with a number of approaches. PyStream uses the same approach used in the Cartesian product algorithm (CPA), which will be discussed in Section 7.4.3.

Heap Sensitivity. Heap sensitivity refers to an analysis algorithm's ability to keep separate the information about different objects. Pointer analysis algorithms do not individually track each object that may be allocated by a program. It is not possible to bound the number of objects allocated by a program except in limited cases, making it impractical to individually track objects. Pointer analysis algorithms put a finite bound on the number of objects that are analyzed by allowing a single object to represent a group of objects at run time. Merging objects into a group can cause precision loss if members of the group are used in fundamentally different ways. Heap sensitivity can be improved by careful definition of groups. PyStream's solution for improving heap sensitivity is discussed in Section 7.8.

Field Sensitivity. Field sensitivity refers to an analysis algorithm's ability to keep separate the information about different fields on the same object. PyStream's pointer analysis algorithms are field sensitive. Using a field-insensitive pointer analysis on Python code would result in a large precision loss because a significant amount of data passes through the heap. Discussion of field sensitivity in the literature has usually focused on pointer analysis for C. Pointer analysis algorithms for C are often designed to be field insensitive because C allows fields to be accessed indirectly using pointer arithmetic. If a pointer analysis is field insensitive, then pointer arithmetic otherwise allowed by C can be ignored. Allowing field sensitivity increases the complexity of pointer analysis for C because it then requires that pointer arithmetic be modeled in the analysis. Pointer arithmetic is not supported by Python.

Flow Sensitivity. Flow sensitivity refers to an analysis algorithm's ability to distinguish the order in which operations occur. Flow-insensitive pointer analysis is easier to implement and tends to run faster because the order of operations does not need to be taken into account. Precision is lost when using a flow-insensitive algorithm. Transforming functions into static single assignment (SSA) allows flow-insensitive analysis to produce results that are partially flow sensitive. PyStream's pointer analysis is flow insensitive and uses SSA.

Control Sensitivity. Control sensitivity is an analysis algorithm's ability to distinguish what control flow can occur during execution. An operation in a function may not be executed if control flow does not reach the operation. A control-insensitive pointer analysis assumes that all operations in a function can be executed. PyStream's pointer analysis algorithm is control sensitive (Section 7.9).

Algorithm 7.1 PySteam's Pointer Analysis

```
initalizeEntryPoints()
while dirtyConstraints:
        current = popConstraint()
        current.evaluate()
```

7.2 Architecture of PyStream's Pointer Analysis

The basic structure of PyStream's pointer analysis is straightforward. PyStream's pointer analysis determines what set of objects each variable in a program can point to. Each operation in a program can affect what objects a variable can point to. For example, a variable copy operation such as `a = b` would imply that `a` can point to whatever objects `b` can point to. This relationship between these two variables is represented with a constraint that indicates that `b` is copied into `a` and that `a` can point to, at minimum, anything that `b` can point to.

Points-to information is represented as a "set" data structure that contains all of the objects that a variable may point to during the execution of the program. Each variable in a program, including both local variables and fields on objects, has its own "set" data structure to contain pointer information.

PyStream's points-to information does not actually contain objects in the "set" data structures; rather, it contains symbolic object names. An object is a run time artifact that is created by a program to store data. When PyStream analyzes a program, it must model how the program being analyzed allocates and uses objects. PyStream does not model each object in a program uniquely, because an unbounded number of objects may be created when the program is run. Instead, PyStream's pointer analysis abstracts the structure of the heap and represents objects symbolically. Groups of run time objects are given a single name in the pointer analysis, and references to the single name can potentially correspond to any run time object in the group. During analysis, a run time object is designated to have a single, unchangeable name. This abstraction allows the number of objects considered by the analysis to be bounded. For example, all instances of the type `Foo` could be given the same name, `Instance(Foo)`, by the analysis. In this case, the pointer analysis would not distinguish between different objects with the type `Foo` because they would all be named `Instance(Foo)`. How groups of objects are named is determined by a policy and does not affect the underlying mechanics of the pointer analysis, although it does affect the precision of the results. Different objects' names can be assigned based on the type of the object, the operation at which the object was allocated, the call path to the object's allocation site, and the argument types passed to the object's constructor. As a rule, objects with different types are never given the same name. In addition, there are special cases in which objects with the same Python type must be distinguished and given different names to maintain the precision of the analysis. The technique for naming an object in these special cases is referred to as *extended types* due to how it interacts with a context sensitivity technique borrowed from CPA. Extended types are further discussed in Section 7.8.

The set of objects that a variable can point to starts empty and increases monotonically as constraints are evaluated. For example, in the case of `a = b`, if the set of objects that `b` can point to is ever updated, then the copy constraint is marked to indicate that it must be reevaluated and the constraint is put in a queue. High-level pseudocode for PyStream's pointer analysis is shown in Algorithm 7.1. Constraints in the queue are removed one at a time and reevaluated. Reevaluating a copy constraint updates the set of values pointed to by the target to include all values pointed to by the source. If this update changes the set of values pointed to by the target, then any constraints that use the target variable as a source are marked for reevaluation and put in the queue. In this manner, constraints are evaluated until the pointer analysis reaches a steady state: no new pointer information can be discovered by reevaluating any constraint. Since the number of object names and variables is bounded and points-to information increases monotonically, it is guaranteed that the algorithm will terminate.

Copy constraints are not the only type of constraint used by PyStream's pointer analysis. In PyStream's pointer analysis, a constraint represents an invariant between the pointer information of two or more variables. The effect of each type of constraint used in PyStream's pointer analysis varies widely. The only features that the constraints have in common are that they read and write points-to information and that they are reevaluated using a work queue. There are eight types of constraint used in PyStream's pointer analysis:

Copy. `copy(src, dst)` Copy constraints indicate that the destination variable can point to any object that the source variable can point to. When a copy constraint is reevaluated, the destination is updated to ensure that it includes all of the points-to information of the source. If the destination already contains all of the points-to information from the source, the destination is not updated.

Load. `load(expr, fieldtype, name, dst)` Load constraints resolve what fields can be accessed by a load operation. Three pieces of information are used to determine what fields are accessed: the object being loaded from, the type of the field (this is discussed further in Section 7.6), and the name of the field. The name of the field is typically a constant string object such as `'x'` that is dynamically discovered by the pointer analysis. Dynamic discovery is necessary because many loads occur in run time functions that are parametrized not just by the object being loaded from, but also by the name of the field being loaded. It is often the case that little can be determined about what field is accessed by a load until pointer analysis has begun. Once it is determined that a load operation may access a specific field, a copy constraint is created to copy the field variable into the destination variable. Fields are modeled like any other type of variable by the pointer analysis.

Store. `store(src, expr, fieldname, name)` Store constraints are similar to load constraints, except that they copy data into a field rather than from it.

Allocate. `allocate(expr, dst)` Allocation constraints take a type object as an input, and produce an instance of that type. Allocation constraints set the type pointer on the instance object, but do no other form of initialization. The type object is dynamically discovered, because many objects are allocated at a single point in the Python run time, and calls to that function are parametrized by the type that is being allocated.

Check. `check(expr, fieldtype, name, dst)` Check constraints are similar to load constraints, except that they produce a Boolean value that indicates whether or not an object contains a specified field, rather than loading the field. Check operations are implemented as a constraint so that they can be resolved as a constant, Boolean value whenever possible.

Is. `is(a, b, dst)` "Is" constraints check if one object is the same as another object. The implementation of the "is" constraint is complicated by the fact that the pointer analysis can group multiple objects into a single name. In general, the "is" constraint can precisely determine that two objects are not the same if they have different names and that they are the same if they have the same name and it is known that the name corresponds to a single run time object. In all other cases, the "is" constraint cannot be resolved precisely.

Call. `call(expr, args, varg, dst)` Call constraints determine what function is invoked by a call operation and transfer the arguments of the call operation to the parameters of the function being called and the return value of the function back to the call site. PyStream's call constraints can handle both indirect and direct

```
o.m(a, b)
```

(a)

```
bound_method = o.m
bound_method(a, b)
```

(b)

Figure 7.3: Method Calls in Python

call operations. Because of operation lowering (Section 6.3), most operations in a Python program are call constraints. Calls are implemented as constraints because the potential targets of a call must be reevaluated when new pointer information is discovered. The mechanics of how PyStream implements call constraints is discussed in Section 7.7.

Deferred Branch. `deferedBranch(cond, t, f)` If a function contains a branch, the creation of constraints for operations that occur after the branch is deferred until it can be determined if the branch may be taken or not. (The uses of a deferred branch constraint are discussed in Section 7.9.) A deferred branch constraint monitors the condition used by the branch. If the condition can evaluate as true, the deferred branch creates constraints on one side of the branch. If the condition can evaluate as false, the constraint does the same for the other side of the branch. If it cannot be determined that the condition will always evaluate to either true or false, then the analysis will assume that both sides of the branch can be taken.

7.3 Pointer Analysis Issues in Python

Python's design presents a number of challenges for pointer analysis. Many of these challenges are illustrated in how method calls are resolved in Python. Method calls in Python are lexically similar to method calls in other languages, but there are a number of semantic complications in the method calls that pointer analysis must deal with. The first complication is that a method call is not a single operation but a combination of two operations. Figure 7.3a and Figure 7.3b show how Python reduces a method call into an operation that creates a temporary bound method object and an operation that calls the temporary object. The first operation looks up the method's name as if it were an attribute on an object. The same mechanism is used for looking up every kind of attribute: instance variables, class variables, methods, and arbitrary descriptors. The semantics of an attribute lookup cannot be determined by inspecting a function. Python functions do not contain type information and the meaning of an attribute lookup depends on the type it is performed on. An attribute lookup followed by a call could either call a method or simply read a function out of an instance variable and then call the function. Algorithms that analyze Python must be able to bootstrap (Section 7.4.1) in order to deal with this lack of information.

How Python implements "attributes" is a topic in its own right (Section 7.6). The semantics contain a number of different cases for getting and setting an attribute that must be supported by an analysis algorithm if the algorithm is to be sound. Most operations in Python suffer from similar complications. Even adding two objects together has a few different cases that must be dealt with by a pointer analysis algorithm.

PyStream simplifies its pointer analysis algorithm by shifting most of Python's semantics into code (Section 7.5). Functionality in the interpreter and in the Python run time is analyzed as if it were part of the program. This creates a new set of challenges for PyStream's pointer analysis, because it must be able to analyze Python's complex semantics precisely when they are represented as code. These challenges are illustrated by examining how a method is called. The following example is lengthy, because significant functionality is subsumed within Python that is not readily apparent when examining a program.

Figure 7.4a shows a simplified version of how the Python interpreter implements the operation for getting an attribute. This simplified version delegates the operation to one of the object's methods. This function is straightforward to analyze, but it can create problems when the entire program is considered. Every operation that gets an attribute calls this function. This makes the function *highly* polymorphic. A pointer analysis algorithm for Python must have enough context sensitivity to separate all operations that get different attributes (Sections 7.4.3 and 7.8.1). Another complication is that the `name` parameter will contain a string object. A pointer analysis for Python must be able to precisely map string objects to fields.

Figure 7.4b shows a simplified version of the most common method used to get an attribute. It should be noted that, like all of the functions in this example, this function is extremely simplified and only shows the case that creates bound method objects. A common idiom in the Python run time is to distinguish between distinctly different cases using control flow. This example checks to determine whether the class dictionary contains a descriptor for the attribute in question. If it does not, then other cases are considered. A pointer analysis for Python must be

```
def interpreter_getattribute(self, name):
        getter = classLookup(self, '__getattribute__')
        return getter(self, name)
```

(a)

```
def object__getattribute__(self, name):
        # Get the class dictionary
        cls = load(self, 'type')
        classDict = load(cls, 'dictionary')

        # Is there an attribute descriptor
        # in the class dictionary?
        if checkDict(classDict, name):
                desc  = loadDict(classDict, name)

                # Use the descriptor to get the attribute.
                get = classLookup(desc, '__get__')
                return get(desc, self)
        else:
                ...
```

(b)

```
def function_get(self, instance):
        return method(self, instance)
```

(c)

```
def type_call(cls, *vparam):
        obj = cls.__new__(*vparam)
        obj.__init__(*vparam)
        return obj
```

(d)

```
def type_new(cls, *vparam):
        return allocate(cls)
```

(e)

```
def method_init(self, func, inst):
        self.im_func = func
        self.im_self = inst
```

(f)

Figure 7.4: Simplified Method Lookup

```
def method_call(self, *vparam):
        func = self.im_func
        inst = self.im_self
        return func(inst, *vparam)
```

Figure 7.5: Simplified Method Call

control sensitive (Section 7.9) to avoid merging these distinct cases. Python does not provide operations for directly loading or storing fields. Implementing the Python run time as analyzable functions requires supporting these memory operations (Sections 5.3.5 and 7.6). In Figure 7.4b, `load`, `checkDict`, and `loadDict` correspond to these operations.

Figure 7.4c shows how a function object acts as a descriptor object. If a function object is used as a descriptor, it creates a bound method object holding both the function object and the object from which the attribute is being read. The function object creates the bound method object by calling the method type object. Figure 7.4d shows what occurs when a type object is called, in this case the method type. This function creates a new instance of the type, as shown in Figure 7.4e, and then calls the initializer of the new instance object, as shown in Figure 7.4f. All bound method objects are allocated at a single point in the program. This creates difficulties for heap-sensitive pointer analysis.

Precisely distinguishing between different bound method objects is critical to the overall precision of the pointer analysis. A lack of heap sensitivity with regard to bound method objects creates ambiguity about which method is being called on which object. Traditional techniques for improving heap sensitivity are insufficient, because bound method objects are all allocated at the same point in the program. Path-based heap sensitivity can distinguish between some bound method objects if the paths are long enough to compensate for Python's multiple levels of indirection. Path-based heap sensitivity cannot distinguish between bound method objects created for a polymorphic method call, however, since the path will be the same in each case. PyStream uses extended types (Section 7.8) to allow heap sensitivity in this critical case.

Figure 7.5 shows a simplified version of the function that the interpreter delegates the call to when the bound method object is called. Even if the pointer analysis can provide sufficient heap sensitivity to distinguish between bound method objects, the correlation between `func` and `inst` may be lost once they are loaded from the bound method objects into local variables. This would eliminate any benefits gained from heap sensitive pointer analysis. To make heap sensitivity worthwhile, this function must be analyzed separately for every bound method object. PyStream uses Cartesian product function contexts (Section 7.4.3) to do this.

7.4 Current Techniques Used to Analyze Python

PyStream uses a number of techniques to analyze Python that are shared with Starkiller, PyPy, and Shed Skin.

7.4.1 Bootstrapping

Bootstrapping is a process in which an analysis algorithm generates the information that it needs as it runs. A bootstrapping analysis algorithm requires minimal information about a program to start its analysis. Analyzing Python requires bootstrapping, because the definition of a Python program is contained in a mutable data structure that can change as the program runs, invalidating a priori assumptions. In Python types, calls, and even global variables are resolved dynamically. Variable types are not specified in Python. The call graph of a Python program is only discovered by evaluating the program. Seemingly simple operations, such as addition, can result in arbitrary functions being called. It is only possible to put a very loose bound on the functions that may be called in a Python program before the program is evaluated. Objects that may be created and the fields that may be accessed are also unknown prior to evaluation.

Pointer analysis algorithms that cannot bootstrap thus cannot be applied to Python due to their inability to bootstrap. Analysis algorithms using binary decision diagrams (BDDs) often require that the set of values encoded in the BDD be of a known, bounded size prior to analysis. Only loose bounds can be placed on a Python program before it is analyzed. Thus analysis algorithms that require an explicit enumeration of a call graph [35] are also unusable for Python.

7.4.2 Closed World

A closed world exists when every function that a program can execute is known by the compiler at the start of compilation. Meaningful pointer analysis for Python is impossible without assuming a closed world. A closed world is required because the definition of a Python program can be modified at run time. In Python, a function declaration creates a function object and assigns it to a global variable that may be later modified, meaning that one function may be replaced with another at run time. An open world requires conservative assumptions about the side effects of calling an unknown function. An unknown function could perform arbitrary modifications to every global variable. This means that almost nothing about a Python program's call graph can be determined by a compiler if unknown functions are invoked. Restrictions in the PyStream language that disallow code creation at run time create a closed world.

7.4.3 Cartesian Product Algorithm Contexts

The Cartesian product algorithm (CPA) [27] is a type inference algorithm that creates separate analysis contexts for different invocations of a polymorphic function. CPA does this by separating the analysis of a function for every possible combination of concrete argument types. This approach is referred to in this dissertation as *CPA contexts*. These contexts were originally called *templates*, but this dissertation uses a different term to clarify how the concept links to the larger body of work on pointer analysis.

A polymorphic function behaves differently depending on the types of the arguments that are passed to it. One kind of polymorphism is parametric polymorphism, also known as generic programming, which occurs when a function has a single definition but behaves polymorphicaly. Parametric polymorphism can degrade the precision of analysis algorithms if the algorithms do not keep separate the different uses of the function. Parametrically polymorphic functions are easily created in Python because it is a dynamically typed language.

Figure 7.6 shows a simple example illustrating parametric polymorphism. The behavior of this function depends on the types of its arguments. For example, calling `add(1, 2)` would invoke functionality that adds integers and returns `3`. Calling `add('foo', 'bar')` would invoke functionality that concatenates strings and returns `'foobar'`. The functions sets called by these two examples are dif-

```
def add(a, b):
        return a+b
```

Figure 7.6: An Example of Parametric Polymorphism

ferent. If these examples are analyzed together, the called functions intermingle, and unrealizable dataflow results. For instance, the analysis of the example would indicate that the program may attempt to arithmetically add `1` with `'bar'`, a situation that causes a run time error. Parametric polymorphism is used in the Python run time, in the Python interpreter, and in user code. Any pointer analysis performed on Python must be able to deal precisely with parametric polymorphism.

PyStream's compiler uses the Cartesian product algorithm to increase precision when performing pointer analysis on parametric polymorphic functions. The Cartesian product algorithm separates invocations of a function into every possible combination of parameter types. The Cartesian product algorithm then analyzes each version separately. Separating function contexts in this manner is commonly used when analyzing Python.

Using CPA contexts constrains the design of a pointer analysis algorithm. Determining the type of each function parameter requires that up-to-date pointer information be maintained for each parameter. Some pointer analysis algorithms defer the computation of pointer information until it is requested for a specific variable [34]. This optimization is unlikely to provide any benefit when analyzing Python since most variables in Python are used as function arguments. Using CPA contexts therefore requires that up-to-date pointer information be maintained for virtually every variable.

7.5 Reducing Complexity of Pointer Analysis for Python

Transforming interpreter functions into Python functions (Section 6.5) and reducing the number of operations that must be supported by the compiler (Section 6.3) significantly simplify pointer analysis for Python. Python's easy programmability arises in part from the large number of special cases in its semantics, but the special cases in Python's semantics must be supported by any analysis algorithm, significantly complicating that algorithm. PyStream's pointer analysis implements most of Python's semantics as analyzable functions, significantly simplifying the algorithm. PyStream can deal more effectively with all of Python's special cases because the special cases are not part of PyStream's analysis algorithm. PyStream's pointer analysis algorithm deals mostly with function calls and memory operations and contains little embedded information about Python.

The simplicity of this approach is made possible by PyStream's evaluation of a large number of additional functions. These additional functions within PyStream contain the language semantics that systems such as Starkiller must otherwise embed in their analysis algorithms. As will be seen later in this chapter, PyStream's pointer analysis processes roughly 10 support function contexts for every user function context but only 2 support operations for every user operation.

7.5.1 Python Semantics Embedded in PyStream

Although PyStream lowers many of Python's semantics into functions, PyStream intentionally retains some of Python's semantics in its pointer analysis algorithm in order to enhance analysis precision. The retained semantics are most often found where control-flow loops would otherwise occur in the lowered code. Like most other pointer analysis algorithms, PyStream's pointer analysis merges all iterations of a loop when the loop is analyzed. If PyStream lowered loops into code instead of retaining them in the analysis algorithm, then the loops would be merged and not be resolved with precision. Implementing these loops in PyStream's analysis algorithm effectively allows the loops to be unrolled during evaluation, increasing the precision of the analysis.

One example of a loop from Python's semantics, which PyStream implements in its analysis, is a loop binding a varg object to function parameters. In Python, varg objects are expanded into positional parameters by iterating over the fields of the varg object. Merging the iterations of this loop would prevent arguments from being precisely transferred to parameters. Embedding this loop in PyStream's pointer analysis allows the loop to be unrolled and each argument individually transferred to its corresponding parameter.

A similar loop occurs when Python looks up values in a class dictionary. If a requested attribute is not contained in the dictionary of a class, then the lookup is performed on each superclass of the class until the attribute is found. This lookup can be implemented as a loop that is terminated when the attribute is found. If this loop is lowered into a function, the resulting analysis would not terminate when the first match was found; rather, it would return the union of all matches. At minimum, this would severely reduce the precision of calls to overridden methods, because the overriding method would not mask the overridden method. Adding support for class dictionary lookups in the analysis algorithm, instead of lowering them into a function, improves precision. Precision is improved by allowing each iteration of the loop to be evaluated separately and by allowing for early termination of the loop.

In order to further simplify class dictionary lookups, the PyStream language requires that class dictionaries be immutable. Otherwise, the analysis would need to monitor each class dictionary for changes and reevaluate certain lookups when changes to a class dictionary were detected. Making class dictionaries immutable avoids this complication. PyStream also improves performance by flattening class dictionaries down the class hierarchy. Immutable class dictionaries are acceptable when compiling a shader program, because all objects passed into the shader are immutable.

7.6 Low-Level Object Modeling in PyStream

> "Everything should be made as simple as possible, but not one bit simpler." - Albert Einstein (attributed)

For use in PyStream, a low-level object model for Python was developed. It is a memory model that allows semantically accurate analysis of Python objects. Python has complex semantics for getting and setting attributes on Python objects. If these semantics are not precisely modeled, the resulting analysis is unsound. Using a low-level object model allows for pointer analysis that actually reflects true executions of a program. Before discussing PyStream's low-level object model, it is necessary to discuss how Python implements attributes, and how other analysis algorithms model objects.

7.6.1 Python Attributes

When an attribute is set on an object, the Python run time first checks if a corresponding descriptor can be found in the object's class dictionary. Descriptors are objects that contain methods for getting and setting the attributes of other objects. How getting and setting an attribute is implemented by a descriptor depends on the descriptor. A descriptor can either get or set a specific field, or call arbitrary code. If no appropriate descriptor is found, the Python run time checks whether the object has an associated dictionary. If it does, then the attribute is written into the dictionary. This association of dictionaries with objects allows Python to attach arbitrary attributes, referred to in this dissertation as *ad hoc attributes*, to some objects. Ad hoc attributes can be modified indirectly by a Python program, a feature that complicates analysis. Direct references can be obtained to ad hoc attribute dictionaries during execution, which allows ad hoc attributes to be read and written as if they were not attributes of the object. For example, Python module objects store their ad hoc attributes in the same dictionary that is used by functions inside the module as a global dictionary.

This overall process by which Python handles attributes creates a situation where the nature of an "attribute" in Python is not clearly defined. Python object attributes are a convention of the language rather than a feature of the implementation. This explanation of Python attributes addresses only the most common methods used by Python for setting attributes. Additional methods can be introduced because a programmer can customize what occurs when attributes are set.

7.6.2 Prior Approaches to Object Modeling

PyStream models objects differently from Starkiller and PyPy, which model objects at a high level. To decrease the complexity of the model, high-level modeling only approximates Python's semantics. Starkiller models attributes according to Python's high-level conventions. Operations in Starkiller that get and set attributes are implemented as if they get and set fields on the target object. Starkiller's high-level model does not consider the full range of Python's semantics but adds features to its model that allow Python's semantics to be approximated. For instance, in Starkiller's model, the creation of bound method objects in Python is supported with a special feature and another special feature supports situations where the getting or setting of attributes is customized. Some features of Python's semantics are unsupported by Starkiller. Two examples are the indirect access of ad hoc attribute dictionaries and the use of descriptor objects, except for those that create bound methods.

PyPy models Python attributes in a manner similar to that described for Starkiller. Unlike Starkiller, PyPy's analysis only applies to the RPython language, which is a restricted version of Python. The RPython language is defined in a manner that ends where it begins as being whatever PyPy's analysis can be applied to. PyPy's analysis is therefore sound by definition. The use of high-level object modeling by PyPy impedes expanding the definition of the RPython language because of the complexities of fully supporting Python's semantics with a high-level model.

7.6.3 PyStream's Low-level Object Model

In contrast to Starkiller and PyPy, PyStream uses a low-level modeling approach, which allows it to fully support Python's semantics for object attributes. Starkiller and PyPy support only a subset of Python's object attribute semantics because of approximations inherent in their high-level models. The model used by PyStream is an abstracted version of how the interpreter implements objects. Correspondence to high-level "object attributes" is not even considered by PyStream's pointer analysis. Instead, PyStream models memory operations at the level of the Python interpreter. This is possible only because PyStream explicitly analyzes code from the interpreter and from the Python run time.

Fields in PyStream are divided into four groups: interpreter fields, attribute fields, array fields, and dictionary fields. These groups are described as follows:

Interpreter Fields. Interpreter fields are low-level fields that store bookkeeping information used by the interpreter. For example, the type pointer for an object and dictionaries used to store ad hoc attributes are held in interpreter fields.

Attribute Fields. Attribute fields are low-level fields that hold attributes that have been explicitly declared for a type. These fields correspond to Python's `__slots__` declaration within a class declaration. Declaring a slot in Python creates an attribute field on each instance object and adds a descriptor object to the class dictionary that can get and set the field. Due to a quirk in Python's implementation, a subclass may declare a slot with a name that has been previously declared in a superclass. This can result in an instance having multiple fields that share a single name. To deal with this rare case, attribute fields in PyStream are given a unique name according to their descriptor object, rather than using their declared name.

Array Fields. Array fields are low-level fields that hold the numerically indexed attributes of tuple objects and list objects. For example, tuple objects have explicitly numbered array fields, 0, 1, 2, etc. The length of a tuple is stored in an interpreter field. List objects have a single array field used to hold every object the list may contain. List objects can be dynamically edited and resized inside loops, which makes tracking their precise contents impractical. If the length of a tuple object cannot be determined by the pointer analysis, then the tuple object is analyzed like a list object.

Dictionary Fields. Dictionary fields are two separate low-level fields that are used together in PyStream's pointer analysis to implement dictionary objects. One dictionary field holds dictionary keys; the other holds dictionary values. When a key/value pair is stored into a dictionary, the analysis checks if the key is a preexisting value object, such as a string or an integer. If this is the case, the interpreter is used to hash the key. Hashing creates a number for the key. The key is then stored in a dictionary key field named for the hash. The value is similarly stored in a dictionary value field, also named for the hash. If the hash cannot be determined, storing the key/value pair is treated as if it were ambiguous. An ambiguous memory operation is an instruction in a program that can access more than one field at run time. The mechanics of ambiguous memory operations are detailed in Section 7.6.4. A similar process is used to read values from a dictionary. Modeling dictionaries in this manner allows global variables and ad hoc attributes to be precisely resolved by PyStream's pointer analysis because the hashes rarely collide. In the rare case that they do collide, the analysis continues with a slightly less precise result, referred to as *graceful degradation*.

7.6.4 PyStream's Loadall and Storeall Fields

Loadall and storeall fields are used in PyStream to support ambiguous memory operations for all of the previously described field types. Loadall fields are read by ambiguous loads. Storeall fields are written to by ambiguous stores. These fields are only created when an ambiguous memory operation arises.

Ambiguity can occur when PyStream's pointer analysis cannot determine which field is being accessed or when a single operation could access one of several fields at run time. Both types of ambiguities may occur, for instance, when using Python's built-in `getattr` function. The built-in `getattr` function can read arbitrary attributes from an object. The attribute `getattr` accesses is determined by a string passed to the function. If the string is a constant value, PyStream can precisely resolve the load. If the string is dynamically allocated, PyStream is unable to determine which attribute will be loaded. In that case, the loadall field is loaded instead. The loadall field contains a merged copy of all of the other similarly typed fields. Ambiguous loads do not directly reference all of these fields because new fields may be discovered later. When a new field is discovered, it is simpler to connect it to the loadall field than to every ambiguous load. A similar storeall field exists to support an ambiguous store. Storeall fields copy their contents into every field. The example shaders used for this research do not contain any ambiguous loads or stores.

7.7 Cascaded Call Constraints in PyStream

Pointer analysis in the PyStream compiler analyzes a program by building and solving a network of constraints. One kind of constraint used in pointer analysis is a call constraint. Low-level languages, such as C, have simple calling conventions for which a call constraint is relatively simple to implement. Python has complex calling conventions that make call constraints difficult to implement, especially as a single monolithic constraint. PyStream factors the implementation of a call constraint into three steps, each in its own constraint. PyStream uses multiple constraints to represent a single function call. This feature of PyStream's pointer analysis is referred to as *cascaded call constraints*. This approach allows subproblems to be dealt with separately in sequence and makes it easier for the pointer analysis to incrementalize the reevaluation of a call constraint.

The complexity inherent in evaluating Python function calls in a pointer analysis arises from both the use of CPA contexts and the degree to which resolving a call depends on the heap. The use of CPA contexts requires that a call constraint be reevaluated whenever new argument types are discovered. If new argument types are discovered by the pointer analysis, then the call must be linked to additional CPA contexts. Determining which contexts are called may also depend on the contents of heap objects. For example, the contents of tuples held by a varg are utilized to determine the CPA context. (PyStream's use of operation lowering, described in Section 6.3, guarantees that varg objects will be tuples.) Unpacking the contents of a varg tuple while resolving a call requires checking the length of the tuple and reading each field that in contains. A given call constraint may have several possible varg tuples to consider. PyStream's pointer analysis must reevaluate a call constraint whenever a new varg tuple is discovered or a field of a varg tuple changes. Default arguments and kargs must be unpacked in a similar manner. Python call constraints must monitor all arguments associated with the call and must monitor any object fields discovered while evaluating how the call used varg tuples, default dictionaries, and kargs. The large number of dependencies in a call constraint make a single monolithic call constraint complex to implement, particularly if incremental reevaluation is designed into the pointer analysis.

PyStream's cascaded call constraints are simpler to implement than a single monolithic constraint. *Cascaded* means that the first constraint can create multiple versions of the second constraint and a second constraint can in turn create multiple versions of the third. The first constraint is created for every call site and monitors the argument being called. When the first constraint discovers a new object pointed to by the argument being called, it creates a second constraint for each new object.

A second constraint unpacks the arguments contained in heap objects and maps arguments to parameters. If the object being called is a Python function object, the second constraint monitors the `func_defaults` field of the object. Each second constraint also monitors the varg and the karg, if they exist. A second constraint unpacks all the arguments contained in the heap objects, and for every possible combination of heap objects, it computes how all of the call's arguments are transferred to the invoked function. Each second constraint can immediately determine what function is being invoked, based on the object which the first constraint used to generate the second. If mapping between arguments and parameters is possible, a second constraint creates a third constraint containing the mapping.

Each third constraint connects the call to the CPA contexts that are invoked. The third constraint monitors all the arguments and fields that are mapped to parameters. The third constraint takes the Cartesian product of the parameters to determine which context is invoked. Arguments are transferred to each context's parameters using filtered copy constraints that eliminate all objects whose types do not match the CPA context.

7.8 Extended Types in PyStream

PyStream uses a technique called *extended types* to increase heap sensitivity in known, critical cases. Extended types give critical objects with identical Python types different names in PyStream's pointer analysis, allowing precise resolution of data polymorphism in these cases. Data polymorphism, also known as *generic data types*, is a data type's ability to hold a variety of object types. Without sufficient heap sensitivity, the different uses of a polymorphic data type become merged by the pointer analysis. Extended types keep critical objects separate using a policy, or a set of rules, applied by PyStream's pointer analysis. Starkiller also contains a policy that separates critical data polymorphic objects, but it only covers a single case. PyPy's approach is to infer one type per variable and to explicitly parametrize polymorphic data types. A policy for separating data polymorphism is irrelevant to PyPy's dataflow analysis, as a variable has only a single, parametrized type.

Extended types are called "types" because of how they interact with CPA contexts. Objects with different extended types are not just given different names in the analysis, but they are treated as separate types when CPA contexts are created from parameter types. An example that illustrates PyStream's extended type system is to describe how it deals with bound method objects, which are objects that Python creates when resolving method calls. A bound method object contains a reference to both the object on which the method is being invoked and to the function implementing the method. Bound method objects are a polymorphic data type because they may hold any type of instance object and any type of function object. If PyStream's pointer analysis happened to merge two unrelated bound method objects, the correlation between instance objects and function objects would be lost. In other words, calling a merged bound method object results in every instance object being passed to every function. Without an extended type policy, an N^2 explosion of combinations would be considered by the pointer analysis, almost all combinations unrealizable in practice. When a bound method object is created during PyStream's pointer analysis, PyStream's extended type policy gives the bound method object a type parametrized by both the instance object and the function object. PyStream never merges objects of different types. Parametrizing the bound method object's type allows every combination of instance object and function to be kept separate. Despite the fact that an object's type is parametrized by the types of its fields, extended types are unrelated to field sensitivity, and are purely a mechanism for improving heap sensitivity.

An extended type must provide that none of its parameters, either directly or indirectly, contain a similar extended type. If this occurred, the analysis might create an unbounded number of extended types. For example, the extended type for a bound method object cannot be parametrized with an extended type for another bound method object. If this occurs, the analysis must unextend the type used as a parameter.

PyStream uses the parameter types present in the CPA contexts to create all of its extend types for dynamically allocated objects. PyStream does not pull any information from additional sources. This makes it resilient with respect to low-level transformations that may destroy the high-level structure of a program. Starkiller generates extended types for bound method objects as well. Starkiller pulls the information it uses to generate the extended types from its high-level object modeling (Section 7.6).

PyStream uses extended types in a number of additional situations:

Vparam Tuples Created in Different Contexts. Different CPA contexts will create vparam tuples with distinctly different content. If these tuples are not kept separate, the precision of data passed through generic functions is reduced.

Ad hoc Attribute Dictionaries. If ad hoc dictionaries are not kept separate, ad hoc attributes cannot be precisely correlated with the objects to which they are attached.

Container Objects. Container objects including tuples, lists, and dictionaries are inherently data-polymorphic.

Preexisting Objects. Preexisting objects are discussed in the following section.

7.8.1 Preexisting Object Types

PyStream's pointer analysis keeps preexisting objects separate. Much of the information that determines the semantics of a Python program is stored in the heap during program initialization. Python stores both function and class declarations as data structures. If a pointer analysis cannot distinguish between these data structures, the analysis cannot determine the structure of the program. PyStream distinguishes between these structures by using extended types to distinguish between preexisting objects. This feature of the extended type system even includes treating different constant strings as if the strings had different types and giving them different names in the analysis. This is essential for resolving attribute lookups because this feature causes differently named attribute lookups to be separated by the CPA contexts.

PyStream's policy for separating preexisting objects works well when compiling shaders, as is demonstrated in this research, but may be too aggressive for the broader range of Python programs. PyStream's policy for distinguishing between every preexisting object can create extraneous objects in the analysis. This phenomenon can be observed when applying PyStream's pointer analysis algorithms to PyStone, a common Python benchmark. PyStone contains a list of lists implementing a two-dimensional array. Modeling each of the sublists separately does not improve precision because the sublists are used interchangeably. PyStream did not encounter any such cases when compiling the example shaders. PyStream's extended type policy could be tuned to group preexisting objects in cases where keeping them separate does not improve precision.

7.8.2 Other Techniques for Increasing Heap Sensitivity

One commonly used technique for increasing heap sensitivity, referred to in this dissertation as *k-op object cloning*, keeps objects separate based on the last k calls (where k may be an arbitrary number) that lead to an object's allocation site. This approach cannot replace extended types, but is complementary to it. An example that illustrates why k-op object cloning cannot replace extended types is polymorphic method calls in Python. Separating objects based on the call path to their allocation site, as k-op object cloning does, cannot distinguish between different versions of a polymorphic bound method object, because all versions of the object are allocated at the same point in the program. Extended types can distinguish between these versions of the object. PyStream has the option of using k-op object cloning during analysis, when desired, to increase heap sensitivity in cases not covered by the extended type policy.

7.9 Control Sensitivity in PyStream

Control sensitivity refers to the ability of an analysis algorithm to distinguish what control flow can occur during execution. Control sensitivity is necessary to precisely analyze Python programs because Python programs commonly contain ad

```
class vec2(object):
    def __init__(self, x, y=None):
        if isinstance(x, vec2):
            assert y is None
            self.x = x.x
            self.y = x.y
        else:
            self.x = x
            self.y = y
```

Figure 7.7: vec2 Initializer

hoc polymorphism implemented using Python's control flow constructs. Ad hoc polymorphism occurs when a program provides different implementations of a function depending on the types of the arguments. For example, function overloading in other languages is a form of ad hoc polymorphism. Python does not support function overloading. A common Python idiom is to explicitly decode parameter types inside of a function. Effectively, this is a dynamic version of function overloading. Figure 7.7 shows how the explicit decoding of parameter types is used to emulate GLSL's overloaded constructors for the `vec2` type in a Python reimplementation of the `vec2` type. If the `vec2` constructor is passed a `vec2` object, it behaves like a copy constructor. If the `vec2` constructor is passed two floating point values, it initializes the new instance's fields with each argument.

When the Python implementation of the `vec2` constructor is analyzed without control sensitivity, otherwise unrealizable dataflow appears in the analysis. Using CPA contexts will cause the two cases to be analyzed separately, but this is not sufficient to maintain precision. Unrealizable dataflow appears because both blocks of the "if" statement are evaluated without regard for the value of the conditional. This has the same effect as if all versions of an overloaded function in a C program were merged together before being analyzed. In the case where `vec2` behaves as a copy constructor, a pointer analysis without control sensitivity will indicate that `self.x` may point to a `float` or to a `vec2`, and `self.y` may point to a `float` or `None`. In actuality, `self.x` and `self.y` can only point to floats. In the case where `vec2` is passed two floating point numbers, the analysis would provide the correct pointer information, but also indicate that the function attempts to read the non-existent attributes `x` and `y` from the `x` argument. The more ad hoc polymorphic cases there are in a function, the more the precision declines. The idiom of explicitly decoding types to implement ad hoc polymorphism occurs repeatedly throughout the Python run time. This idiom occurs in critical sections of code such as resolving an object's attribute.

PyStream is affected by this problem to a greater degree than other Python compilers, such as Starkiller, because PyStream analyzes the Python run time as code. Starkiller effectively contains specialized versions of each ad hoc polymorphic function in the run time and has a policy that indicates when each version should be applied. Although this approach improves Starkiller's precision in targeted cases, it does not improve precision when dealing with arbitrary ad hoc polymorphic functions. Embedding all of this information in Starkiller's analysis algorithm also increases the complexity of its algorithm.

PyPy handles ad hoc polymorphism by adding multimethods to Python for use in its interpreter. A multimethod is analogous to an overloaded function, separating ad hoc polymorphism into a collection of methods. An analysis algorithm can then select among these implementations as it resolves an invocation to a multimethod. This avoids the need for control sensitivity because ad hoc polymorphism is not implemented with control flow.

PyStream uses control sensitivity, rather than explicit specialization or multimethods, so that it can precisely resolve a wide range of ad hoc polymorphic functions. PyStream is able to do this without embedding large amounts of language-specific knowledge into its analysis algorithm. PyStream implements control sensitivity by performing pointer analysis on an operation only when there is a proven possibility that it may be executed. Determining if an operation will be executed must be allowed in the design of the analysis algorithm. For example, to precisely resolve an "if" statement in a control-sensitive pointer analysis will require that the conditional evaluate to a constant Boolean value. This means that conditions used to implement ad hoc polymorphism must evaluate to constant Boolean values, which in turn entails that some form of constant folding must be performed.

7.9.1 Dynamic Constant Folding

Dynamic constant folding is constant folding that occurs while a pointer analysis is run. Control sensitivity is impossible in Python without the use of dynamic constant folding. In general, constant folding cannot be performed dynamically during pointer analysis, because it can create an unbounded number of objects

```
if isinstance(x, (vec2, vec3, vec4)):
        ....
```

(a)

```
isinstance = getGlobal(func, 'isinstance')
t0 = getGlobal(func, 'vec2')
t1 = getGlobal(func, 'vec3')
t2 = getGlobal(func, 'vec4')
tup = buildTuple(t0, t1, t2)
cond = isinstance(x, tup)
bcond = bool(cond)
if bcond:
        ....
```

(b)

Figure 7.8: Dynamic Constant Folding Example

when there are loops in the constraint graph. The cases in which PyStream's pointer analysis performs dynamic constant folding are restricted to avoid creating an unbounded number of objects. Dynamic constant folding is performed only when a function's arguments are all preexisting objects and the function depends only on its arguments, does not have side effects, and produces a finite number of values. PyStream's pointer analysis performs dynamic constant folding on a function only if the function produces Boolean values or tuples of a known number of non-tuple preexisting objects. This bounds the number of objects that can be created during analysis, making it safe to perform dynamic constant folding. The values produced by constant folding are considered preexisting objects, allowing the output of one constant folding to be the input for another. Restrictions on dynamic constant folding are designed so that control sensitivity is possible in most precision-critical cases.

Figure 7.8a shows a prototypical example where dynamic constant folding is necessary to enable control sensitivity. In this case, an "if" statement controls the execution of subsequent code based on the type of the object held by x. Although it is not immediately apparent, the example contains multiple operations that must be precisely resolved to determine the resulting control flow. When the example is converted into PyStream's standard IR (Section 5.5), it is broken down into the sequence of operations shown in Figure 7.8b. Resolving the control flow requires that four global variable lookups be resolved precisely and that three function calls be dynamically folded.

The global variable lookups are resolved precisely with the help of extended types. The global dictionary attached to `func` will typically be preexisting and unique, unless it has been replaced at run time. Preexisting objects are kept separate from each other, allowing the global lookups to be precisely resolved. The call to `buildTuple` is folded because all of its arguments are existing objects and the arguments are not tuples. The call to `isinstance` *cannot be folded* if `x` does not point to a preexisting object. It is unlikely that `x` will point to a preexisting object in practice. The call can be resolved precisely, however, since `isinstance` is implemented as calling `issubclass` on `x`'s type object. Type objects are almost always existing objects, allowing `issubclass` to be folded. The call to `isinstance` therefore produces the same result as if it were folded. Finally, the call to `bool` is folded, returning the same constant Boolean object that is passed to it. Control sensitivity can be applied because `bcond` is a constant Boolean value.

In most cases, dynamic constant folding allows for the precise resolution of Python's idiomatic ad hoc polymorphism. Dynamic constant folding is a necessary condition for control sensitivity, since without dynamic constant folding very few conditionals could be precisely resolved. Utilizing CPA contexts is another necessary condition for making control sensitivity worthwhile. CPA contexts force each argument to have a single type, which allows precise folding of type checks.

7.9.2 Field Check Operations

A field check operation is a low-level memory operation that checks if an object contains a specified field. These operations are used in the Python interpreter to implement ad hoc polymorphism depending on what fields an object contains. For a pointer analysis to resolve a field check operation with precision, it must make a determination of whether a field may be defined and whether a field may not be defined. One does not necessarily exclude the other, as pointer analysis considers all possible executions simultaneously. The existence of pointer information for a field implies that the field may be defined, but does not guarantee it. For example, a field may be deleted after it is defined. A flow-insensitive pointer analysis algorithm will not delete the pointer information when a field is deleted. PyStream augments pointer analysis by simultaneously performing a dataflow analysis that determines whether a field can ever be undefined. Finding that a field may be deleted will cause the dataflow analysis to indicate that the field may be undefined.

Combining pointer information with this dataflow analysis allows field check operations to be precisely resolved. PyStream is able to precisely resolve all check constraints in the example shaders used for this research (Section 8.4).

7.10 Quantifying the Results of PyStream's Pointer Analysis

Figure 7.9 shows the result of performing PyStream's pointer analysis on the Python example shaders. Figure 7.9a shows the number of live functions discovered by the pointer analysis and the number of corresponding function contexts. The functions are divided by type:

User. User functions are written by a programmer when creating a Python shader program.

GLSL. GLSL functions have been reimplemented in Python to provide compatibility with GLSL.

Interpreter. Interpreter functions are introduced by operation lowering. For example, the addition of two objects is lowered into a direct call to an interpreter function that implements Python's semantics for addition.

Run time. Run time functions are part of Python but are not part of the Python interpreter. For example, the default functionality for getting an attribute from an object is implemented as a method and is considered part of the run time.

Primitive. Primitive functions implement operations such as adding two integers together. Primitive functions are abstracted versions of the low-level functionality that manipulates Python's data types.

Figure 7.9b shows the number of operations in the functions discovered by the pointer analysis and the number of operations weighted by the number of contexts. This gives a rough idea of the work done during pointer analysis.

7.10.1 Number of Contexts

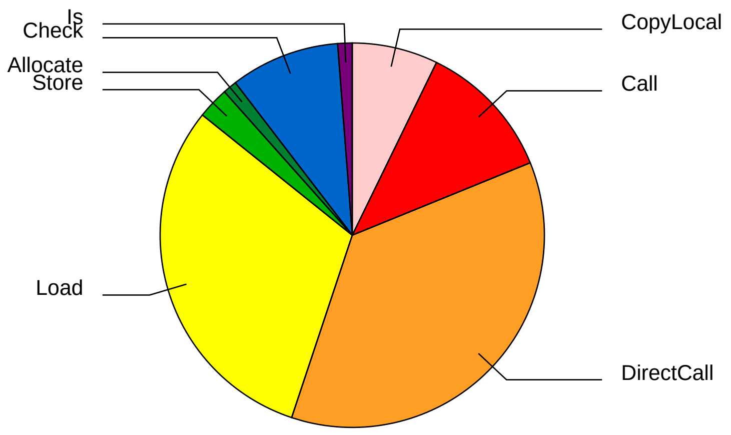
The PyStream compiler creates a large number of function contexts. Many of the functions in the example shaders used in this research are polymorphic. PyStream uses multiple contexts to keep polymorphic instances of a function separate. The transformations applied by the PyStream compiler immediately after pointer anal-

	functions	**%**	**contexts**	**%**	**ratio**
user	47	32.0	69	3.4	1.5
glsl	34	23.1	97	4.7	2.9
interp	22	15.0	565	27.4	25.7
runtime	32	21.8	1234	59.9	38.6
primitive	12	8.2	94	4.6	7.8
total	147	100.0	2059	100.0	14.0

(a)

	ops	**%**	**c ops**	**%**	**ratio**
user	661	30.2	788	4.2	1.2
glsl	1125	51.3	6255	33.1	5.6
interp	260	11.9	4398	23.3	16.9
runtime	134	6.1	7376	39.0	55.0
primitive	12	0.5	94	0.5	7.8
total	2192	100.0	18911	100.0	8.6

(b)



(c)

Figure 7.9: Initial Pointer Analysis Statistics

ysis focus on specializing polymorphic functions. These transformations reduce the ratio of contexts per function by a factor of six (Section 8.4).

The large number of contexts is also necessary to precisely resolve global variables and attributes. PyStream creates a unique context for each unique global variable. PyStream creates, for each unique attribute access, a unique context for three different functions.

Looking at Figure 7.9a, user functions compose 32.0% of the functions analyzed, but only 3.4% of the contexts analyzed. There are two reasons for this. First, the user functions are not polymorphic and do not require additional contexts to maintain precision. Second, multiple interpreter and run time functions are called from each user function, which means that interpreter and run time functions will be called more often during execution than user functions. These function are also highly polymorphic.

7.10.2 Precision

PyStream's initial pointer analysis produces excellent precision for the example shaders. This was determined by manually inspecting the results of the pointer analysis. The call graphs produced were not overestimated. The pointer information produced contains only objects with types that may occur at run time. There were some false dependencies between memory operations. These false dependencies arose because PyStream's heap sensitivity is not particularly aggressive during the initial pointer analysis, outside of the cases when PyStream uses extended types. These false dependencies were easily removed by reanalyzing the program with greater heap sensitivity after abstraction overhead has been eliminated.

Table 7.1a shows the ApO using various levels of k-op object cloning during the initial pointer analysis of the example shaders. The ApO is measured relative to the ApO that can be inferred from precise, concrete type information. The time is measured relative to pointer analysis without object cloning. PyStream's pointer analysis is able to provide better precision than type inference, improving ApO by 62.5% when using only extended types for heap sensitivity. During the initial pass, k-op object cloning is not profitable, providing only a minor improvement in ApO at a large performance cost. Object cloning does not significantly improve ApO because most memory operations in the original program access preexisting

Table 7.1: ApO Improvement

(a) First Pass

k	% APO	Time
0	37.4%	1.00
1	37.3%	1.45
3	35.5%	3.54
5	35.5%	7.60
7	35.2%	10.86

(b) Second Pass

k	% APO	Time
0	57.9%	1.00
1	46.1%	1.71
3	44.1%	2.44
5	44.1%	2.47
7	44.1%	2.49

objects that define the structure of the program, such as type objects and global dictionaries. Extended types already distinguish between these objects. Large amounts of indirection in the original program also reduce the usefulness of k-op object cloning for reasonable values of k.

Table 7.1b shows the same measurements, but taken later in the compiler pipeline, after most abstraction overhead has been eliminated. These measurements are not directly comparable to the first set of measurements, as the structure of the program has changed. In this situation, object cloning shows a larger improvement in ApO, because memory operations that read the structure of the program have been optimized away, and most of the remaining memory operations access objects that are not handled by extended types. Object cloning is only profitable up to a point, because the call graphs of the example shader programs are shallow due to the transformations that have been applied.

Using a summary-based top-down bottom-up pointer analysis algorithm [34, 36] would further improve precision, reduce the number of function contexts that are needed to maintain precision, and make object cloning cheaper. This type of analysis algorithm carefully controls what information flows backwards through invocations. A version of this algorithm is being developed for PyStream. This is an area of anticipated future research. Early results indicate that this new algorithm is faster and more precise than the algorithm currently used in PyStream. Adapting a new version of this pointer analysis algorithm, one originally formulated for C, to Python has so far been straightforward using the techniques detailed in this chapter.

7.10.3 Importance of Each Feature in PyStream's Pointer Analysis

It would be instructive to disable key features of PyStream's pointer analysis, such as CPA contexts, extended types, and control sensitivity, and see how they each contribute to the overall precision. Unfortunately, these features interact in a highly non-linear manner, and disabling any one of them prevents the pointer analysis from terminating. PyStream treats interpreter and run time functions as part of the program being compiled, and performs pointer analysis on them with the rest of the program. These functions are highly polymorphic, and many of them can take an unlimited number of arguments because of Python's calling conventions. Disabling any key feature of PyStream's pointer analysis causes a large amount of imprecision, and this imprecision is amplified by loops in the call graph running through interpreter and run time functions. Because these functions are highly polymorphic, almost any way that these functions are invoked is valid, which allows an unbounded amplification of spurious data flow. The number of arguments passed to certain functions, such as the one that is invoked when a method object is called, increases in an unbounded manner when any of these features is disabled. No meaningful measurements can be taken when this occurs. The importance of each feature in PyStream's pointer analysis cannot be examined separately; it is the interaction of all of these features that makes analysis possible.

7.11 Chapter Summary

This chapter has discussed pointer analysis for Python, based upon PyStream's pointer analysis applied to shader examples. The results obtained from running PyStream's pointer analysis on Python shader examples contribute to a general understanding of how pointer analysis can be performed on Python.

There are significant advantages to analyzing high-level languages, such as Python, with a low-level approach as used by the PyStream compiler. Existing high-level analysis algorithms, used in Starkiller and PyPy, approximate Python's semantics in an unsound manner in order to simplify the high-level analysis algorithm. Low-level analysis algorithms have a less complex structure and do not encourage unsound simplifications. A low-level approach also makes it straightforward to adapt algorithms formulated for low-level languages to Python but provides minimal high-level information about a program. Because Python's semantics have so many special cases, high-level information, even if entirely sound, may not be completely useful to a compiler because a transformation would need to take into account all of these cases. In some situations, high-level information can be reconstructed by the compiler from the results of a low-level analysis, as discussed in Section 8.2. Reconstructing high-level information after a low-level analysis allows the compiler to selectively ignore unsupported special cases by not reconstructing them.

Whether using a low-level or high-level approach to analysis, the analysis must still support the complex calling conventions of Python. The PyStream compiler demonstrates that Python's function calls can be evaluated using cascaded call constraints. Breaking a call in Python into cascaded steps greatly simplifies implementation of a call constraint for pointer analysis.

A pointer analysis algorithm for Python must give special attention to precisely resolving polymorphism in all its forms. CPA contexts are a common way of dealing with parametric polymorphic functions in Python. In contrast, the existing techniques for dealing with polymorphic data types are less well established. PyStream introduces the concept of extended types for increasing precision when dealing with specific polymorphic data types. Ad hoc polymorphism has received minimal attention as an issue for pointer analysis because it is not an issue in languages that support function overloading, which Python does not. Python's idiosyncratic methods for implementing ad hoc polymorphism present a new problem for pointer analysis. PyStream's compiler demonstrates that these idiosyncrasies in Python can be dealt with using control sensitivity and dynamic constant folding.

An advantage of using a low-level approach to analyzing Python is the opportunity it offers to adapt pointer analysis algorithms from low-level languages such as C. There are many pointer analysis algorithms available for C, but very few available for Python. The preliminary research required to develop PyStream's compiler relied on a simple pointer analysis algorithm and not one of the more sophisticated algorithms available. Using a simpler algorithm helped clarify, as the research progressed, how more sophisticated algorithms can be adapted to Python. A function of this research was to determine how pointer analysis for Python relates to what is currently known about pointer analysis.

Chapter 8

Eliminating Abstraction Overhead in Python

This chapter presents a sequence of transformations that the PyStream compiler uses to eliminate much of Python's abstraction overhead. Abstraction overhead refers to parts of a program that were added to make programming easier, rather than contributing to the program's computational goals. This overhead takes the form of indirection in the interpreter and the run time: indirect function calls, deep call chains, temporary objects, and switches used to implement ad hoc polymorphism. The PyStream compiler must eliminate as much of this overhead as possible for two reasons. First, shaders are a performance-sensitive application. Second, GLSL does not support pointers, making it difficult to implement many forms of indirection. PyStream applies a sequence of transformations, some new and some in common use, that have been engineered to eliminate much of Python's abstraction overhead.

8.1 Direct References to Preexisting Objects in PyStream

If a local variable always points to a preexisting object, PyStream transforms the variable into a direct reference to that preexisting object. Direct references to preexisting objects can significantly simplify Python programs. Python programs are defined as preexisting data structures containing class objects, function objects, global dictionaries, and similar. These data structures are always dynamically traversed by a Python program as they can change at run time. This type of data structure rarely changes, but Python must always assume it can. Direct references to preexisting objects therefore allow dynamic traversal to be bypassed when the traversal always produces the same result.

Transforming variables into direct references is a low-level transformation in the PyStream compiler that subsumes a number of high-level transformations. For

```
def hello():
        print "hello"

def test():
        hello()
```

(a)

```
def test():
        obj = global_lookup(test, 'hello')
        obj()
```

(b)

```
def test():
        <hello>()
```

(c)

Figure 8.1: Direct Reference Example

example, global variable lookup loads a global dictionary from a field on the current function and checks if the global name exists in the dictionary. If it does not, then global variable lookup falls back on checking the "built-in" global dictionary for the same name. If the fields read by a global lookup are not modified at run time, then the variable holding the result of the lookup can be transformed into a reference to a preexisting object. The global lookup is later eliminated because it has no side effects and its output is unused. Techniques that explicitly transform global lookups are unnecessary in PyStream because they are eliminated with this technique.

Figure 8.1a shows how a function call is lexically specified in a Python program. In this example, the function `test` calls the function `hello`. Semantically, the function call is implemented in Python with a combination of a dynamic global value lookup and a call to the result of the lookup, as shown in Figure 8.1b. If the global variable is not modified during execution, then the PyStream compiler can replace the global variable lookup with a reference to an existing object, the `hello` function object. There is no Python syntax for this situation, so Figure 8.1c represents it with angled brackets.

8.2 Method Fusion in PyStream

Method fusion is a Python-specific transformation developed for this research that detects and optimizes method calls. Python does not have a specialized bytecode for performing method calls. This stands in stark contrast to other bytecode-based object-oriented languages such as Java and Smalltalk. In Python, method calls are performed with two separate operations. First, the attribute is resolved, which results in a bound method object if the attribute resolves to a function in the class dictionary. Second, the result of the lookup is called. If the result of the lookup is a bound method object, then the call results in a method call. Method calls can only be inferred by considering all the operations in a function. In addition, the call graph must be taken into account, because it is not guaranteed that an attribute lookup corresponds to a method lookup, even if it returns a bound method. For example, a bound method may be stored in an instance variable.

PyStream uses a transformation, *method fusion*, to fuse method calls whenever it is sound to do so. Soundness is inferred from the presence of specific patterns in the call graph. Method fusion uses a high-level understanding of the semantics of Python's method calls to ensure soundness. A fused method call is implemented as a single, PyStream-specific operation (Section 5.3.3) instead of two separate operations. Fusing method calls simplifies the call graph of the program and eliminates the need for a temporary method object. Method fusion in PyStream has a threefold benefit to the compilation process. First, eliminating the method object in turn eliminates a number of memory operations. Second, consolidating method calls into a single operation simplifies any subsequent transformations. Finally, eliminating the method object makes it easier to precisely resolve method calls in any subsequent reanalysis. There are 91 method calls in the example shaders, and they are all fused using this transformation.

8.2.1 Transforming Method Calls

Method fusion can be broken down into three phases. First, invocations that bind methods and invocations that call methods are identified. Second, cases where method fusion is sound are identified. Third, method fusion is performed.

The call graph of a program is examined to identify invocations that bind methods and invocations that call methods. To increase the applicability of this transformation, invocations between contexts are considered rather than invocations

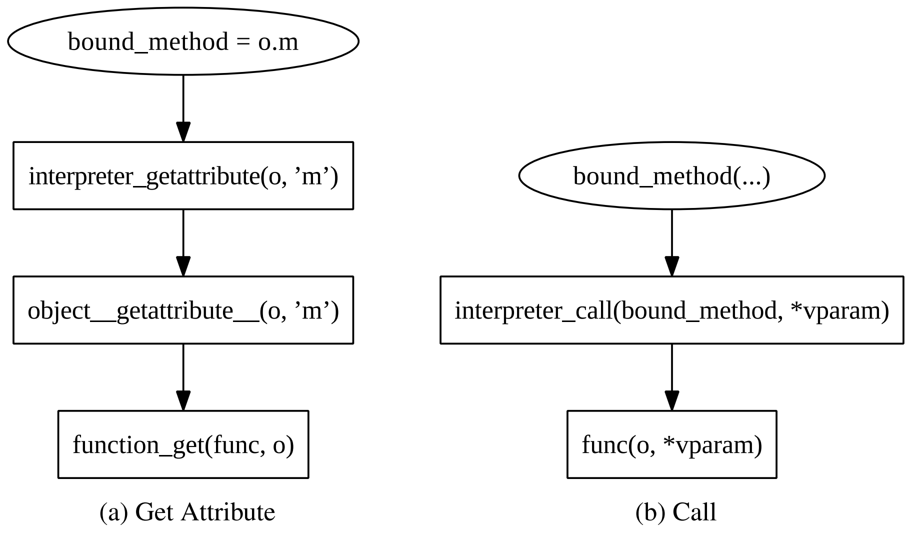

bound_method = o.m
interpreter_getattribute(o, 'm')
object__getattribute__(o, 'm')
function_get(func, o)
(a) Get Attribute
bound_method(...)
interpreter_call(bound_method, *vparam)
func(o, *vparam)
(b) Call

Figure 8.2: Method Fusion Call Chains

between functions. Python's mechanism for getting attributes is highly polymorphic and the precise way it is being used can only be determined by examining the individual contexts.

The analysis identifies unambiguous call chains that correspond to the standard implementations for binding and calling methods. Figure 8.2a shows the standard call chain for creating a bound method object. Creating bound method objects starts with an interpreter operation to get an attribute, which then calls the default object method for getting an attribute, which then uses a function object as a descriptor object and calls the getter method of that function. Figure 8.2b shows the standard call chain for calling a bound method object. Calling a bound method object starts with an interpreter operation to call an object, which then calls a bound method object, which in turn calls the function implementing the method. Any deviation from these expected patterns or ambiguity in the call chain results in the call chain being ignored. Any ambiguity in the data structures used for method lookup also causes the call chain to be ignored.

Intraprocedural analysis is then performed to find cases where a bound method object is created, called once, and never used again. In this case, the binding and the calling operations are fused into a single method call operation, because the transformation is known to be sound. It is worth noting that this transformation changes the order of evaluation for a method call. Python's semantics are such that the method lookup is performed before any of the arguments are evaluated. If evaluation of arguments associated with a method has any side effects that affect method lookup, then this will cause the call chain corresponding to method lookup to be ignored for reasons of ambiguity. Further, method lookup is guaranteed to have no side effects that affect argument evaluation since method lookup only modifies the bound method object, and the bound method object cannot leak in cases where the transformation is applied. Changing the order of evaluation is sound.

8.2.2 Necessity of a Specialized Transformation

Method fusion as used in the PyStream compiler cannot be duplicated with any combination of aggressive low-level transformations. This is an exception to PyStream's overall low-level design. This exception is necessary because method calls are an important feature in Python, but have no visible and concrete implementation when analyzed at a low level. The structure of method calls in Python can be destroyed by low-level transformations because they do not understand higher-level meaning. Knowledge about the structure of method calls in Python enables an aggressive transformation in the PyStream compiler. Method fusion infers and uses high-level information about a program.

Using a low-level approach, if the two operations implementing a method call were fully inlined into a function, then it might be possible to use load elimination to eliminate the use of a bound method object. Method fusion addresses two significant problems with this low-level approach. First, a low-level approach does not guarantee that inlining will produce such a serendipitous circumstance. In fact, unless exhaustive inlining is preformed, or finely tuned heuristics are used, it is highly unlikely that both operations implementing the method call will be fully inlined. Second, the analysis of polymorphic method calls is imprecise unless there is a way to correlate the instance object with the function implementing the method. The use of extended types initially allows this correlation to be maintained by keeping each combination of instance and function separate in the pointer analysis. Load elimination would dismantle this mechanism when it eliminates the bound method object. If the method call is transformed into a single operation, maintaining this correlation is trivial.

Method fusion is low-level optimization that infers high-level information. This allows for better results than using purely low-level optimizations. The low-level formulation of this optimization makes it resilient when dealing with special cases in Python's semantics because high-level information is only inferred in the cases that method fusion can be performed.

8.3 Initial Optimization Results

The transformations both creating direct references to preexisting objects and fusing method calls are not stand-alone optimizations; rather, they enable aggressive constant folding, dead code elimination, and the conversion of method calls into direct calls. All of these transformations taken together reduce the size of the example shaders by half, both in terms of actual operations and in terms of contextual operations. The result of applying these transformations is shown in Figure 8.3. Figure 8.3c shows the percentage of contextual operations that were removed by these transformations. In general, the number of contextual operations removed by various transformations better reflects the elimination of polymorphism and abstraction overhead than the number of actual operations removed. For example, it is easier for a compiler to implement two load operations that access one field each, than it is for the compiler to to implement one load operation that can access two different fields. Measuring the number of actual operations would not reflect this. In some cases the number of actual operations can increase while the number of contextual operations can decrease significantly.

8.4 Function Cloning in PyStream

PyStream uses function cloning to create specialized copies of polymorphic functions. Function cloning uses a number of heuristics to determine when duplicating a function is worthwhile. The method used in PyStream does not differ significantly from that described by Plevyak and Chien [37]. Function cloning is quite important to the PyStream compiler, because the ensuing specialization eliminates large amounts of overhead stemming from the use of polymorphism.

For example, Figure 8.4a shows an ad hoc polymorphic function, `vec2_add`, being called from the function `test` with two different type signatures. Figure 8.4b shows the result of cloning `vec2_add` into two specialized versions, `vec2_add_0` and `vec2_add_1`. Because each version is no longer used as a polymorphic function, the control flow used to implement ad hoc polymorphism can

	functions	%	contexts	%	ratio
user	47	33.1	69	4.8	1.5
glsl	34	23.9	97	6.7	2.9
interp	20	14.1	413	28.7	20.6
runtime	29	20.4	768	53.3	26.5
primitive	12	8.5	93	6.5	7.8
total	142	100.0	1440	100.0	10.1

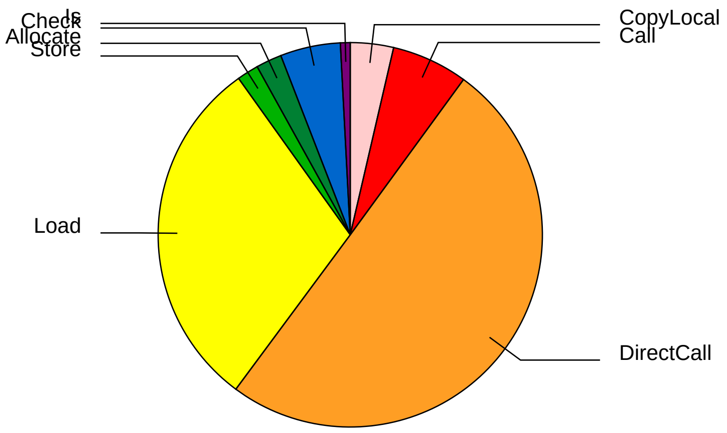
(a)

	ops	%	c ops	%	ratio
user	515	42.0	639	8.5	1.2
glsl	560	45.7	2419	32.2	4.3
interp	69	5.6	1147	15.2	16.6
runtime	69	5.6	3226	42.9	46.8
primitive	12	1.0	93	1.2	7.8
total	1225	100.0	7524	100.0	6.1

(b)

	previous	current	% removed
CopyLocal	1366	273	80.0
Call	2199	482	78.1
DirectCall	6857	3771	45.0
Load	5795	2255	61.1
Store	502	137	72.7
Allocate	222	163	26.6
Check	1740	381	78.1
Is	230	62	73.0
Total	18911	7524	60.2

(c)



(d)

Figure 8.3: Initial Optimization Statistics

```
def vec2_add(self, other):
        if isinstance(other, vec2):
                return vec2(self.x+other.x, self.y+other.y)
        elif isinstance(other, float):
                return vec2(self.x+other, self.y+other)
        else:
                return NotImplemented

def test():
        a = vec2(1.0, 2.0)
        b = vec2(2.0, 1.0)
        c = vec2_add(a, b)
        d = vec2_add(c, -1.0)
        return d
```

(a)

```
def vec2_add_0(self, other):
        return vec2(self.x+other.x, self.y+other.y)

def vec2_add_1(self, other):
        return vec2(self.x+other, self.y+other)

def test():
        a = vec2(1.0, 2.0)
        b = vec2(2.0, 1.0)
        c = vec2_add_0(a, b)
        d = vec2_add_1(c, -1.0)
        return d
```

(b)

Figure 8.4: Function Cloning Example

be specialized out of each cloned function. Despite the fact that the number of functions increases, the number of operations remains the same.

Figure 8.5 shows the effects of applying function cloning on the example shaders used in this research. It is interesting to note that despite quadrupling the number of functions, the number of operations only increases by ~30%. This is can be attributed to the extensive elimination of Python's idiomatic ad hoc polymorphism. Ad hoc polymorphism is mostly found in functions that have multiple contexts, so the number of contextual operations is decreased by ~70% even though the number of actual operations is slightly increased. After cloning, all call operations in the example shaders can be converted into direct call operations. All check and "is" operations can also be discarded from the example shaders because their results are constant once polymorphism is eliminated via cloning.

8.5 Argument Normalization in PyStream

Argument normalization is a Python-specific transformation developed for this research that simplifies the use of Python's calling conventions whenever possible. Python's calling conventions (Section 7.7) are useful for implementing generic functions, but most generic functions are completely specialized by function cloning. Most of the remaining vargs, vparams, and similar are therefore vestigial, and can be eliminated with a transformation. Elimination of these features simplifies the program and eliminates a number of memory operations and object allocations. The PyStream compiler simplifies calling conventions with a simple but novel transformation called argument normalization.

Figure 8.6a shows the function that is invoked when a type object is called to create an instance object. After function cloning, the number of variable parameters may be fixed. For example, when calling the `vec2` type object with two `float` values as parameters, the vparam will have a length of two. Figure 8.6b shows the effect of argument normalization if it is known that the function has exactly two variable parameters.

Argument normalization is an intraprocedural transformation. The transformation first looks for functions with a vparam parameter containing tuples of a single known length. This means that the parameter can hold more than one tuple, but all of the tuples it holds must have the same finite length. The transformation then checks if the vparam is only used inside the function as a varg, or indexed with

	functions	%	contexts	%	ratio
user	52	8.4	69	5.0	1.3
glsl	43	7.0	97	7.1	2.3
interp	187	30.4	409	29.9	2.2
runtime	322	52.3	700	51.2	2.2
primitive	12	1.9	93	6.8	7.8
total	616	100.0	1368	100.0	2.2

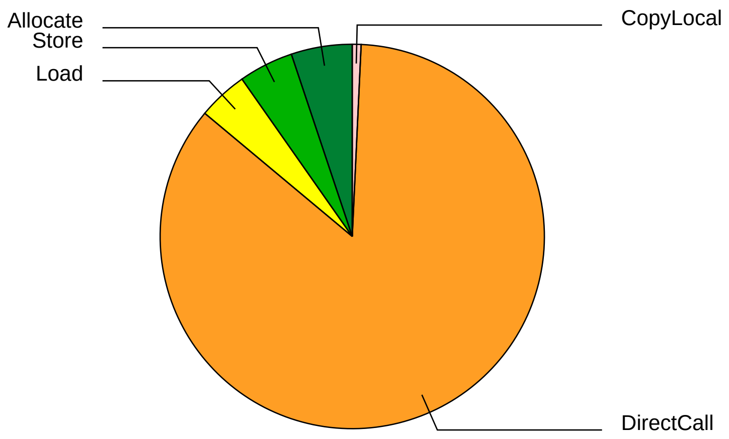
(a)

	ops	%	c ops	%	ratio
user	525	32.1	609	20.4	1.2
glsl	551	33.7	1023	34.3	1.9
interp	202	12.3	436	14.6	2.2
runtime	347	21.2	818	27.5	2.4
primitive	12	0.7	93	3.1	7.8
total	1637	100.0	2979	100.0	1.8

(b)

	previous	current	% removed
CopyLocal	273	22	91.9
Call	482	0	100.0
DirectCall	3771	2542	32.6
Load	2255	125	94.5
Store	137	137	0.0
Allocate	163	153	6.1
Check	381	0	100.0
Is	62	0	100.0
Total	7524	2979	60.4

(c)



(d)

Figure 8.5: Function Cloning Statistics

```
def type_call(cls, *vparam):
    obj = cls.__new__(*vparam)
    obj.__init__(*vparam)
    return obj
```
(a)

```
def type_call(cls, p0, p1):
    obj = cls.__new__(p0, p1)
    obj.__init__(p0, p1)
    return obj
```
(b)

Figure 8.6: Argument Normalization Example

Table 8.1: Argument Normalization Statistics

	Original	**Cloned**	**Normalized**
vparam	5	27	0
varg	4	29	0

constant values. If this is the case, then the function parameters are transformed, eliminating the vparam and adding new positional parameters. The number of positional parameters matches the length of the vparam tuple. The body of the function is transformed by replacing uses of the old vparam with the corresponding new positional parameters.

Table 8.1 shows the number of vparams and vargs in the original shaders, after cloning, and after argument normalization. The vparams and vargs in the example shaders are largely found in the interpreter and run time code. Argument normalization is able to eliminate them all. Argument normalization is quite effective despite the strict conditions for soundness, similar to method fusion. This can be attributed to the predictable nature of most code.

8.6 Exhaustive Function Inlining in PyStream

After argument normalization, PyStream exhaustively inlines functions. Exhaustive inlining is a transformation that consolidates a Python shader into a single function. Function inlining is a transformation that replaces a call site with the body of the function being called. This eliminates function call overhead and allows additional opportunities for optimization. Exhaustive inlining is a transformation that inlines functions until it is no longer possible to do so.

Algorithm 8.1 shows the algorithm that PyStream uses to exhaustively inline functions. The PyStream compiler exhaustively inlines all of a Python shader's functions into the root function of the shader. PyStream prevents functions corresponding to built-in GLSL functions from being inlined so that the functions can

Algorithm 8.1 Exhaustive Function Inlining

```
for function in bottomUp(callGraph):
    for op in function:
        if isDirectCall(op):
            target = invokes(op)
            if notGLSLFunction(target):
                replace(op, body(target))
```

```
def vec2_add_0(self, other):
    return vec2(self.x+other.x, self.y+other.y)

def vec2_add_1(self, other):
    return vec2(self.x+other, self.y+other)

def test():
    a = vec2(1.0, 2.0)
    b = vec2(2.0, 1.0)
    c = vec2_add_0(a, b)
    d = vec2_add_1(c, -1.0)
    return d
```

(a)

```
def test():
    a = vec2(1.0, 2.0)
    b = vec2(2.0, 1.0)
    c = vec2(a.x + b.x, a.y + b.y)
    d = vec2(c.x + -1.0, c.y + -1.0)
    return d
```

(b)

Figure 8.7: Function Inlining Example

be replaced when the shader is translated into GLSL. Exhaustive inlining allows the subsequent transformation of a shader program into GLSL to be formulated as an intraprocedural transformation. However, exhaustive inlining can duplicate large amounts of code. This can reduce performance of a program and make a compiler do more work, but for this research project, reducing the complexity of translating Python into GLSL was worth using exhaustive inlining. In further research, exhaustive inlining may become unnecessary when translating Python into GLSL.

Figure 8.7a shows the result of function cloning, as previously shown in Figure 8.4b. Figure 8.7b shows the result of applying function inlining to this example. Even if a Python shader was not exhaustively inlined, a less aggressive

	functions	contexts	ratio
total	67	208	3.1

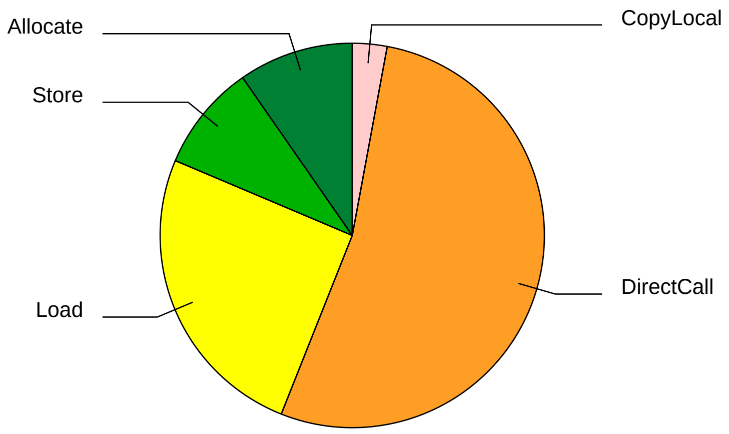
(a)

	ops	c ops	ratio
total	1279	1851	1.4

(b)

	previous	current	% removed
CopyLocal	22	54	-145.5
DirectCall	2542	983	61.3
Load	125	469	-275.2
Store	137	166	-21.2
Allocate	153	179	-17.0
Total	2979	1851	37.9

(c)



(d)

Figure 8.8: Function Inlining Statistics

form of inlining would still be necessary to clean up the call graph after function cloning. Function cloning and the ensuing specialization result in a call graph that contains a large number of small functions that do nothing but call other functions. Most of these small functions are specialized versions of ad hoc polymorphic functions.

Figure 8.8 shows the result of performing exhaustive function inlining on the example shaders. Unlike previous figures, the number of functions and operations are not broken down by function type because functions intermingle once inlining is performed. Function inlining reduces the number of direct calls by ~60%, while increasing the number of loads and stores by ~240%. The overall effect on the shader programs is a ~40% reduction of contextual operations because there are more direct call operations that any other type.

8.7 Chapter Summary

The first set of transformations applied by the PyStream compiler reduces the actual size (number of operations) of the example shaders by 43%. This percentage belies the amount of polymorphism that is specialized by these transformations because, if weighted by the number of contexts, the size of the example shaders is reduced by 92%. This percentage reduction in shader size is more indicative of the importance of these transformations, which were primarily designed to eliminate indirection and ambiguity. Eliminating indirection and ambiguity effectively eliminates abstraction overhead, making it easier to implement memory operations when the shaders are translated into GLSL.

Based on the operations that remain after inlining, it can be inferred that the PyStream compiler is very successful at eliminating abstraction overhead in the example shaders. All calls have been converted into direct calls and many direct calls have been eliminated. In addition, all of the check and "is" operations — which are primarily used to implement ad hoc polymorphism — are eliminated. The PyStream compiler's success at eliminating abstraction overhead is the result of several factors. The PyStream language contains a number of design choices that make it easier to compile than the full Python language. Requiring a closed world and modeling shaders so that all input objects are immutable significantly increase the latitude the PyStream compiler has in applying transformations. Another factor that makes compiling shaders easier is that the example shaders are programmed in a style that favors allocating new objects over modifying existing objects. This style is not particularly onerous to follow and it reduces the number of difficult cases the compiler must deal with. Another factor is that the example shaders do not contain any cases that are particularly tricky for the compiler to deal with. On one hand this indicates that the problem has not been solved in the general case, and on the other hand it indicates that the problem has been solved in a specific usable case.

PyStream uses two language-specific transformations to eliminate part of the abstraction overhead in Python: method fusion and argument normalization. Neither of these transformations can be duplicated without incorporating high-level information about Python into the transformations. Extrapolating results from the PyStream compiler indicates that it may be easier to eliminate abstraction overhead for high-level languages than from low-level languages. Low-level languages may initially have minimal abstraction overhead, but abstractions are often built into a program to compensate for the lack of abstractions in the low-level language. In general, a compiler can be built to eliminate known abstractions provided by a language. If the abstractions are defined within a program, and not the language itself, it is less likely that the compiler can eliminate the abstractions.

Chapter 9

Eliminating Memory Operations in PyStream

The PyStream compiler is designed to eliminate as many memory operations as possible when translating Python into GLSL. The PyStream compiler eliminates memory operations in shaders for two reasons. First, it makes it easier to translate a Python shader into GLSL. References are not supported in GLSL, requiring that reference semantics be emulated in the translated code. Reducing the number of memory operations in turn simplifies the reference semantics that must be emulated. Second, eliminating memory operations makes optimization easier. Dataflow through memory is harder to track than local dataflow. Transforming memory operations into local variables makes potential optimizations obvious to the PyStream compiler because a program's dataflow becomes local.

In general, it is impossible to eliminate every memory operation. However, the PyStream compiler needs only to eliminate memory operations in Python shaders, and not in all programs. When considering Python shaders, there are a number of features in the PyStream language and its restrictions that simplify the solution to eliminating memory operations:

- The PyStream language is strongly typed; that is, the types of objects do not change.
- Python does not allow references to the internal structure of an object, a feature carried over into PyStream.
- Recursion is not allowed in PyStream's shaders.
- PyStream was designed so that objects are copied in and out of shaders.

The combination of strong typing and the absence of references to the internal structure of objects allows for fast and precise field-sensitive analysis. Field-sensitive analysis in turn allows the PyStream compiler to build a sufficiently precise picture of the structure of the heap. PyStream does not allow recursive function calls because it cannot translate them into GLSL. Taking advantage of this recursive function call restriction, PyStream's shaders can be exhaustively inlined, which significantly simplifies the subsequent elimination of memory operations by the PyStream compiler. PyStream's design in which objects are copied in and out of shaders means that transformations that eliminate memory operations do not need to take into account what happens outside of a specific shader. The combination of these features in PyStream make it possible to eliminate the majority of memory operations in the example shaders.

This chapter is an overview of the sequence of transformations and analysis by which memory operations are eliminated in Python shaders by the PyStream compiler. The sequence of steps is:

1. Heap-sensitive reanalysis
2. Load/store elimination
3. Tree transform and re-analysis
4. Output flattening
5. Field transform and object analysis

9.1 Heap-Sensitive Reanalysis

Before load/store elimination is performed, pointer analysis is re-applied to the Python shaders, but with a greater heap sensitivity than previously used. During reanalysis, objects are named according to the three call sites immediately preceding the site at which the object is allocated. Additional heap sensitivity during reanalysis allows the compiler to distinguish between all objects that are not allocated in loops, which aids the elimination of memory operations. This increased level of heap sensitivity was not used during previous pointer analysis. Before Python programs are transformed, they allocate most objects at a single site and have very long call paths through Python interpreter and run time functions. Function cloning and inlining duplicate the single allocation site and reduce the length of the call paths. Before these transformations are performed, achieving a similar level of heap sensitivity in the pointer analysis would require naming objects according to the seven to nine call sites immediately preceding the allocation.

```
obj0.f = data
result = obj0.f

first  = obj1.f
second = obj1.f
```

(a)

```
obj0.f = data
result = data

first  = obj1.f
second = first
```

(b)

```
result = data

first  = obj1.f
second = first
```

(c)

Figure 9.1: Load/Store Elimination Example

Additional heap sensitivity was not needed earlier when eliminating abstraction overhead, so the cost of additional heap sensitivity was previously not worthwhile.

9.2 Load/Store Elimination

Load/store elimination is a transformation that turns load and store memory operations into references to local variables. Load/store elimination is a combination of two transformations: redundant load elimination and dead store elimination. These transformations together significantly reduce the number of memory operations in a Python shader. These are standard transformations, and their effectiveness is boosted by exhaustive inlining and precise pointer information. Figure 9.1a shows two cases where load/store elimination is worthwhile. In these figures, attribute access is used as a shorthand for memory operations, even though there is not a direct correspondence in Python. In the first case, data is stored to a field of `obj0` and immediately loaded. In the second case, a redundant load is made from a field of `obj1`. Figure 9.1b shows the result of applying redundant load elimination in these two cases. In the first case, the load is redirected to copying the variable that is stored into the field, eliminating a load operation. In the second case, the redundant load is redirected to copying the result of the first load, again eliminating a load operation. Figure 9.1c shows the result of applying dead store elimination. In the first case, assuming there are no other loads from `obj0` and `obj0` does not escape from the program, stores that affect `obj0` can be eliminated because they will never be read.

Load/store elimination is performed throughout the process of eliminating memory operations. Load/store elimination is interleaved with other transformations to opportunistically eliminate both existing and newly produced memory operations. Existing memory operations are exposed by other transformations, which allows these operations to be removed in situations where load/store elimination was previously unable to remove them. Other transformations generate new memory operations with the expectation that all new memory operations will be removed by subsequent load/store elimination.

The PyStream compiler uses a simple formulation of load elimination. Load elimination in PyStream is performed in straight-line code and through control-flow splits. Load elimination is not performed in the PyStream compiler through merges, which is a more complex formulation of load elimination. When compiling the example shaders used in this research, the simpler formulation was sufficient because the field transform (9.6.2) was able to eliminate memory operations in the cases where the simpler formulation of load elimination was not sufficient. For a production compiler, a more aggressive form of load elimination would be desirable to ensure that more cases could be dealt with. Code hoisting is also an option to regain any precision lost by control flow merges and further boost the effectiveness of load elimination.

The sequence of transformations described in this chapter interact synergistically with load elimination. Eliminating some of the memory operations subsequently allows the elimination of other memory operations by the PyStream compiler.

9.3 Characterization of the Remaining Memory Operations

Exhaustive inlining combined with load/store elimination significantly reduces the number of memory operations in the example shaders but some still remain. There are four groups of remaining memory operations: loads from input objects, stores to output objects, operations using local variables carried through merges, and operations on objects allocated inside of a loop.

The first two groups of memory operations cannot be eliminated from a shader using load/store elimination because these operations load fields that are defined outside of the shader and store fields on objects that leak outside of the shader. Loads from input objects and stores to output objects cannot be eliminated because the transformations cannot see the operations that create or consume the corresponding fields.

The third group is memory operations on variables carried through merges.

Variables carried through merges sometimes produce ambiguous analysis information because the analysis cannot distinguish which branch of the merge produced the variable. This lack of information limits the effectiveness of redundant load elimination since there may be more that one store that defines the field. In a production compiler, a more aggressive formulation of load/store elimination would be able to deal with many of the variables carried through merges.

The fourth group is memory operations on objects that are allocated inside a loop. No matter how heap sensitive the pointer analysis, it is usually not possible to distinguish between objects allocated in different iterations of a loop. Load/-store elimination can eliminate memory operations in some cases, but in others the lack of object uniqueness prevents elimination. For example, if a store is performed on one version of a non-unique object, then all subsequent loads from any other version of the non-unique object must be performed because the possibility exists that the objects may alias. This issue arises only if more that one version of the loop allocated object can exist at the same time. If there is a unique pointer to the loop allocated object, load elimination functions normally.

9.4 Tree Transform

The tree transform is a re-analysis of the Python shaders that assumes the input data structures to each shader are *tree shaped*, which means that a unique access path in an input data structure always references a unique object. A tree shape eliminates any potential aliases between input objects. If input objects may alias, the tree transform duplicates them. This duplication is sound because input objects are immutable in Python shaders. Eliminating aliases is important for ensuring that the input data structures have a well-defined form. This simplifies implementing the data structures on a platform without pointers.

When transforming the input data structures into a tree shape, the transformation only considers field names as the access path and not the specific type of object that contains those fields. This prevents similarly named fields on polymorphic objects from being duplicated. Objects that the PyStream compiler can determine are unique are not cloned. This means that preexisting objects, including type objects, are not duplicated.

9.5 Output Flattening

Output flattening is a transformation in the PyStream compiler that turns data structures output from a Python shader into a tree shape. Shader output information is used by subsequent operations in the rendering pipeline. For example, a vertex shader produces the screen-space position of a vertex and any attributes that will be used by the subsequent fragment shader. Similar to the issues dealt with by the tree transform, passing arbitrary data structures in and out of a shader is complicated by the lack of pointers and can be simplified by enforcing a tree shape on the data. Unlike applying the tree transform on input data structures, output flattening is more than just an assumption about the shape of the output. Output flattening adds operations to the shader to reshape the shader's output. Output flattening makes the tree shape implicit and reduces the output to a flat list of variables with built-in types. Output flattening generates a series of load operations to accomplish this. Objects that may be shared are cloned. This is a sound transformation because the output objects are immutable in subsequent stages of the rendering pipeline.

Load elimination is then performed after output flattening to eliminate the loads that were generated and the stores that were previously used to build the output structures. Mappings from the flat list of outputs back into the original data structures are maintained because the output of one shader must be linked to the input of the next shader.

The PyStream compiler transforms the output data structures of a Python shader into a flat list of variables with built-in types because GLSL does not support transferring structures between shaders. PyStream requires output data structures to have an unambiguous structure, in part to help this flattening to occur. PyStream primarily requires unambiguous output data structures because functionality of the rendering pipeline requires it, such as the rasterizer. A rasterizer cannot interpolate between dissimilar data structures.

Figure 9.2a shows an output of a vertex shader that generates a basis of three three-dimensional vectors and passes them to the fragment shader. This is simplified from one of the example shaders used in this research. Figure 9.2b shows the code after exhaustive inlining. Figure 9.2c shows the code after output flattening, but before load/store elimination. Figure 9.2d shows the code after load/store elimination.

Due to a quirk of PyStream's escape analysis, stores made redundant by out-

```
...
tsbasis = TangentSpaceBasis(normal, tangent, bitangent)
return tsbasis
```

(a)

```
...
tsbasis = allocate(TangentSpaceBasis)
tsbasis.normal = normal
tsbasis.tangent = tangent
tsbasis.bitangent = bitangent
return tsbasis
```

(b)

```
...
tsbasis = allocate(TangentSpaceBasis)
tsbasis.normal = normal
tsbasis.tangent = tangent
tsbasis.bitangent = bitangent
out0 = tsbasis.normal
out1 = tsbasis.tangent
out2 = tsbasis.bitangent
return out0, out1, out2
```

(c)

```
...
return normal, tangent, bitangent
```

(d)

Figure 9.2: Output Flattening Example

put flattening cannot be immediately removed because the analysis believes that the stores affect objects that may escape the shader. After output flattening, these object are no longer used and do not escape, but pointer information has not been updated to reflect this. Statistics are not given for output flattening in this dissertation, as output flattening combined with load elimination combined with stale pointer information results in virtually no change in the number of operations in the shaders. Output flattening clarifies data flow to the outputs of a shader, but does not immediately remove allocations and stores that were previously used to build output structures. A subsequent transformation removes these operations indirectly by transforming the fields into locals, and then immediately eliminating them because they are unused.

9.6 Field Transform and Object Analysis

The field transform turns fields of unique and mutually exclusive objects into local variables. To make this transformation, the field transform needs information about which objects are unique and which objects are mutually exclusive. Object analysis is performed before the field transform to provide this information.

9.6.1 Object Analysis

Object analysis is done in PyStream before the field transform for the purposes of identifying unique and mutually exclusive objects. Unique objects are objects in the pointer analysis that correspond to only one object at run time. Mutually exclusive objects are objects that cannot exist at the same time during run time. Identifying unique and mutually exclusive objects in turn identifies fields that can be transformed into local variables. Unique objects are identified during object analysis by two means. Objects contained in a Python shader's input data structure are unique due to the tree transform. Objects allocated inside of a Python shader are unique as long as they are only allocated by a single operation that is not contained inside a loop.

Mutually exclusive objects are identified during object analysis, again with different means for input data structures and objects allocated inside the shader. If an ambiguous reference exists in an input data structure, such as occurs when a reference holds a polymorphic object, then the objects held by that reference are

```
def shadeFragment(self, context, color):
    modulated = self.material.color*color
    return modulated,
```

(a)

```
def shadeFragment(self, context, color):
    modulated = field0*color
    return modulated,
```

(b)

Figure 9.3: Field Transform Example

mutually exclusive. This can be inferred from the fact that each object held by a reference must be unique and only one of the objects may exist when the shader is run. Mutual exclusion is determined for objects allocated inside of a shader based on the mutual exclusivity to the control flow that leads to the point at which they are allocated.

9.6.2 Field Transform

The field transform in PyStream converts the fields of unique objects into local variables. Transforming each field directly into a local variable is insufficient because there are cases in which a field can be loaded from more than one object. In these instances, the load would need to be transformed into an "if" statement that selects between multiple local variables. To only create necessary "if" statements, the field transform identifies memory operations that affect multiple fields that are mutually exclusive and unique. The field transformation then merges these mutually exclusive fields into a single field before transforming it into a local variable. If an accessed field is not mutually exclusive from every other field accessed by a memory operation, then the transformation is not applied. After this transformation is done, local variables that are created may not be in the SSA form, so the SSA transformation is reapplied to the Python shader.

Figure 9.3a shows a simple fragment shader that reads fields from the shader program's uniform data structure. Again, attribute accesses are used as a shorthand for memory operations. Figure 9.3b shows the result of applying the field transform to this shader. The field transform will work even if the material object is polymorphic because the tree transform guarantees that polymorphic objects in the uniform data structure are mutually exclusive and unique.

The field transform effectively flattens the input data structures of a Python shader. This is caused by a transformation of the fields of the input data structures into local variables. Some fields may not be transformed, so the input data structures may not be entirely flat. This means that some of the memory operations loading from input data structures may need to be translated into GLSL.

Similar to output flattening, the field transform maintains a mapping of the original fields onto the new local variables. This mapping is used to connect the shaders in a shader program to one another. The field transform also overlaps with output flattening because the field transform could flatten the output data structures by converting their fields to local variables. This transformation would not necessarily result in a tree shape for the output, so the output transform is an explicit part of the sequence of eliminating memory operations.

9.7 Chapter Summary

After the sequence of transformations detailed in this chapter have been applied by the PyStream compiler, all of the memory operations in the example shaders, except for intrinsic memory operations, are eliminated. Table 9.1a shows the number of memory operations contained in each shader program after the tree transform. ILoad and IStore are categories for intrinsic memory operations, such as loading the field `x` from a `vec3` object. Memory operations that load and store from objects with GLSL intrinsic types are intentionally not eliminated, because they can be efficiently implemented in GLSL. Table 9.1b shows the result of applying the transformations in this chapter. Instances exist where memory operations might not be eliminated by PyStream's transformations, but none of these instances were encountered during research. How these instances would be dealt with when translating them to GLSL is discussed in Chapter 10.

In three identified instances, memory operations may not actually be eliminated and must be translated into GLSL to preserve the semantics of the Python shader program. If two non-mutually-exclusive objects flow into the same variable because of a merge, then any subsequent memory operations loading or storing through that variable cannot be transformed by the field transform. As discussed in Section 9.2, the use of an aggressive load elimination algorithm in PyStream would minimize occurrences of this instance by performing load elimination through merges. If an object is allocated in a loop and multiple versions of

Table 9.1: Shader Program Operations

(a)

	DirectCall	**Load**	**ILoad**	**Store**	**IStore**	**Allocate**
material	105	26	19	14	0	3
skybox	11	5	0	9	0	2
ssao	248	4	0	6	0	2
bilateral	50	9	0	6	0	2
ambient	17	8	0	6	0	2
light	41	13	2	6	0	2
blur	20	2	0	6	0	2
post	33	10	0	6	0	2
total	525	77	21	59	0	17

(b)

	DirectCall	**Load**	**ILoad**	**Store**	**IStore**	**Allocate**
material	103	0	19	0	0	0
skybox	11	0	0	0	0	0
ssao	231	0	0	0	0	0
bilateral	44	0	0	0	0	0
ambient	16	0	0	0	0	0
light	41	0	2	0	0	0
blur	20	0	0	0	0	0
post	33	0	0	0	0	0
total	499	0	21	0	0	0

that object may be simultaneously held by a Python shader, then load/store elimination may not be able to eliminate all of the memory operations. In this second instance, the field transform cannot transform the memory operations because the objects are not uniquely named. In a third instance, if a memory operation is ambiguous in terms of the field accessed, then the memory operation cannot be eliminated. This third instance can occur when reading array fields in a loop, for example.

Chapter 10

Emulating Python Reference Semantics in GLSL

The last step in the pipeline of the PyStream compiler translates Python into GLSL. This last step requires the compiler to implement the semantics of Python on an architecture that does not support features that Python relies on. For example, Python expects that data can be held by reference and GLSL only allows data to be held by value. This is a significant semantic gap. Compilers normally do not bridge significant semantic gaps because the code being compiled is usually written for the architecture on which it will run.

Earlier steps in the PyStream compiler pipeline eliminated as many of the impediments to translating Python into GLSL as possible before the last step. The need for an interpreter is eliminated by compiling the Python interpreter in tandem with the program. Many of Python's semantics are transformed into code, reducing the semantics that must be explicitly translated. Information about a Python shader, previously implicit in a memory image, is made explicit while transforming and specializing the program's functions. Abstraction overhead is eliminated both to improve performance and to simplify what must be later translated. Memory operations are aggressively eliminated because GLSL does not offer the memory model that Python expects. By the last step in the compilation process, many of the features that would make Python difficult to translate into GLSL have been eliminated.

The biggest remaining impediment for the compiler is the semantic gap between the memory model that Python assumes and the memory model that GLSL provides. Python assumes that data can be held by reference and memory can be dynamically allocated. GLSL assumes that data is held by value and has no concept of dynamic memory allocation. This chapter describes how PyStream takes the last step in the compiler pipeline and translates code across the semantic gap, and what PyStream cannot translate across the gap.

10.1 Reference Semantics

Reference semantics is defined in this dissertation as the manner by which a computer language deals with memory. Two widely used forms of reference semantics are those that hold data by reference and those that hold data by value. There are several issues that must be dealt with when translating a language that holds data by reference into a language that data holds by value. The most obvious issue is that aliasing is no longer possible; languages that hold by value copy the data between variables, and modifications to that data is only reflected in the current copy. To emulate the semantics of aliases, there must be a single, central copy of the value. On the other hand, in cases where aliased values are not modified, data can be freely copied. Another, more subtle issue is that languages that hold by value must be able to resolve a field access into an exact memory location before the program is run. In languages that hold by reference, field access can be implemented with pointer arithmetic by simply adding an offset to the pointer from which the field is being read. If aliasing issues can be emulated, then the base pointer becomes constant, which allows the field location to be statically determined in most cases. Emulating field access is complicated in situations where memory operations are inherently ambiguous and may be able to access multiple fields. For example, a loop that iterates over an array will access a different field every iteration using the same memory operation. Emulating array accesses is a straightforward process since GLSL itself contains arrays. Not all ambiguous memory operations are as easy to emulate. For example, Python dictionary lookups generally have no simple equivalent in GLSL if they can access more than one field in the dictionary.

Emulating memory allocation is another issue in languages that hold by value. In effect, variables in GLSL preallocate the space needed to hold a given value. Aliasing issues can make it hard to determine the number of values in a data structure. Loops can also result in data structures of unbounded size. In order to translate code that holds by reference into code that holds by value, the translator must statically and pessimistically overallocate memory to account for the largest possible data structure.

10.2 Volatile Values

Volatility is a term used in this dissertation to describe fields and objects in a shader that are both held by multiple references and can be modified. This means that a volatile value can be modified using one reference and read through another. When a value is volatile, it cannot be freely copied. When a value is not volatile, it can be freely copied. Thus, volatility is important.

The PyStream compiler implements objects and fields formerly held in Python by reference as values in GLSL. The PyStream compiler copies values whenever possible, a feature making it easier to subsequently optimize the generated GLSL shader. Volatile values cannot be copied, but they can be identified. If a field of an object is modified while the object is held by multiple references, then copying the corresponding value would prevent the modified field from being visible through other references. In cases such as these, the field is volatile. Volatility is determined by the PyStream compiler on a per-field basis, and not per object, so that as much of the object as possible can be copied.

Whether a field can be modified is determined by inspecting information produced by the pointer analysis. To increase precision, the PyStream compiler only uses modification information if it is either the result of an explicit store or the result of an assignment to a swizzle attribute, which is a feature in GLSL that simultaneously reads or writes multiple fields of an intrinsic type. Built-in GLSL functions implicitly modify an object when the function creates an object, but the functions never modify existing objects. Objects created by built-in GLSL functions are uniquely held until they are returned, so any modifications to an object before it is returned do not affect volatility. It is therefore safe to ignore the side effects of built-in GLSL functions when analyzing volatility.

Determining if an object is uniquely held requires counting the number of references to the object inside of a given shader. If the number of references is greater than one, the object is not uniquely held. The fact that multiple versions of a field may exist at run time does not need to be taken into account since an object may only be modified if it is also held in a local variable. This implies that any object used in a shader is not uniquely held if it is held by a field.

Objects that are in fact uniquely held during execution may not be marked as uniquely held by the analysis. The method used by the PyStream compiler to determine if an object is uniquely held is conservative. The method counts the number of variables that hold a given object without taking into account the life-

Table 10.1: Object Classifications in the Example Shaders

	Multiple Refs	**Uniquely Held**
Mutable	0	0
Immutable	45	307
Samplers	0	22

time of each of the variables. If the lifetimes of all of the variables do not overlap, then the object is uniquely held in practice, despite being held by multiple variables. This situation was not encountered in the example shaders.

The PyStream compiler uses a flow-insensitive pointer analysis that does not distinguish between iterations of a loop (Chapter 7). Such a pointer analysis is unable to distinguish situations in which an object is uniquely held when it is modified, and then later loses its uniquely held status. In this case, using a more aggressive pointer analysis could allow additional fields to be declared non-volatile. In a production compiler, the use of a flow-sensitive pointer analysis algorithm or iteration disambiguation [38] could distinguish instances where a value is uniquely held when it is modified, but later shared between variables. This situation was not encountered in the example shaders.

GLSL does not allow texture samplers to be copied, so the PyStream compiler marks all texture samplers as volatile to ensure that they are not copied. The inability to copy texture sampler in GLSL affects the entire process of translating Python shaders into GLSL, well beyond needing to mark them as volatile. For example, texture samplers are treated as a special case when generating glue code to transfer texture samplers from Python into GLSL.

Table 10.1 shows how all the objects in the example shaders are classified. There are no mutable objects in the example shaders once they have reached the last step in the compiler pipeline, due in part to PyStream's aggressive elimination of memory operations and in part to the coding style used in the shaders. The aggressive elimination of memory operations also has synergies with the SSA form, resulting in most objects being held by only a single reference. The net effect is that there are no volatile values in the example shaders other than the texture samplers. The last step in PyStream's compiler pipeline has been designed to accommodate volatile values, but in practice they seem to be rare after optimizations have been applied.

10.3 Implementing References

The approach used by the PyStream compiler to preserve Python's reference semantics in GLSL follows an unexpected pattern. One would expect that reference semantics should be preserved by preserving object structure, because programs are generally described by their object structure. The PyStream compiler directly preserves reference semantics and the effects of accessing fields through a reference. PyStream does not concern itself with preserving object structure during translation. This approach, for emulating Python's reference semantics in GLSL, exposes more opportunities for optimization as each accessed field can be implemented separately for each reference. In contrast, if object structure was preserved, then there would be only a single, central copy of each field in a shader.

A local variable in Python is implemented in GLSL as a data structure that can contain up to three distinct components, depending on how the variable is used. These components allow a local variable to support Python's reference semantics in GLSL. The three possible components of a local variable, when it has been implemented in GLSL, are:

TypeID. If a variable can point to more than one type of object, then the data structure contains a typeID field to indicate what type of object is currently being held by the variable. Object types are immutable in PyStream so it is safe to embed the type of an object in the variable that holds it. If a variable only contains a single type, then the typeID field is implicitly a constant value and is not explicitly implemented.

Objects with Intrinsic Types. If a variable can point to an object with an intrinsic type, then the data structure contains a field for every intrinsic type of all objects that the variable can hold. Although certain intrinsic types could be packed within others, PyStream does not pack them. For example, a three-dimensional vector could be packed into the same field that could hold a four-dimensional vector. PyStream does not pack these types because multiple intrinsic types are never held by the same variable in the example shaders.

Fields on Objects with Non-Intrinsic Types. If a variable can point to a non-intrinsically typed object that contains fields which are accessed in the shader, then the data structure implementing the variable contains a field for each field that can be accessed. Each field is implemented in GLSL with the same type of data structure that is used to implement local variables since they are both references. Variables can have nestled substructures. PyStream restricts the use of recursive data structures in a Python shader which keeps nestled data structures finite in the GLSL shader. As previously shown in Figure 9.1b, all non-intrinsic memory operations in the example shaders are eliminated before translation. Since these operations were eliminated, PyStream's implementation for emulating fields has not been completely tested.

10.4 References to Volatile Values

This section details how the PyStream compiler implements references to volatile values in GLSL. A GLSL data structure that implements a reference can contain fields that holds a volatile values. These volatile values cannot be directly embedded in the GLSL data structure and are held indirectly so that they can be shared between references and modified. Values that are not volatile are directly embedded in the GLSL data structure and copied whenever the reference is copied. Fields that are not volatile are effectively inlined into the local variable.

In a GLSL shader, a volatile value is held indirectly in a pool. A pool is an array containing a group of volatile values. A field that contains a volatile value contains an index into the pool array instead of the volatile value itself. A volatile value can either be a field from an object without an intrinsic type or the entire object if the object has an intrinsic type.

A pool array in GLSL must be of a fixed size due to the limitations of GLSL. It is necessary for the PyStream compiler to bound the number of different values a pool may contain, so that the size of the underlying array can be bounded. The size of a pool can be bounded by counting the number of values in the pool. If the pool array only contains objects that are not allocated in loops, then the objects can be counted directly. If some of the values are loop allocated, then the number of loop iterations must be bounded and the values weighted and counted by the maximum number of loop iterations. The PyStream compiler does not contain heuristics to bound the number of loop iterations, so objects allocated inside loops must be nonvolatile for the translation process to occur.

All objects contained in a pool may be mutually exclusive. Methods for determining if objects are mutually exclusive are discussed in Section 9.6.1. A pool containing mutually exclusive objects that are not loop-allocated is called a *singleton pool* because the pool array only needs to hold a single value. Because the pool array contains a single value, references to a singleton pool do not need to retain an index into the array.

When pools are implemented, they are paired with an integer that indicates the number of values that currently reside in that pool. When a new value is added to the pool, it is placed at the indexed position indicated by the integer and the integer is incremented by one.

PyStream simplifies implementation of pools by marking all of the values contained in a pool as volatile values, even if they are not. By this procedure, PyStream keeps nonvolatile values from being copied in and out of pools. Although copying values in and out of pools may expose more opportunities for optimization in a shader, doing so disrupts counting objects in a pool. A single nonvolatile value could be copied in and out of a pool multiple times. If this occurs, then the value may be duplicated an unknown number of times and counting cannot be done within the pool. Marking all pooled values as volatile prevents any value from being copied in and out of a pool.

Within GLSL, when values are transferred from a CPU to a GPU, these values are initially transferred into immutable memory on the GPU. A GPU exposes both immutable and mutable memory to a shader, but the GPU initially copies only to immutable memory. If any of the transferred values are to be placed in a pool on the GPU by the compiler, then these values must be copied into that pool at the beginning of a shader's execution. Such a pool is defined by the compiler and it will contain both mutable and immutable values. To implement such a pool, GLSL requires that the pool be contained in mutable memory on the GPU. The entire pool must be contained in mutable memory, even though the pool may contain some immutable values. All of the values transferred from a CPU to a GPU are immutable inside a PyStream shader. PyStream generates a GLSL shader which copies into the GPU's mutable memory any immutable values placed in pools.

An exception to the requirement that a GLSL shader copy immutable values into a mutable pool is with texture samplers. Texture samplers are discussed in Section 10.2. Texture samplers are immutable values that cannot be copied, according to a specification in GLSL. As a consequence of this specification, pools of texture samplers are also immutable and cannot be copied. An entire pool of texture samplers is transferred to the GPU in final form.

10.5 Implementing Ambiguous Memory Operations in GLSL

The PyStream compiler eliminates as many memory operations as possible before translating a Python shader into GLSL. An ambiguous memory operation — a single load or store that can access multiple fields — is an instance where memory operations are not always eliminated by the PyStream compiler. Ambiguous memory operations must be specifically supported by a compiler during the translation process because a single ambiguous memory operation can access multiple fields, and this affects how field access semantics are implemented in GLSL. For example, iterating over the contents of a tuple results in an ambiguous memory operation. Ambiguous memory operations can be supported by grouping into an array all fields that may be accessed by a specific memory operation. This allows ambiguous memory operations to be implemented in a GLSL shader as an indirection through an array. When arrays (lists and tuples) are accessed ambiguously in Python, then array fields in the low-level object model can be emulated in GLSL by direct lookup of the field index, effectively implementing Python arrays as GLSL arrays. Ambiguous access to other types of fields can be dealt with in a compiler by assigning a constant index to each possible field. In practice, function cloning eliminates all ambiguous access to interpreter and attribute fields as a side effect of specializing polymorphic functions. Dictionary fields are another possible source of ambiguous memory operations. There are more troublesome issues with dictionary fields, which are discussed in Section 10.7.2. Emulating ambiguous memory access is currently not supported by the PyStream compiler.

10.6 Algorithm Summary for Emulating References

The PyStream compiler has an algorithm to select how every reference in a Python shader program is implemented in GLSL. The algorithm takes into account the types of values that can be held by each reference and what volatile values are shared between references. The values considered by the algorithm include both objects with intrinsic types and fields from objects without intrinsic types. These values are found by examining the pointer analysis information associated with each local variable in a program. All memory operations in a program are examined to determine what fields contained in an object may be accessed together and what fields are modified. Objects with non-intrinsic types are inlined into each GLSL reference data structure, but the semantics of dereferencing fields is preserved. The exact structure of each reference depends on how the values held by the reference are used throughout the shader program.

The following is a statement of conditions (rules) incident to the algorithm which are necessary for the algorithm to emulate reference semantics. The PyStream compiler implements reference data structures in GLSL following these rules:

1. A GLSL reference data structure has a typeID if the reference can hold more than one type of object.

2. A GLSL reference data structure has a separate field for every type of value it can contain. For example, a reference data structure could have one field for holding `vec3` objects and one for holding the `diffuseColor` field of `Material` objects.

3. A value held in a reference data structure is marked volatile if the value is ever modified while held by more than one reference.

4. A nonvolatile value is embedded in a reference data structure and is copied with the reference.

5. All volatile values are stored in pools and reference data structures contain indexes into such pools instead of directly containing volatile values. Only the index into the pool is copied when a reference is copied.

6. A specific volatile value is stored in only one pool. There is only ever one copy of a volatile value, and it remains in the same pool for its entire lifetime.

7. If a single memory operation can access fields from different objects, then all these fields are implemented as having the same name inside of any reference data structure that can hold any of these objects. This allows memory operations to be implemented by reading a field from a reference data structure without taking into account the specific object held by the reference.

8. If a single memory operation can ever access multiple fields on a single object, then these fields are held in an array in the reference data structure.

This allows memory operations to select between the fields without using control flow. For instance, if a loop iterates over the contents of a tuple, then all of the array fields in the tuple are stored in an array in the GLSL reference.

9. If a reference data structure holds different volatile values in the same field — for example, if it can hold two volatile `vec3` objects — then all of the values must be stored in the same pool. This allows a volatile value to be retrieved by indexing into a single, known pool.

The algorithm emulating references is applied to all references in a Python shader. In the example shaders, the most common category of references dealt with by this algorithm point to one or more non-volatile objects sharing a single intrinsic type. The implementation of this particular category is efficient because the data structure implementing the reference will only contain a single intrinsic value which will be copied between the emulated references.

10.7 Boundaries to Emulating Python Reference Semantics

The semantics of Python can be emulated in GLSL, but there are scenarios that cannot be dealt with using the algorithm described in this chapter. In general, the PyStream compiler can translate Python programs into GLSL if the programs are mostly numeric and do not create volatile data structures inside of loops. This describes most all shader programs. There are scenarios where the PyStream compiler cannot emulate Python reference semantics in GLSL because the compiler cannot bound the size of a data structure. It is also troublesome to emulate certain built-in Python data structures, such as strings and dictionaries, because GLSL offers little support for anything but numeric computation. Python shaders can also be further constrained by the resource limits of GLSL. These scenarios remain as an issue only if they continue to exist after compilation. Transformations applied by the compiler may eliminate all of these scenarios as the compiler simplifies the program, making it unnecessary to translate these scenarios into GLSL.

```
def unboundedLoop():
    list = []
    while random():
        list.append(0)
    return list
```

Figure 10.1: Unbounded Data Structure Example

10.7.1 Bounding Data Structure Size

The PyStream compiler must be able to bound the size of a data structure to translate the data structure into GLSL. Because GLSL does not support references, memory must be pessimistically preallocated for every possible data structure. Two features in Python that may create data structures of unbounded size are recursive calls and loops. Recursive calls are disallowed in the PyStream language because they cannot be implemented in GLSL. Loops are allowed by the PyStream language and they can create data structures of unbounded size in two cases: when they modify recursive data structures and when they modify container objects, such as lists and dictionaries. Recursive data structures are also disallowed in the PyStream language, which leaves container objects that are manipulated in a loop as the only data structure with a potentially unbounded size. In some cases it may be possible to bound the number of iterations in a loop, and thereby bound the size of the container object, but there may be cases in which is it impossible to bound the number of loop iterations. Figure 10.1 shows a Python function that cannot be translated into GLSL because the size of the data structure cannot be bounded.

The PyStream compiler must be able to bound the number of volatile values created during the execution of a shader. The bound must be on the number of volatile values created, not on the number that may exist at any one time, because the algorithm for emulating references currently has no way to deallocate values once they are no longer used. It is hard to create an unbounded number of volatile values in a Python shader. To do so, a Python shader must create and mutate objects inside of an unbounded loop while maintaining multiple references to the created objects and do it in such a way that frustrates load/store elimination from eliminating the memory operations that result in the objects being volatile. This situation is difficult to contrive without using recursive data structures or container objects of unbounded size. Figure 10.2 shows an example where this scenario is

```
def unboundedVolatile():
    old = vec2(1.0, 0.0)

    while random():
        current = vec2(1.0, 0.0)
        old.x = old.x + 1
        current.x = current.x+old.x
        old = current

    return old.x
```

Figure 10.2: Unbounded Volatile Value Example

achieved using loop-carried variables. Because PyStream's pointer analysis indicates that `current` and `old` can point to the same object, the objects are not considered to be uniquely held, and are marked volatile. In addition, the interleaving of field modifications (both in the allocation and in the stores) prevents load elimination from being performed because the pointer analysis cannot distinguish between objects created in different iterations of a loop. The research results suggest that the use of both aggressive pointer analysis and aggressive load elimination could make this scenario extremely rare.

10.7.2 Emulation of Built-in Python Data Structures

Python has certain built-in data structures not easily emulated in GLSL. These data structures are provided by Python to break some general data structure rules that apply to user-defined classes. These built-ins include variable-sized container objects (lists, dictionaries, and sets), strings, and long integers. These built-in data structures did not affect the example shaders because they were compiled out of the shaders and because the example shaders were designed to operate without them.

Built-in data structures not easily emulated in GLSL include:

Lists. Lists are variable-sized container objects whose size must be bounded to be translated into GLSL. In general, fixed-size tuples can be used as a substitute for lists.

Dictionaries. Similar to lists, their sized must be bounded to be translated into GLSL.

```
def uglyDictionary():
    d = {}
    i = 0
    while i < 10:
        d[i] = i
        i += 1

    return d[5]
```

Figure 10.3: Unemulatable Dictionary Keys Example

Sets. Sets are similar to dictionaries and the same issues apply.

Strings. Strings are effectively immutable container objects with character data, but string manipulation can produce new strings with hard-to-bound lengths.

Long Integers. Long integers are effectively immutable container objects with integer data with similar problems to strings.

In some special cases, a compiler could emulate a built-in data structure by emulating the effect of using the structure and not its actual contents. For example, preexisting constant string objects could each be assigned a unique identifier, and equality operations on these objects could be emulated by comparing the unique identifiers. This idea could be extended to support other types of operations on preexisting constant values.

Python dictionaries cannot be easily emulated on the GPU. Dictionaries are widely used in Python to define the structure of a program, but using them does not require emulating them on the GPU. The PyStream compiler optimizes away most of these dictionary uses while maintaining the semantics of a Python program. A dictionary only needs to be translated into GLSL if the dictionary is used directly in a computation and load/store elimination cannot transform data flow through the dictionary into local data flow. In the general case, there is no clear way to emulate the semantics of mapping arbitrary keys onto values. Figure 10.3 shows an example where it would be hard to determine how to map keys to values, even though the keys are all integers. If keys used to look up values in a dictionary are a known set of preexisting value objects, then each of the keys can be assigned a unique identifier and the dictionary can be implemented as an array on the GPU.

10.7.3 GLSL Resource Limits

Even if a compiler can bound the size of data structures in a shader program, there is no assurance that the final bounded size will be small. The compiler must make pessimistic assumptions about data structure size to allow the generated shaders to support the worst case. Both GLSL and GPUs in general place a hard limit on the resources available to any one shader. On modern GPUs this limit is comfortably large, but pessimistic assumptions about data structure size during translation can inadvertently push a shader toward the limits of a GPU.

The `list` data type is a feature of Python that can cause a mismatch with the resource limits of GLSL. Even if the maximum size of a list is bounded, the compiler must allocate space for the potentially longest version of the list holding the largest objects that the list can hold. A list would be useful, for example, in any shader that calculates the contribution of multiple lights to the color of a surface in a single pass. Such a list could hold an arbitrary collection of polymorphic lights. However, each light in the list can use one or more texture samplers when computing its own effect on a surface. GLSL imposes a limit on the number of texture samplers that can be declared in a shader program and a list of polymorphic lights would quickly exceed GLSL's limit. The limit is on the number of texture samplers *declared* without regard to how many are actually used. Implementing a polymorphic light in GLSL requires the implementation to declare enough texture samples to match the maximum number of texture samples used by any type of light. This potentially causes an explosive growth in the number of texture samplers declared in a list of polymorphic lights.

Resource limits of GLSL were not explored in this research. As previously mentioned, the PyStream platform design did not support the list data type, which made resource limits not an issue. The Python example shaders render each light using a separate shader. This design kept the shaders well within the resource limits of GLSL. Automatic shader partitioning [39] could be later used to partition PyStream's shaders in those cases where the shaders exceed GLSL's resource limits. Further research may be necessary to determine how pools of volatile values may interact with shader partitioning.

10.7.4 Performance Considerations

In general, any Python shader that can be translated into GLSL should run fast. GLSL is designed to ensure performance by eliminating language features that would slow it down. The main question is not whether a shader will perform well once translated into GLSL, but whether a shader can be translated into GLSL at all. Python offers an abundance of ways to write a shader program. Some translated Python features may reduce performance, and should this happen, other features from Python can be selected instead. Among the potential performance considerations for Python shaders are:

- Objects should not be modified after they are created, particularly if the object is held by multiple references. Volatility does prevent a value from being copied in GLSL, can require indirection, and can result in additional memory usage. Allocating new objects should be favored over modifying existing objects in a Python shader.

- Ambiguous memory operations (operations that can access more than one field) should be minimized, as implementing them in GLSL requires indirection.

- Ambiguous control flow should be minimized. Ambiguous control flow inserts branches in the generated GLSL shader program. Poorly used polymorphic method calls in a Python shader can result in a large number of branches in the GLSL shader. This issue may not be immediately apparent when inspecting a Python shader program, but it is implicit in Python's method call semantics.

10.8 Chapter Summary

All steps in the PyStream compiler combined to make it easier than expected to translate Python reference semantics into GLSL. The last step in the pipeline requires the PyStream compiler to implement the semantics of Python on an architecture that does not support some features Python expects to have available. There is a significant semantic gap between how Python and GLSL each deal with memory and references to memory. PyStream bridges this gap to allow Python shaders to run on the GPU, which is an architecture on which Python was not designed to run. This chapter discussed how to translate into GLSL the broadest possible range of Python's reference semantics, even though the example shaders did not require it. The PyStream compiler was capable of simplifying the Python shaders and eliminating cases that would have been hard to translate. These hard-to-translate cases did not exist by the last step in the PyStream compiler pipeline.

Chapter 11

PyStream's Implementation Outcomes

A real-time rendering system was created, written entirely in Python. The shaders in the rendering system were compiled by the PyStream compiler to run on the GPU. Existing real-time rendering algorithms were used as a means to test the effectiveness and efficiency of the PyStream compiler design and to test the new algorithms developed for the compiler to run Python shaders on the GPU. Existing shader algorithms were reimplemented on the PyStream platform because novel *compiler* algorithms, not novel rendering algorithms, were the focus of this research. Compiler algorithms developed for the PyStream compiler demonstrate that Python, in the form of Python shaders, can run on the GPU.

11.1 Example Python Shaders Used to Test PyStream

A shader is a specialized function that is used to process a specific type of data (for example, vertices, fragments, and others) while producing an image in a real-time rendering system. A shader program contains multiple shaders working together to produce a desired effect while rendering an image. A real-time rendering system contains multiple shader programs and combines the effects of each program to produce the desired image. PyStream offers a platform for creating shader programs that run on the GPU. These programs can combine to build a real-time rendering system.

All example shader programs created for this research were designed to be combined into a single real-time rendering system. Figure 11.1 shows an image produced by this rendering system. Existing Python shaders for real-time rendering were not found in the literature, so new example Python shaders were created which were patterned after the rendering techniques used by state-of-the-art games [8, 40]. The rendering system created for this research uses a deferred rendering approach to produce images. *Deferred rendering* is a technique that di-

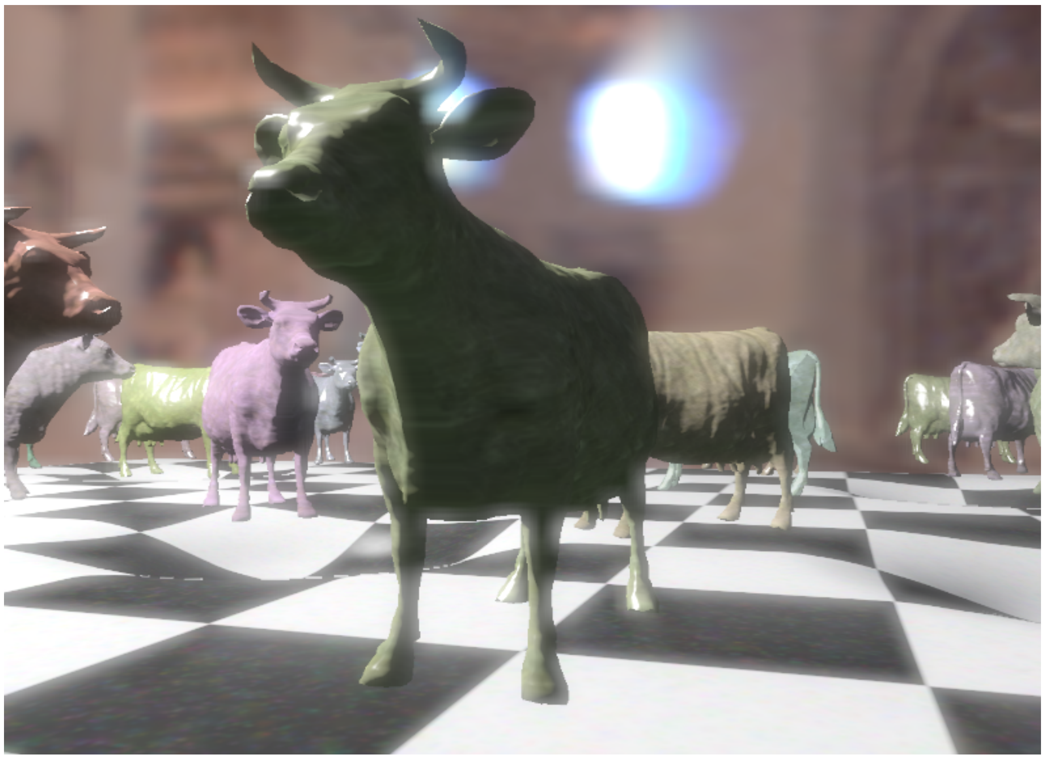


Figure 11.1: An Image Produced by the Example Rendering System

vides the calculation of geometric properties and the calculation of lighting into separate shader programs. This decouples the run time cost of the lighting from that of the geometry. The rendering system is also built to support high dynamic range rendering (HDR). In HDR rendering, colors can exceed the dynamic range of the output device. This is important for rendering realistic images since the dynamic ranges seen in nature can be five orders of magnitude, whereas modern output devices can only reproduce two to three orders of magnitude.

There are eight individual shaders in the example rendering system created for this research. Each shader corresponds to types of shaders used in other real-time rendering systems, although there are no established names for these individual shaders. These shaders are described as follows:

Material. The material shader generates the geometric properties of each thing rendered. Figure 11.2 shows the material shader program. For each pixel covered by the thing rendered, the material shader generates the position, the surface normal, and properties related to how the surface reflects light. All of these properties are stored and later read by other shaders.

Sky Box. The sky box shader draws the background behind everything else.

```
class Material(object):
    __slots__ = ['objectToWorld', 'light', 'material', 'perturb', 'env']

    def createTSBasis(self, trans, normal, tangent, bitangent):
        newnormal  = transformNormal(trans, normal)
        newtangent = transformNormal(trans, tangent)
        newbitangent  = transformNormal(trans, bitangent)
        return TangentSpaceBasis(newnormal, newtangent, newbitangent)

    def shadeVertex(self, context, pos, normal, tangent, bitangent, texCoord):
        trans      = self.env.worldToCamera*self.objectToWorld
        newpos     = trans*pos

        # Create an orthonormal basis that translates from
        # texture space to camera space
        tsbasis = self.createTSBasis(trans, normal, tangent, bitangent)

        # Pass the screen-space position to the rasterizer
        context.position = self.env.projection*newpos

        return newpos.xyz, tsbasis, texCoord

    def shadeFragment(self, context, pos, tsbasis, texCoord):
        e = -pos.normalize()

        # Perturb the texture coordinate
        texCoord = self.perturb.perturb(tsbasis, texCoord, e)

        # Get the normal at the peturbed texture coordinate
        normal   = self.perturb.normal(tsbasis, texCoord)

        # Create a structure holding properties of the surface
        # at this fragment
        surface = self.material.surface(pos, normal, e, texCoord)

        # Reflect the sky box off the model
        envSpecular = self.env.specularColor(-e.reflect(normal))
        surface.specularLight += envSpecular*fresnel(normal, e)

        # Calculate the color or the surface
        surfaceColor = surface.litColor()
        mainColor = self.env.processSurfaceColor(surfaceColor, surface.p)

        # Write the outputs into the material buffer
        context.colors = packGBuffer(mainColor,
                surface.diffuseColor,
                surface.specularColor.r, self.material.shiny,
                pos, normal)
```

Figure 11.2: Material Shader Program

```
class SkyBox(object):
    __slots__ = ['env']

    def shadeVertex(self, context, pos):
        newpos     = self.env.worldToCamera*pos
        context.position = self.env.projection*newpos
        # Use the offset from the origin as a texture coordinate
        return pos.xyz,

    def shadeFragment(self, context, texCoord):
        emissive = self.env.envColor(texCoord)
        # The sky box does not correspond to real "geometry"
        # so most of the data written into the GBuffer is bogus.
        context.colors = packDistantGBuffer(emissive)
```

Figure 11.3: Sky Box Shader Program

```
class LightPass(FullScreenEffect):
    __slots__ = ['gbuffer', 'light', 'env']

    def process(self, texCoord):
        g = self.gbuffer.sample(texCoord)
        e = -g.position.normalize()

        # Calculate the direction to the light and the
        # light's color at this point in the image
        l, color = self.light.lightInfo(g.position, self.env.worldToCamera)

        # Calculate how the light reflects off the surface
        diffuseLighting = g.diffuse*nldot(g.normal, l)
        specularLighting = blinnPhong(g.normal, l, e, g.shiny)*g.specular
        lit = (diffuseLighting+specularLighting)*color
        return vec4(lit, 1.0)
```

Figure 11.4: Lighting Shader Program

Figure 11.3 shows the sky box shader program. The background is referred to as a sky box because the background is modeled as an infinitely large textured cube containing everything else.

Lighting. The lighting shader is used to draw each light in the scene. Figure 11.4 shows the lighting shader program. The lighting shader reads the information produced by the material shader to calculate how a light interacts with the surface for every pixel covered by the light.

Screen-space ambient occlusion (SSAO). The SSAO [8, 40] shader calculates how occluded the surface is at each pixel in the image. For example, the bottom of a crease is only visible from directly above the crease and it is occluded if viewed from other directions. The overall occlusion of a point on a surface is used to modulate the amount of ambient light that reaches that point. In other words, creases are rendered darker than flat surfaces. Figure 11.5 shows the SSAO shader program.

```
class SSAO(FullScreenEffect):
    __slots__ = ['gbuffer']
    __fieldtypes__ = {'gbuffer':GBuffer}

    def sample(self, texCoord, position, normal, offset):
        # Flip the offset to be in the normal's hemisphere.
        facing = normal.dot(offset)
        if facing < 0.0:
            offset -= normal*facing*2.0

        # Sample at the screen space position
        g = self.gbuffer.sample(texCoord+offset.xy/512.0)

        diff = g.position-position
        dist = diff.length()

        # Make the occlusion fall off with distance.
        weight = max(1.0-dist*dist, 0.0)
        amt = 1.0

        # Calculate the occlusion with a bias to avoid artifacts
        if dist > 0.001:
            dir = diff.normalize()
            amt = clamp(1.01-dir.dot(normal)*1.01*weight, 0.0, 1.0)

        return amt

    def process(self, texCoord):
        g = self.gbuffer.sample(texCoord)
        p = g.position
        n = g.normal

        # Average a number of occlusion samples
        ssao  = self.sample(texCoord, p, n,
            vec3(-0.515199, 0.641665, 0.568187)*4.0)
        ssao += self.sample(texCoord, p, n,
            vec3(0.627079, 0.543492, 0.558022)*4.0)
        ssao += self.sample(texCoord, p, n,
            vec3(-0.530851, 0.609046, 0.589288)*4.0)
        ssao += self.sample(texCoord, p, n,
            vec3(0.785924, -0.551296, 0.279993)*4.0)
        ssao += self.sample(texCoord, p, n,
            vec3(-0.642479, 0.591967, 0.486616)*4.0)
        ssao += self.sample(texCoord, p, n,
            vec3(-0.731236, 0.672430, -0.114596)*4.0)
        ssao += self.sample(texCoord, p, n,
            vec3(0.535317, 0.487092, -0.690056)*4.0)
        ssao += self.sample(texCoord, p, n,
            vec3(-0.623743, 0.199685, -0.755692)*4.0)

        ssao /= 8.0

        return vec4(ssao)
```

Figure 11.5: SSAO Shader Program

```
class DirectionalBilateralBlur(FullScreenEffect):
    __slots__ = ['gbuffer', 'source', 'offset', 'samples', 'falloff', 'similarity']

    def rawSample(self, texCoord):
        g = self.gbuffer.sample(texCoord)
        color = self.source.texture(texCoord)
        return g.position, color

    def sample(self, texCoord, offset, position):
        sposition, color = self.rawSample(texCoord)

        distance = (position-sposition).length()

        diff = offset*self.falloff + distance*self.similarity

        weight  = math.exp(-diff)

        return color*weight, weight

    def process(self, texCoord):
        position, color = self.rawSample(texCoord)
        weight = 1.0

        offset = 1.0
        while offset < self.samples:
            scaledOffset = self.offset*offset
            c, w = self.sample(texCoord+scaledOffset, offset, position)
            color  += c
            weight += w

            c, w = self.sample(texCoord-scaledOffset, offset, position)
            color  += c
            weight += w

            offset += 1.0

        return color/weight
```

Figure 11.6: Directional Bilateral Blur Shader Program

Directional Bilateral Blur. The directional bilateral blur [41, 42] shader is used to smooth the output of the SSAO shader. Figure 11.6 shows the directional bilateral blur shader program. Blurring can be used to achieve good effects in real-time rendering. The SSAO shader calculates ambient occlusion by sampling occlusion in a number of directions and averaging the result. For efficiency purposes, the SSAO shader calculates only a limited number of samples, so the result contains sampling noise and the directional bilateral blur filter is used to smooth out this noise. A bilateral blur is different than other types of blurs because it takes into account the underlying geometry of the surface and only blurs values between similar points on the surface. Values are not blurred across geometric edges in an image. The blur filter is referred to as "directional" because it only performs the blur in one direction at a time. To blur an image in two dimensions the shader needs to be applied twice in two separate directions. Applying a directional blur

```
class AmbientPass(FullScreenEffect):
    __slots__ = ['gbuffer', 'env', 'ao']

    def process(self, texCoord):
        g = self.gbuffer.sample(texCoord)
        ao = self.ao.texture(texCoord).xyz
        ambientLight = self.env.ambientColor(g.normal)*ao
        return vec4(g.diffuse*ambientLight, 1.0)
```

Figure 11.7: Ambient Lighting Shader Program

```
class DirectionalBlur(FullScreenEffect):
    __slots__ = ['colorBuffer', 'offset']

    def process(self, texCoord):
        color = self.colorBuffer.texture(texCoord)
        color += self.colorBuffer.texture(texCoord+self.offset)*0.825
        color += self.colorBuffer.texture(texCoord-self.offset)*0.825
        color += self.colorBuffer.texture(texCoord+self.offset*2.0)*0.384
        color += self.colorBuffer.texture(texCoord-self.offset*2.0)*0.384

        weight = (1.0+0.825*2.0+0.384*2.0)

        return color/weight
```

Figure 11.8: Directional Blur Shader Program

twice is usually more efficient than applying a two-dimensional blur.

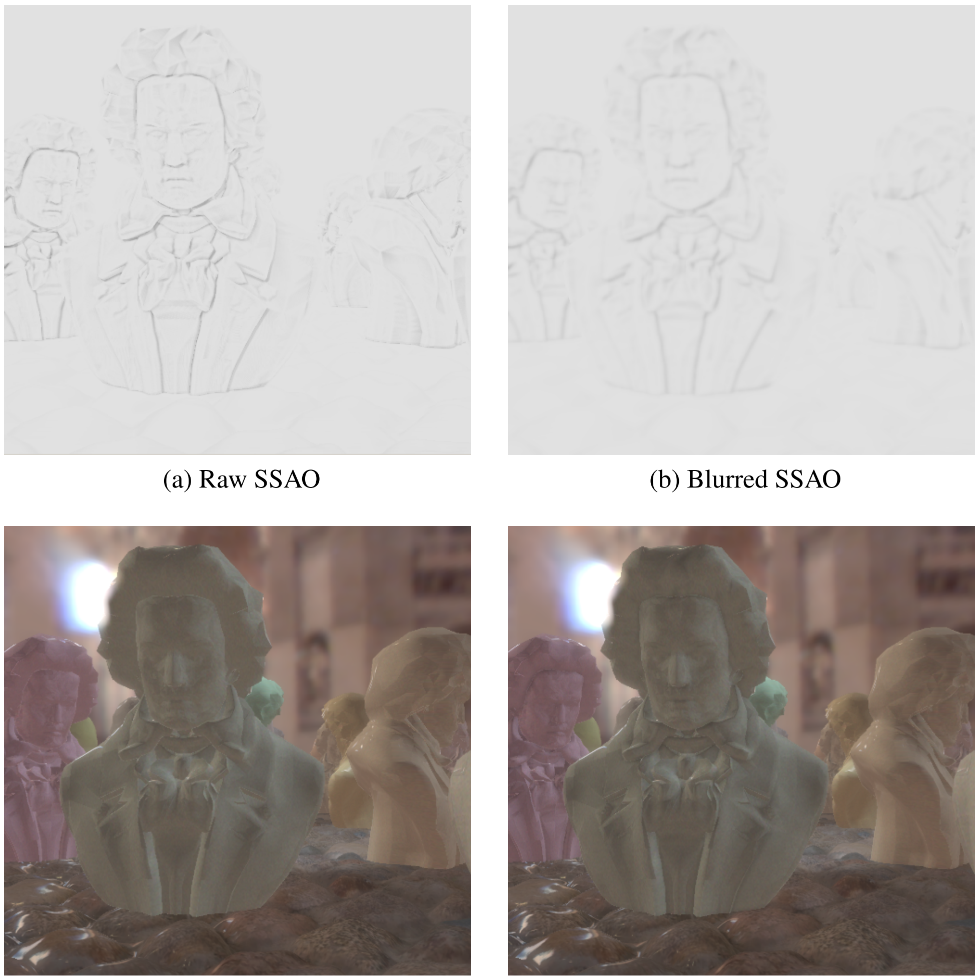
Ambient Lighting. The ambient lighting shader computes how colors radiating from the sky box light the scene. Figure 11.7 shows the ambient lighting shader program. The ambient lighting shader reads the information produced by the material shader to calculate how every pixel on the screen is lit. The output of the SSAO shader is used as an input to the ambient lighting shader to modulate the lighting.

Directional Blur. The directional blur shader is similar to the directional bilateral blur shader, but it does not take into account the underlying geometry of an image. Figure 11.8 shows the directional blur shader program. This shader is used to "bloom" bright spots in an image. Blooming the bright spots in an image simulates how the human optical system reacts to high-dynamic range lighting.

Post-Processing. The post-processing shader computes the final color for each pixel. Figure 11.9 shows the post-processing shader program. This shader first combines a blurred copy of the image with an unblurred copy to simulate bloom. Then it computes a radial blur to simulate "streaking" effects in the bloom. As a last step, it performs *tone mapping* to increase the effective dynamic range of the displayed image.

```
class PostProcess(FullScreenEffect):
    __slots__ = ['colorBuffer', 'blurBuffer', 'samples', 'scaleFactor',
        'falloff', 'bias', 'blurAmount', 'radialAmount', 'env']

    def computeRadial(self, texCoord):
        color  = vec4(0.0)
        scale  = 1.0
        amount = 1.0
        weight = 0.0

        i = 0
        while i < self.samples:
            # Shift to the center
            uv = (texCoord-0.5)*scale+0.5
            scale *= self.scaleFactor

            # Weighted sample
            color += self.blurBuffer.texture(uv, self.bias)*amount
            weight += amount
            amount *= self.falloff

            i += 1

        return color/weight

    def process(self, texCoord):
        # Blend the original image and the bloom
        original = self.colorBuffer.texture(texCoord)
        blured   = self.blurBuffer.texture(texCoord)
        color = original.mix(blured, self.blurAmount)

        # Add radial streaks to the bloom
        color += self.computeRadial(texCoord)*self.radialAmount

        return vec4(self.env.processOutputColor(color.xyz), 1.0)
```

Figure 11.9: Post-Processing Shader Program



(a) Raw SSAO (b) Blurred SSAO

(c) Without SSAO (d) With SSAO

Figure 11.10: Example of SSAO

SSAO is an important effect in the example rendering system, and it is produced in three steps. Figure 11.10a shows the output of the SSAO shader. Figure 11.10b shows the effect of applying a bilateral blur to the SSAO output. Figures11.10c and 11.10d show the final image, with unmodulated ambient lighting and with SSAO. The effect is subtle, but it helps define the shape of the objects in the scene.

11.1.1 Use of Python's Abstractions in the Example Shaders

The PyStream platform allows abstraction mechanisms from the Python language to be used when writing Python shader programs. Other shader languages in

```
class Environment(object):
    __slots__ = ['worldToCamera', 'projection',
        'cameraToEnvironment', 'envMap', 'ambientMap',
        'fog', 'exposure']

    def processSurfaceColor(self, color, p):
        return self.fog.apply(color, p)

    def processOutputColor(self, color):
        # Scale, tonemap, and apply gamma correction
        return rgb2srgb(tonemap(color*self.exposure))

    # Called on the CPU
    def clearColor(self):
        return self.processOutputColor(self.fog.color)

    def ambientColor(self, n):
        edir = transformNormal(self.cameraToEnvironment, n)
        return self.ambientMap.texture(edir).xyz

    def specularColor(self, dir):
        edir = transformNormal(self.cameraToEnvironment, dir)
        return self.envColor(edir)

    def envColor(self, dir):
        return self.envMap.texture(dir).xyz
```

Figure 11.11: Environment Class

common use do not provide abstractions to the degree available in PyStream. As discussed in Chapter 2, these other shader languages call for the use of custom tools to overcome the deficiencies in the shader languages. PyStream does not need these other tools because it has Python's abstractions.

Using abstractions contained in Python improves the structure of a shader program and makes the shader program easier to modify throughout its lifetime. One abstraction used throughout the examples is the concept of an environment, encapsulated in a single, shared object. Figure 11.11 shows the class implementing the environment. The environment contains data that is shared between shaders, for example, the environment contains the transformation between the world and the camera. This transformation is used by the material, sky box, lighting, and ambient lighting shaders. This transformation is consistent across all invocations of each of these shaders. In the rendering system, a single environment object is created and shared between all shader program instances. Updates to the environment object are therefore seen by all shaders. The environment object takes advantage of two features that are offered by Python and not by GLSL: shared values can be held by reference, and shader functionality can be defined by composing data structures. In existing shader languages, sharing values would be performed with specially written glue code. Shader composition would be done

```
class GBuffer(object):
    __slots__ = ['color', 'surface', 'position', 'normal']

    # Allocate the buffers when the rendering system
    # is initialized.
    def allocate(self, fbo, width, height):
        self.color    = fbo.createTextureTarget(width, height)
        self.surface  = fbo.createTextureTarget(width, height)
        self.position = fbo.createTextureTarget(width, height)
        self.normal   = fbo.createTextureTarget(width, height)

    # Setup the buffers as rendering targets for
    # the material shader.
    def bindForWriting(self, fbo):
        fbo.bind()
        fbo.bindColorAttachment(0, self.color.textureData)
        fbo.bindColorAttachment(1, self.surface.textureData)
        fbo.bindColorAttachment(2, self.position.textureData)
        fbo.bindColorAttachment(3, self.normal.textureData)
        fbo.drawBuffers([0, 1, 2, 3])

    # Sample the buffers inside of a shader.
    # Dead code elimination will eliminate unused portions
    # of the sample, so just read everything.
    def sample(self, coord):
        surface  = self.surface.texture(coord)
        position = self.position.texture(coord)
        normal   = self.normal.texture(coord)

        g = GSample()
        g.diffuse  = surface.xyz
        g.specular = surface.w
        g.shiny    = normal.w
        g.position = position.xyz
        g.normal   = normal.xyz.normalize()

        return g


class GSample(object):
    __slots__ = ['diffuse', 'specular', 'shiny', 'position', 'normal']
```

Figure 11.12: GBuffer Class

with custom code generation.

A similar object shared between shaders encapsulates the buffers that hold information generated by the material shader. Figure 11.12 shows the class implementing the material buffer. Unlike the environment object, the material buffer object is used both inside shaders and by the rendering system. The material buffer object is used inside shaders when they need to read values from the material buffer. The material buffer object is used by the rendering system to create the underlying material buffers when the rendering system is initialized, and to configure the rendering pipeline so the result of the material shaders gets written into the buffers. The material buffer is interesting because the abstraction is shared by the rendering system and the shaders. Without PyStream's unified code base, this

```
class FullScreenEffect(object):
    __slots__ = []

    def shadeVertex(self, context, pos, texCoord):
        context.position = pos
        return texCoord,

    def shadeFragment(self, context, texCoord):
        context.colors = (self.process(texCoord),)
```

Figure 11.13: Fullscreen Effect Base Class

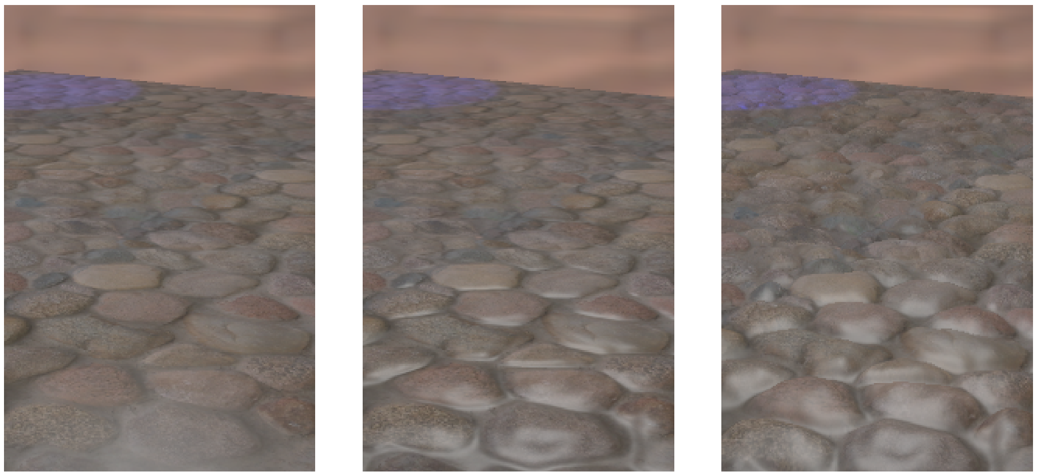

(a) Pass Through (b) Normal Map (c) Parallax Occlusion

Figure 11.14: Surface Perturbation Algorithms

would require code duplication and glue code.

All of the shaders, except for the material and sky box shaders, inherit from the `FullScreenEffect` base class. Figure 11.13 shows this class that specializes the basic PyStream shader into one that processes all the pixels in an input image. This eliminates the need to write boilerplate code when creating a shader that processes an entire image. Some existing rendering systems support this abstraction using a custom code generator. PyStream supports this abstraction by implementing it as class inheritance.

Object-oriented polymorphism is used to parametrize the functionality of two of the shaders. For instance, a material shader contains an object that can affect how the surface normal is computed. Figure 11.14 shows the results produced by the three classes that can be used parametrize the material shader. One class passes through the surface normal of the underlying geometry, and another perturbs the normal based on normal data stored in a texture map. The third class performs parallax occlusion mapping [15] which traces rays against a texture map to perturb the surface. Polymorphism is used in the lighting shader to implement different kinds of lights. Currently, a lighting shader can render either a point light or a spot light, depending on the object attached to it.

Early in the research, the rendering system was built as a forward rendering system, not a deferred rendering system as was subsequently used. A forward rendering system combines the functionality of the material and lighting shaders into a single shader. In the course of research, the forward rendering system was refactored into a deferred rendering system to better match the current state-of-the-art in rendering systems. Refactoring the system into a deferred rendering system was straightforward, however, because the existing shaders encapsulated much of their functionality in objects. Refactoring the rendering system only required changing the top-level function of each shader. The rendering system developed for this research strongly suggests that use of Python's abstraction mechanisms can significantly simplify the engineering of a real-time rendering system.

11.2 Performance of the Example Python Shaders

Measuring the overall performance of a rendering system is complicated because performance depends on a number of factors, some of which interact. The choice of what rendering algorithms are used affects performance and it is generally not informative to compare two rendering systems if they do not render images using the same algorithms. For example, one rendering system may focus on image quality at the cost of performance, whereas another may make the opposite choice. Algorithms can also be optimized and tuned at a high level, meaning that the same algorithm can perform differently depending on how it was optimized. Performance is also affected by low-level choices about how a rendering algorithm is implemented on the hardware. Measuring a rendering system's performance is further complicated by the fact that it can vary depending on what is being rendered. For instance, a simple image can be rendered faster than a complex image. In addition, various aspects of a rendering system can bottleneck overall performance depending on what is being rendered. Overall performance can be limited by processing vertices, processing fragments, or even CPU overhead from API calls. The example rendering system used in this research does not have an existing rendering system with which to compare it. The number of real-time rendering systems written in Python is small, and the number of real-time rendering systems with Python shaders is non-existent beyond this research.

The performance of the example rendering system is evaluated indirectly using a number of measures. First, images produced by the rendering system are shown. The speed at which these images are produced is also documented. The images and speed allow the inference that the rendering system indeed runs in real time and produces high-quality images. Second, the performance of each individual shader is compared to the same shader interpreted by a Python interpreter to show the speedup achieved by the PyStream compiler. Third, a manual inspection of the generated GLSL code is performed to determine how efficiently the Python shaders are implemented on the GPU. Fourth, measurements are taken to show that the PyStream compiler can improve shader performance by generating glue code for different versions of an API.

All of the measurements were performed in Windows XP on an AMD Athlon 64 X2 3800+ CPU and on an NVidia 9800 GT GPU. The CPython 2.6.4 interpreter was also used.

11.2.1 Performance of the Overall Rendering System

Figure 11.15a shows the number of frames that can be rendered every second (FPS) as the number of objects drawn increases. These measurements were taken while rendering to a 1024x1024 frame buffer. In general, more than 15 FPS is considered interactive, and 60 FPS provides a high-fidelity illusion of smooth movement. The example rendering system easily provides interactive frame rates, and scales well in terms of the number of objects being drawn.

Figure 11.15b shows the FPS of the example rendering system as the size of the frame buffer increases. The specified resolution is in both dimensions, so doubling the resolution actually quadruples the number of pixels drawn. Deferred rendering systems are bandwidth intensive and it is not surprising that performance tapers off as the resolution is increased. It is apparent that when rendering to a 1024x1024 frame buffer, the performance of the example rendering system is largely bound by the number of pixels drawn.

Overall, the performance of the example rendering system is good. The rendering algorithms themselves have not been aggressively optimized, and it is expected that even better performance could be achieved by hand-tuning the algo-

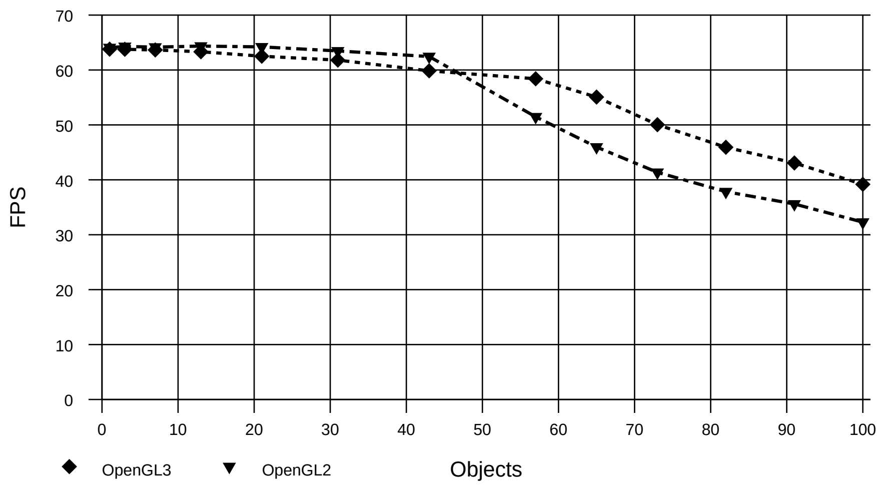
Table 11.1: Shader Performance

	Pixels	**GPU**	**GPU/Pixels**	**CPU**	**CPU/Pixels**	**Improvement**
material	0.77	4.33 ms	5.62 ms	169.8 s	220.5 s	39,211x
skybox	0.27	0.22 ms	0.81 ms	9.6 s	35.5 s	43,568x
ssao	1.00	1.44 ms	1.44 ms	444.9 s	444.9 s	308,958x
bilateral	2.00	2.97 ms	1.49 ms	858.2 s	429.1 s	288,956x
ambient	1.00	0.84 ms	0.84 ms	64.1 s	64.1 s	76,310x
light	5.00	4.75 ms	0.95 ms	635.5 s	127.1 s	133,789x
blur	4.00	2.14 ms	0.54 ms	296.8 s	74.2 s	138,692x
post	0.23	2.20 ms	9.57 ms	101.8 s	442.6 s	46,272x
total	14.27	17.59 ms	1.23 ms	2580.7 s	180.8 s	146,712x

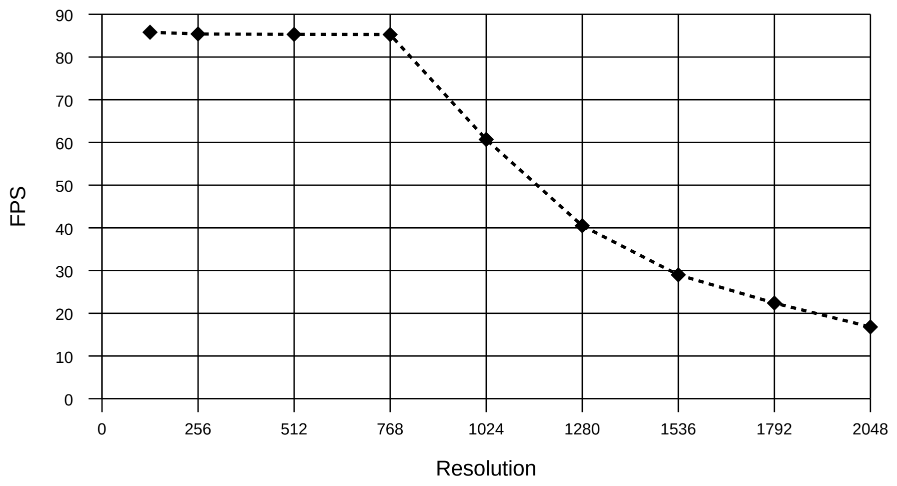
rithms, but that is outside the scope of this research.

11.2.2 API Optimization in PyStream

The PyStream compiler can improve shader performance by changing the API (Section 2.5) for which it generates shader code. It has been previously demonstrated that incorporating high-level information about an API into the compilation process can improve performance for the domain of linear algebra [43]. PyStream can generate glue code and GLSL shader code for both the OpenGL 2 API and the OpenGL 3 API. As it relates to PyStream's code generation, the main difference between OpenGL 2 and OpenGL 3 is that OpenGL 3 requires fewer calls within its glue code. When generating code for OpenGL 2, the example Python shaders are often bound by API call overhead. Figure 11.15a shows that generating code for OpenGL 3 improves the performance of the rendering system by ~20% when large numbers of objects are drawn and the performance of the rendering system is limited by API call overhead. This performance improvement was obtained without changing the Python shaders. PyStream's capability to improve performance without modifying the rendering system demonstrates that a domain-specific compiler for real-time rendering has both performance and engineering advantages.



(a)



(b)

Figure 11.15: Rendering System Performance

11.2.3 Compiled GPU vs. Interpreted CPU Shaders

Table 11.1 shows the total execution time of each shader when rendering a typical scene. The measurements were taken when rendering to a 1024x1024 frame buffer. The “Pixels” column shows the number of pixels drawn by a shader program, divided by the number of pixels in the frame buffer. This shows the number of times a shader program was run per pixel. The “GPU” and “CPU” columns show the time taken by each shader when rendering the image. The “GPU/Pixels” and “CPU/Pixels” columns show the time taken, divided by the number of times the shader was run per pixel. This illustrates the relative performance of each shader. Comparing the total time taken by Python shaders running on the GPU to the same Python shaders interpreted on the CPU shows that PyStream can run shaders on the GPU at least 146,000 times faster than the Python interpreter can on the CPU. This number is a lower bound because no complete rendering system exists to use the Python shaders on the CPU. The performance measurements of a Python shader program interpreted on the CPU are extrapolated from the average performance of its fragment shader and weighted by the number of pixels that would be processed. This estimate is biased in favor of the shader being interpreted on the CPU because it does not factor in time taken by the shader on the CPU to transform vertices, rasterize triangles, sample textures, write to the frame buffer, and similar rendering functions. The measurements of the Python shader programs running on the GPU take into account all costs. In general, the shaders that show the least speedup have few numeric operations and mostly sample textures, a situation that causes the time taken on the CPU to be significantly underestimated.

A speedup of five orders of magnitude may seem astonishing, but it is reasonable if the sources of the speedup are individually accounted for. For example, consider the directional blur shader, which performs 48 floating point operations per pixel. This means that the measured performance of the blur filter is 95 GFLOPS (billions of floating point operations per second). This is 28% of the theoretical maximum achievable on this GPU (Table 11.2). Based on benchmarks for LINPACK, a common CPU benchmark, this CPU can achieve 945 MFLOPS in practice. This meansthat , if only FLOPS are considered, two orders of magnitude speedup can be achieved by moving from the CPU to GPU. In the CPython interpreter running on the same CPU, an optimized loop of add/multiply operations can achieve 11 MFLOPS. This indicates that interpreted Python is roughly 2 or-

Table 11.2: Floating-Point Operations Per Second

	FLOPS
CPython Shader	0.8 M
CPython Benchmark	11.0 M
LINPACK Benchmark	945.0 M
Observed GPU	95.0 G
Theoretical GPU	336.0 G

ders of magnitude slower than code running directly on the CPU. This slowdown includes interpreting byte codes, keeping each floating point value as a separate object, and using double-precision floating-point math instead of single-precision. Taken together, this means that a speedup of roughly four orders of magnitude can be accounted for by changing the underlying architecture from interpreting the shader on the CPU to directly running it on the GPU.

The final order of magnitude can be attributed to removal of abstraction overhead by the PyStream compiler. Many of Python's abstraction mechanisms were used in the example shaders. The example Python shaders were written for simplicity, not for speed. The cost of many of the abstractions used is orthogonal to the cost of interpreting Python. For example, most of the shaders divide their functionality into a number of short methods and functions, increasing the number of calls that must be made. Each shader splits its data across multiple objects, reducing cache coherency. Python's idiom for ad hoc polymorphism was extensively used when reimplementing GLSL functions and types in Python. This means that every class initializer contains extensive control flow. GLSL arithmetic operations also contain control flow to determine the types of the arguments. The PyStream compiler eliminates most memory operations in a shader, eliminating the overhead of both allocating objects and loading and storing fields. An order of magnitude improvement for eliminating these abstractions is more than reasonable. Taking all of these sources of improvement into account, a speedup of five orders of magnitude is not unexpected.

11.2.4 Quality of the Generated Code

Manual inspection of the example shaders shows that the generated code implements the functionality specified in the Python shaders without any abstraction overhead. In many situations, the PyStream compiler generates code that copies

Table 11.3: Performance of PyStream's Compiler Pipeline

	Time (ms)	**% Total**	**% No Clone**
Decompilation	558	1.5%	2.7%
Pointer Analysis	6178	16.1%	29.8%
Method Fusion	278	0.7%	1.3%
Escape Analysis	1883	4.9%	9.1%
Simplify	470	1.2%	2.3%
Function Cloning	17510	45.8%	-
Function Inlining	1683	4.4%	8.1%
Pointer Analysis 2	2596	6.8%	12.5%
Escape Analysis 2	974	2.5%	4.7%
Load/Store Elimination	751	2.0%	3.6%
Tree Transform	4005	10.5%	19.3%
Output Flattening	803	2.1%	3.9%
Field Transform	204	0.5%	1.0%
GLSL Transform	368	1.0%	1.8%

variables when a human being would not. These extra copy operations generated by the PyStream compiler are not a performance overhead for the GLSL shaders. The extra copies are likely eliminated with a straightforward copy-elimination optimization when the GLSL shader is compiled onto the GPU.

11.3 Compilation Speed in PyStream

Table 11.3 shows the time consumed by each step in the PyStream compiler pipeline when compiling all of the example shaders, both in terms of absolute time and in terms of a relative percentage. The entire process takes 38.5 seconds. The two most time-expensive operations in the compiler are the initial pointer analysis and the function cloning.

Function cloning is time-expensive in the PyStream compiler due to an indirectly observable flaw in the implementation. As the size of the example shaders increases, the function cloning transformation slows down by a disproportional amount. This indicates that performance characteristics of the function cloning transformation are unintentionally non-linear. The source of this non-linearity is unknown, but it does not significantly affect the results of this research. In Table 11.3, the column "% No Clone" shows the relative performance of each step if the time taken by cloning is ignored. This provides a clearer picture of the time taken by each step.

The initial pointer analysis is time-expensive, in part due to how it is implemented and in part due to the nature of Python. The selected underlying pointer analysis algorithm used in PyStream is intentionally unsophisticated. Section 7.10.2 indicates that a more sophisticated pointer analysis algorithm can run faster. Features inherent in Python generally make it time-expensive to analyze. Python has abstraction overhead that has both a run-time-cost and slows down analysis. Pointer analysis of Python has to contend with multiple levels of indirection, both for pointer data and in the call graph. When the PyStream compiler reanalyzes the Python shaders after abstraction overhead has been eliminated, the analysis is about twice as fast despite the fact that heap sensitivity is increased during reanalysis. The improved performance during reanalysis indicates that abstraction overhead is time-expensive for pointer analysis. When analyzing Python, inherently expensive analysis algorithms, such as flow-sensitive pointer analysis, should be deferred until after transformations have eliminated as much as possible of Python's abstraction overhead.

Table 11.3 shows that the novel algorithms of the PyStream compiler are efficient. PyStream's novel algorithms, including method fusion, argument normalization, output flattening, field transform, and emulating reference semantics, are all time-inexpensive compared to other operations in the PyStream compiler.

The PyStream compiler operates at a speed usable for research. For a production compiler, this compilation speed could be increased with a number of optimizations. As a general concept, a compiler written in Python is slower than a compiler written in C. Writing the PyStream compiler in Python causes it to run one to two orders of magnitude slower than it would if it were written in C. The design of the PyStream compiler was largely unconcerned with compilation speed, so further optimizations to increase compilation speed can be readily accomplished.

Chapter 12

Future Work

The PyStream platform explores running a high-level language on a restricted high-performance architecture for which the language was not designed. This opens up new areas of future research. There are aspects of the PyStream platform that can be modified, extended and expanded, either to accommodate evolving GPUs or to increase the subset of Python which is translated.

12.1 Integrating the CPU and GPU for Real-Time Rendering

The PyStream platform allows both a CPU and a GPU to be programmed in tandem with a single code base, a feature that invites future research topics. The capacity of the PyStream platform to compile Python can be applied to more than just shaders. In the long-term, the PyStream platform could be used to compile an entire rendering system. This would allow whole-program domain-specific optimizations to be applied by the compiler. Possible topics for future research include:

Debugging Shaders on the CPU. CPUs typically provide a better environment in which to debug programs. Using the PyStream platform, shaders could be debugged on the CPU because a single code base would allow the same code to be run on either processor.

Dynamic Compilation. The PyStream compiler was designed and implemented as a static compiler. An area of future research is to determine what benefits may be realized by using a dynamic compiler to generate code for the GPU.

CPU/GPU Optimizations. Transformations can move code between the CPU and the GPU. The single code base allowed by the PyStream platform makes it easier. What benefits may be obtained by exploiting the ease of moving code between the CPU and the GPU is an area of future research. For instance, certain computations can be lifted out of a shader and onto the CPU. This optimization would reduce the frequency with which the computation is performed.

Global Domain-Specific Optimizations. If an entire rendering system is compiled by the PyStream platform, global domain-specific optimizations can be applied. For instance, an area of research would be how to globally optimize data passed between each shader in a rendering system.

Improving Low-Level Shading Languages. A native shader language with fewer restrictions than GLSL would simplify running Python shaders on the GPU. CUDA was considered for research use because it presents a flexible programming model. GLSL was used instead of CUDA in the research because CUDA does not expose rendering-specific hardware on the GPU. A future area of research is whether a shading language can be created that combines the flexible programming model of CUDA with access to rendering-specific hardware.

12.2 Pointer Analysis for Python

The articulation within the PyStream compiler of how to perform pointer analysis for Python invites future research on pointer analysis for Python. Specific topics for future research include the following:

Supporting More Features of Python Using a Low-level Approach. There are a number of features in Python that are not supported by PyStream's pointer analysis because they were outside of the scope of this research. The PyStream platform can be modified to support these features. Future research may be able to demonstrate that a low-level approach can support all Python features, such as closures, generators, and kargs.

Supporting Exceptions. Supporting exceptions when compiling Python requires further research. Supporting exceptions in a pointer analysis for Python would require that the analysis be flow sensitive, due to how Python deals with undefined attributes. If an undefined attribute is read in Python, an exception is raised at run time. All attributes are initially undefined when an object is allocated. A flow-insensitive analysis would conclude that attributes of a dynamically allocated object can always be undefined and any attempt to read one of these attributes can always result in an exception. If reading an attribute can cause an exception in most cases, very few transformations can be applied to a program because exception semantics of Python would need to be preserved. A flow-sensitive analysis is required to verify that most attributes are defined in a Python program before they are read. Efficient flow-sensitive analysis of Python programs is a topic of research. PyPy assumes that an exception does not occur unless the program contains code needed to handle a specific exception; this approach is inadequate for compiling the full Python language.

Increasing Heap Sensitivity. Another area of future research is how to improve heap sensitivity for pointer analysis of Python. Python stores a lot of information in the heap that is critical for resolving the behavior of a program. PyStream uses extended types (Section 7.8) to target known, critical situations where heap sensitivity is required. Extended types work in specific cases, but they do not improve heap sensitivity in general. An Andersen-style formulation of the pointer analysis algorithm described by Lattner et al. [36] could improve heap sensitivity.

Python Interpreters Written in Python. PyStream's low-level approach to pointer analysis may have synergies with PyPy's implementation of a Python interpreter in Python. This may be a matter worth further research. PyStream's low-level approach to pointer analysis, when fully developed, will result in most of the Python interpreter functions being rewritten in Python for analysis and compilation onto the GPU. It is unknown whether these two different implementations of Python interpreter functions written in Python will converge, because each implementation is being engineered for entirely different purposes.

Generating a Pointer Analysis Algorithm from a Python Interpreter. The low-level approach used in the PyStream compiler reinforces that there is a correspondence between analyzing and interpreting a program. PyPy has demonstrated that its Python interpreter can be translated into a JIT compiler for Python [44]. Using a similar approach, it may be possible to generate a low-level pointer analysis algorithm for Python by transforming a Python interpreter.

12.3 Emulating Reference Semantics

Establishing the Limits of the Translation Process. PyStream demonstrates that Python's semantics can be efficiently translated into GLSL for the example shaders. The limits of this translation process were not tested during this research. Future work may determine what more can be translated efficiently and what cannot. This translation process can likely be adapted to other domains, such as translating a Python program onto an FPGA.

Avoiding Exhaustive Inlining. The PyStream compiler uses exhaustive inlining to simplify translation of Python into GLSL. There are two issues associated with use of exhaustive inlining. Exhaustive inlining has the undesirable side effect of duplicating functions. Exhaustive inlining would fail if recursive function calls were allowed in the PyStream language. A future area of research is to determine whether transformations in the PyStream compiler can be reformulated to eliminate the need for exhaustive inlining.

Chapter 13

Conclusions

The PyStream platform successfully runs Python shaders on a GPU. This is an important step toward understanding how to run high-level languages on restricted high-performance architectures for which the language was not designed. An important feature of the PyStream platform is that it generates and optimizes glue code to connect the CPU and the GPU. This allows the GPU coprocessor to share a single code base with the CPU. This approach can be extended to other languages and coprocessors. For example, it is possible that portions of a program written in the Ruby language could be compiled onto an FPGA using techniques similar to those created for the PyStream platform. Within boundaries, Python can run on a GPU when translated into a very restricted low-level language.

The PyStream platform requires fewer programming decisions than are required in a low-level language, making it easier to use. The PyStream platform has the potential to run faster, because complicated and delicate implementation issues can be shifted onto the compiler. When a high-performance architecture, such as a GPU, is introduced into the discussion, then the burden inherent in low-level programming is increased. This problem is illustrated by the issues confronted when engineering a real-time rendering system, issues which the PyStream platform was designed to address.

The PyStream platform demonstrates that Python can be run on a GPU for the subset of Python used in the example shaders. Previously, none of Python could be run on the GPU. The PyStream platform provides comparable functionality to that available in existing shading languages, but it does even more. The PyStream platform brings object-oriented programming to the restricted architecture of a GPU, including references to objects, methods, and polymorphic types. The PyStream platform also brings a subset of Python's semantics onto a GPU, such as attribute lookup. The PyStream platform demonstrates that some features of Python not supported on a GPU can be used in Python shaders that run on the GPU because these features can be eliminated by a compiler. For instance, all of the example Python shaders implicitly use dictionary objects as a consequence of Python's semantics for global variable lookup. The PyStream compiler eliminated all uses of these dictionary objects in the example shaders. Although emulating dictionary objects may be necessary to fully support Python on the GPU in future extensions of this research, it was not necessary in the example shaders used in the research.

The PyStream platform easily and efficiently runs portions of Python on the GPU. While there may be undiscovered complications when the full Python language is run in a similar manner on a GPU, the PyStream platform demonstrates that running a restricted subset of Python offers greater ease of programming than existing shader languages. The PyStream platform offers a number of engineering advantages as well, because the shaders and the rest of the rendering system are written in the same language. The PyStream platform demonstrates that these engineering advantages happen even when running a subset of Python on the GPU.

13.1 Analyzing Modern High-Level Dynamic Languages

Analyzing Python is an open problem. The PyStream compiler advances the solution in two areas. The PyStream compiler takes a low-level approach to analysis which directly addresses the complexities of implementing an analysis for a modern high-level language like Python. An analysis algorithm should use lower-level abstractions with minimal special cases so that it can deal with a larger portion of the language in a sound manner. The low-level approach further simplifies the analysis algorithm by moving language semantics into analyzed code. Results from using a low-level analysis approach in the PyStream compiler indicate that the subset of Python being compiled can be increased more easily than with the high-level approaches previously used.

The other advance toward analyzing Python, as demonstrated by the PyStream compiler, is linking pointer analysis for Python to the larger body of work on pointer analysis. This is made possible in part by using a low-level approach which increases the similarity between analyzing Python and analyzing languages such as C.

13.2 Abstraction Overhead in High-Level Dynamic Languages

PyStream demonstrates, in a specific case, the feasibility of using high-level languages for high-performance applications. It is not well understood how much of the “abstraction penalty” in high-level languages is inherent in their design and how much is due to failures in their implementation. The research for this dissertation demonstrates that a good portion of the abstraction penalty in a Python shader can be compiled away. A number of novel techniques, described in this dissertation, were created to assist in this process.

13.3 Domain-Specific Compilers

The PyStream platform demonstrates the utility of domain-specific compilation for real-time rendering. The architecture and programming interfaces for GPUs have been constantly changing as GPUs have evolved and their utility increasingly understood. This trend will likely continue. Domain-specific compilation allows the compiler to transparently take advantage of new and changing features. This capacity of the PyStream platform is important for real-time rendering, because it allows performance improvements by using new features without the need to rewrite an existing program.

Appendix A

Research Tangents

The PyStream platform came into being through extensive experimentation with various compiler techniques. As with any experimentation, there were things that were tried that turned out to be less than optimal. Some of these techniques were good, but they did not work in the context of the PyStream compiler.

Experimentation indicated that Python is singularly unsuited for generating code via metaprogramming. The first iteration of the PyStream platform used metaprogramming to generate shaders, similar to Sh [16, 17]. This did not work. Python's lexical convention of using white space to indicate program structure can only be applied to Python code and not to a metaprogram. A metaprogram embedded in Python code is therefore difficult to read and understand if it contains any significant structure. PyGPU [18] avoids this problem by manually interpreting the byte codes of a Python function rather than executing the function directly.

Experimenting with pointer analysis techniques that used binary decision diagrams (BDDs) did not work. Particularly, attempting to formulate a BDD pointer analysis as described by Whaley and Lam [35] had a number of issues including problems bounding the size of a Python program and overall difficulties getting the algorithm to perform well.

A novel BDD-inspired technique was used to maintain the correlation between different sets of pointer information using a sparse tree structure. This did not work either. This approach increased the complexity of the analysis because the analysis algorithm needed to simultaneously traverse all of the trees for all of the values that might be needed to resolve a given operation. The larger idea of keeping values correlated is important, however, since it is what makes CPA work.

Region-based shape analysis with tracked locations [45] is a novel form of shape analysis that seems as if it should be able to be performed efficiently. During development, the PyStream compiler contained an implementation of this analysis algorithm so that PyStream could better deal with shaders which did not have tree shaped inputs and outputs. This did not work either. It was hard to determine how to efficiently implement the algorithm, particularly when the complexities of using CPA contexts were added.

At one point during development, the PyStream compiler converted shader code into a dataflow IR [46], which explicitly represents dependencies between memory operations. This did not work either. It was thought that this would make eliminating memory operations easier, but in fact it made it harder. Because the dependencies were explicitly represented, any re-analysis that refined the compiler's view of memory dependencies (Section 9.4) required rewriting the IR. Similarly, transformations that eliminated memory operations (Chapter 9) needed to rewrite the affected dependencies. Keeping memory dependencies implicit in the IR simplified many of PyStream's transformations.

References

[1] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, 2004, pp. 777–786.

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," in *SIGGRAPH '08: ACM SIGGRAPH 2008 Classes*, 2008, pp. 1–14.

[4] A. Munshi. (2009) The OpenCL specification - version 1.0. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf

[5] A. Lefohn, M. Houston, J. Andersson, U. Assarsson, C. Everitt, K. Fatahalian, T. Foley, J. Hensley, P. Lalonde, and D. Luebke, "Beyond programmable shading (parts i and ii)," in *SIGGRAPH '09: ACM SIGGRAPH 2009 Courses*, 2009, pp. 1–312.

[6] F. Mannuß and A. Hinkenjann, "Interactive shader development using Python scripts," *Brazilian Symposium on Computer Graphics and Image Processing*, 2005.

[7] Blender. [Online]. Available: http://www.blender.org/

[8] M. Mittring, "Finding next gen: CryEngine 2," in *SIGGRAPH '07: ACM SIGGRAPH 2007 Courses*, 2007, pp. 97–121.

[9] NVIDIA FX Composer 2.5 GPU shader authoring environment. [Online]. Available: http://developer.nvidia.com/object/fx_composer_home.html

[10] RenderMonkey toolsuite. [Online]. Available: http://ati.amd.com/developer/rendermonkey/

[11] M. McGuire, G. Stathis, H. Pfister, and S. Krishnamurthi, "Abstract shade trees," in *I3D '06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, 2006, pp. 79–86.

[12] J. Andersson and N. Tatarchuk, “Rendering architecture and real-time procedural shading & texturing techniques,” in *SIGGRAPH ’07: ACM SIGGRAPH 2007 Classes*, 2007.

[13] Unreal development kit. [Online]. Available: http://developer.nvidia.com/object/udk.html

[14] R. L. Cook, “Shade trees,” *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 223–231, 1984.

[15] N. Tatarchuk, “Dynamic parallax occlusion mapping with approximate soft shadows,” in *I3D ’06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, 2006, pp. 63–69.

[16] M. D. McCool, Z. Qin, and T. S. Popa, “Shader metaprogramming,” in *HWWS ’02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2002, pp. 57–68.

[17] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, “Shader algebra,” in *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*, 2004, pp. 787–795.

[18] C. Lejdfors and L. Ohlsson, “Implementing an embedded GPU language by combining translation and generation,” in *SAC ’06: Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006, pp. 1610–1614.

[19] D. Tarditi, S. Puri, and J. Oglesby, “Accelerator: using data parallelism to program GPUs for general-purpose uses,” in *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 325–335.

[20] W. Thies, M. Karczmarek, and S. P. Amarasinghe, “StreamIt: A language for streaming applications,” in *CC ’02: Proceedings of the 11th International Conference on Compiler Construction*, 2002, pp. 179–196.

[21] Scipy. [Online]. Available: http://www.scipy.org/

[22] A. Rigo, “Representation-based just-in-time specialization and the Psyco prototype for Python,” in *PEPM ’04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 2004, pp. 15–26.

[23] G. Ewing. (2010, Apr.) Pyrex: A language for writing Python extension modules. [Online]. Available: http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/

[24] Pypy. [Online]. Available: http://codespeak.net/pypy/dist/pypy/doc/

[25] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis, “RPython: A step towards reconciling dynamically and statically typed OO languages,” in *DLS ’07: Proceedings of the 2007 Symposium on Dynamic Languages*, 2007, pp. 53–64.

[26] M. Salib, “Starkiller: A static type inferencer and compiler for Python,” Master’s thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004. [Online]. Available: http://hdl.handle.net/1721.1/16688

[27] O. Agesen, “Concrete type inference: Delivering object-oriented applications,” Sun Microsystems Laboratories, Mountain View, CA, USA, Tech. Rep. SMLI TR-96-52, 1996.

[28] Shed Skin. [Online]. Available: http://code.google.com/p/shedskin/

[29] O. Agesen and U. Hölzle, “Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages,” in *OOPSLA ’95: Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, 1995, pp. 91–107.

[30] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, “Stratego/XT 0.16: Components for transformation systems,” in *PEPM ’06: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2006, pp. 95–99.

[31] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

[32] V. C. Sreedhar and G. R. Gao, “A linear time algorithm for placing ϕ-nodes,” in *POPL ’95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995, pp. 62–73.

[33] K. D. Cooper, T. J. Harvey, and K. Kennedy, “A simple, fast dominance algorithm,” *Software - Practice and Experience*, vol. 4, pp. 1–10, 2001.

[34] E. Nystrom, “Fulcra pointer analysis framework,” Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 2005. [Online]. Available: http://www.crhc.uiuc.edu/IMPACT/ftp/report/phd-thesis-erik-nystrom.pdf

[35] J. Whaley and M. S. Lam, “Cloning-based context-sensitive pointer alias analysis using binary decision diagrams,” in *PLDI ’04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004, pp. 131–144.

[36] C. Lattner, A. Lenharth, and V. Adve, “Making context-sensitive points-to analysis with heap cloning practical for the real world,” *SIGPLAN Not.*, vol. 42, no. 6, pp. 278–289, 2007.

[37] J. Plevyak and A. A. Chien, “Type directed cloning for object-oriented programs,” in *LCPC ’95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, 1996, pp. 566–580.

[38] S. Ryoo, C. I. Rodrigues, and W.-M. W. Hwu, “Iteration disambiguation for parallelism identification in time-sliced applications,” *The 20th International Workshop on Languages and Compilers for Parallel Computing*, pp. 110–124, 2007.

[39] A. Heirich, “Optimal automatic multi-pass shader partitioning by dynamic programming,” in *HWWS ’05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2005, pp. 91–98.

[40] D. Filion and R. McNaughton, “Effects & techniques,” in *SIGGRAPH ’08: ACM SIGGRAPH 2008 Classes*, 2008, pp. 133–164.

[41] G. Petschnigg, R. Szeliski, M. Agrawala, M. Cohen, H. Hoppe, and K. Toyama, “Digital photography with flash and no-flash image pairs,” in *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*, vol. 23, no. 3, 2004, pp. 664–672.

[42] E. Eisemann and F. Durand, “Flash photography enhancement via intrinsic relighting,” in *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*, 2004, pp. 673–678.

[43] S. Z. Guyer, “Incorporating domain-specific information into the compilation process,” Ph.D. dissertation, The University of Texas at Austin, 2003.

[44] A. Cuni, D. Ancona, and A. Rigo, “Faster than C#: Efficient implementation of dynamic languages on .NET,” in *ICOOOLPS ’09: Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, 2009, pp. 26–33.

[45] B. Hackett and R. Rugina, “Region-based shape analysis with tracked locations,” in *POPL ’05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005, pp. 310–323.

[46] M. Budiu and S. C. Goldstein, “Pegasus: An efficient intermediate representation,” Carnegie Mellon University, Tech. Rep. CMU-CS-02-107, May 2002.