DOI: 10.1111/cgf.14382
High-Performance Graphics 2021
N. Binder and T. Ritschel
(Guest Editors)

Volume 40 (2021), Number 8

Cooperative Profile Guided Optimizations

Mark Stephenson
mstephenson@nvidia.com

Ram Rangan
rrangan@nvidia.com

Stephen W. Keckler
skeckler@nvidia.com

Abstract

Existing feedback-driven optimization frameworks are not suitable for video games, which tend to push the limits of performance of gaming platforms and have real-time constraints that preclude all but the simplest execution profiling. While Profile Guided Optimization (PGO) is a well-established optimization approach, existing PGO techniques are ill-suited for games for a number of reasons, particularly because heavyweight profiling makes interactive applications unresponsive. Adaptive optimization frameworks continually collect metrics that guide code specialization optimizations during program execution but have similarly high overheads. We emulate a system, which we call Cooperative PGO*, in which the gaming platform collects* piecemeal *profiles by sampling in both time and space during actual gameplay across many users; stitches the piecemeal profiles together statistically; and creates policies to guide future gameplay. We introduce a three-level hierarchical profiler that is well-suited to graphics APIs, that commonly operates with no overhead and occasionally introduces an average overhead of less than 0.5% during periods of active profiling. This paper examines the practicality of* Cooperative PGO *using three PGOs as case studies. A PGO that exploits likely zeros is particularly effective, achieving an average speedup of 5%, with a maximum speedup of 15%, over a highly-tuned baseline.*

CCS Concepts

• ***Software and its engineering*** → ***Compilers;*** • ***Computer systems organization*** → ***Cloud computing;***

1. Introduction

As Moore's Law comes to an end, runtime systems and code specialization will play an increasing role in boosting the performance of graphics processing units (GPUs). *Profile-guided* and *adaptive* optimizations are related techniques that can potentially specialize graphics shaders for likely runtime characteristics. Profile-guided optimization (PGO) relies on an offline profiling phase in which a developer uses representative training inputs to discover likely runtime behaviors. Because PGO profiling occurs offline, the compiler can aggressively instrument an application to collect a varied set of run-time behaviors from basic-block execution frequencies to value profiles. While PGO has been shown to be effective and has no runtime overheads, it has four significant shortcomings that have hindered widespread usage, particularly for interactive gaming applications: (1) PGO complicates the compilation process by requiring two compilation and link steps, along with a potentially lengthy offline profiling phase. (2) PGO's generalization success (i.e., how a PGO-compiled application behaves in the wild) depends heavily on the inputs used during the profiling phase. (3) Wholesale profiling of the kind typically used for PGO makes interacting with interactive applications, including games, difficult. (4) Code patches to either a game or the GPU driver, both of which are frequent, can invalidate prior profiles.

Adaptive optimization (AO) does not have PGO's generalization shortcomings because it transparently specializes code *during* program execution and therefore achieves generality by dynamically optimizing for any possible execution profile. However, the overheads associated with profiling and code re-compilation must be minimized for AO to be worthwhile. Expensive profiling and frequent re-compilations can negate the benefits of AO. Furthermore, the tight feedback loop required for adaptive optimizations is subject to oscillation in programs with phases [AFG*05].

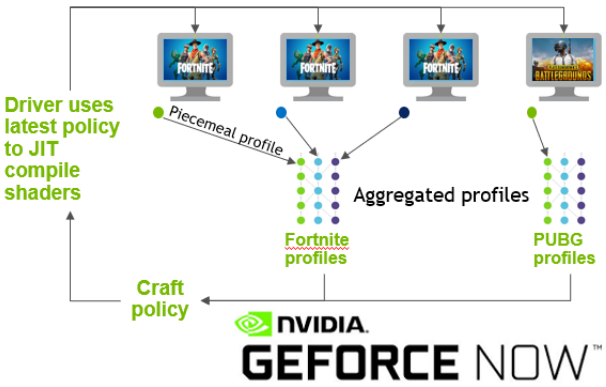


Figure 1: *High-level operation of* Cooperative PGO *in a gaming cloud (e.g. NVIDIA's GeForce NOW). Direct3D clients collect* piecemeal *profiles, which a centralized server aggregates to create actionable, complete profiles of games.*

In this paper we present *Cooperative PGO*, which is a middle ground between PGO and AO. The key insight that enables *Cooperative PGO* is that millions of gamers play the same, finite, and small set of games on the same, finite, and small set of GPUs and consoles. Unlike AO which must aggressively profile and re-compile code, *Cooperative PGO* clients infrequently collect small, *piecemeal* profiles as Figure 1 shows. Therefore, the run-time overhead of *Cooperative PGO* is extremely low. Because *Cooperative PGO* clients collect piecemeal profiles across frames of a gaming application, it can take tens of thousands of profiles to create a complete profile of a program's execution. As we argue in this paper, with a large community of gamers *Cooperative PGO* can quickly cover large profiling spaces. Some tech experts predict that cloud-based gaming such as GeForce NOW, Stadia, and Luna, will increase in popularity [D'A20]. With millions of gamers playing games on consolidated hardware, cloud-based gaming is the ideal target for *Cooperative PGO*.

Cooperative PGO, like AO, works best in the context of a just-in-time compiling virtual machine since it relies on dynamically creating instrumented program variants for collecting profiling data. Graphics languages, such as Direct3D's HLSL [Mic18b] and Vulkan's GLSL [KBR17], are just-in-time compiled and well-suited for *Cooperative PGO*. Like AO, the profiling and re-compilation required by *Cooperative PGO* are seamless and impose no additional burden on developers. Like PGO, *Cooperative PGO* uses previously collected run-time profiles to guide future compiles. However, developers do not have to guess what inputs are representative for their applications with *Cooperative PGO*.

Finally, given that a *Cooperative PGO* system stitches together piecemeal profiles spanning game scenes, levels, and users, the aggregated whole-program profile is an approximation and may even have internal inconsistencies (e.g., the sum of incoming and outgoing execution weights through parts of a control flow graph may not match). We show that *Cooperative PGO* can gainfully leverage approximate profiles to improve gaming performance with three candidate PGOs.

In this paper, we focus on demonstrating the viability of *Cooperative PGO* by emulating it in a modified production driver on a non-distributed research prototype (i.e., a single-node system). Before concluding, in Section 7, we outline the steps required to engineer a fully distributed *Cooperative PGO* system, which we will pursue as future work. Our paper's main contributions are:

- A description of a *Cooperative PGO* system's main components, including a novel hierarchical profiler that is well suited to graphics APIs.
- Characterization of the data collection abilities of *Cooperative PGO* as a function of sampling rate and gaming population size.
- Characterization of the overhead of profiling using a production-quality driver.
- Demonstration of *Cooperative PGO*'s usefulness through the evaluation of three PGOs. A PGO that optimizes for *likely* zero values in the computation improves the performance of several games scenes by an average of 5%, and achieves greater than 10% speedups on varied scenes from two popular games. We also evaluate a proof-of-concept, timing-based PGO that increases performance by nearly 3% on average.

2. Related Work

2.1. Profile-Guided Optimizations

Profile-guided optimizations have a rich history in the literature and many modern compilers support PGO (e.g. [Mic19c, Fre08]). While compilers have limited support for general value profiling [Fre08], control flow profiling to determine the execution weights of various blocks of code is prevalent. Compiler passes such as inlining, register allocation, predication, and loop unrolling then use the execution weights to generate optimized code. For example, a compiler can arrange code based on profile feedback to improve instruction cache performance by packing frequently executed code blocks closer to each other. Execution weight based PGOs have proven effective in a variety of programs, from general purpose programs [CL99] to Web browsers [Chr20]. *Cooperative PGO* crowdsources this proven approach to code optimization.

Value profilers have historically been run offline and have served to guide manual or automatic software optimizations [CFE97, MWD00, WCL17, YYL*20, RSU*20, SR21, ZHMC*20]. Using offline zero-value profilers, researchers have shown that dynamically zero-valued operands can be opportunistically leveraged to perform forward and backward slice code-specialization in gaming applications through the *Zeroploit* transform [RSU*20] and automated in a compiler with *PGZ* [SR21]. We consider *PGZ* as one of the candidate PGOs for *Cooperative PGO* by employing a piecemeal zero-value profiler.

Recently, Leobas and Pereira presented a sampling-based, low-overhead online value profiler to dynamically optimize silent stores in long-running loops [LP20]. Their approach samples a few iterations of long-running loops to decide whether a silent store opportunity exists. This novel technique, which performs well for hot loops in CPU programs, is not suitable for graphics shader programs, which typically do not have loops or at best, have loops with small trip counts. To the best of our knowledge, our piecemeal profiling approach in a *Cooperative PGO* system is the first viable online value profiler for gaming applications.

2.2. Low Overhead and Adaptive Profiling

Multi-versioning dynamically evaluates multiple, differently compiled versions of the same source code and chooses the best version at runtime [DR97, AR01, LAHC06, PCL11, LMnJC20]. Multi-versioning inspired a PGO that we evaluate in this paper, called A-Z (read as A-to-Z) testing, that times differently-compiled shaders online. The key difference between our PGO and multi-versioning is that *Cooperative PGO* significantly increases the number of versions that can be compared.

Any system that evaluates performance online must take care to minimize measurement overheads. We adapt prior research on sample-based profiling [AR01, Lib04, CMH*13] to simultaneously reduce overheads and better mesh with the peculiarities of computer graphics. Two of the three PGOs we explore in this paper have considerable measurement requirements and could not operate in a traditional adaptive optimization framework and meet the realtime requirements of computer graphics. We explain later how such profiling-intensive optimizations can be successfully tamed with *Cooperative PGO*.

2.3. Crowd Sourced Compilation and Profiling

The earliest example of a crowd sourced profiler is *statistical bug isolation* [Lib04, JTLL10, LNZ*05]. In his thesis, Liblit notes that with millions of people all running the same application, a low overhead, sample-based approach can be used to profile infrequent events. By the same token, with *Cooperative PGO*, millions of gamers play the same, small set of games. Even with low sampling rates, the gaming community is large enough to quickly profile whole games with high coverage. Liblit's instrumented code is compiled into the released binary, and therefore suffers some overhead even when profiling is disabled. Similarly, Stephenson et al.'s *mainstream computing* system for enforcing security constraints, is based on crowd-sourced control flow and value profiles from instrumented released binaries, and unfortunately experiences slowdowns up to three orders of magnitude greater than *Cooperative PGO* [SRYVH10]. *Cooperative PGO* leverages just-in-time (JIT) compiling technology, and includes a novel hierarchical sampling approach that dramatically reduces profiling overheads.

Fursin and Temam's *collective optimization (CO)* shares the same high-level motivation as *Cooperative PGO*, i.e. crowd-source runtime information to specialize a compiler to its target application [FT11]. CO targets *static* compilation systems and performs offline or install-time profiling to reduce the long training times of *iterative compilation*. *Cooperative PGO*, on the other hand, works in the context of a JIT and performs profile collection and aggregation *online*, where low-overhead profiling is a requirement. Iterative compilation performs compiler knob tuning, or policy search to minimize runtimes or binary sizes. *Cooperative PGO* subsumes iterative compilation by also allowing for fine-grained profile collection more suited for PGO.

We end this section by drawing a distinction between gaming workloads and CPU workloads, such as those considered in prior research. Gaming workloads tend to be heavily optimized and tuned by high-performance programmers. GPU drivers, such as those distributed by NVIDIA and AMD perform aggressive optimizations, and games are regularly patched to provide faster gameplay. Thus, our focus on gaming is a double-edged sword. Extracting additional frames per second on expert-tuned software is difficult, yet the gaming market continues to significantly reward performance gains.

3. Sparse Random Profiling

While our approach to sample-based profiling was inspired by Liblit et al. [LNZ*05, Lib04, JTLL10], and Arnold and Ryder [AR01], we have adapted both the sampling strategy and instrumentation to the particulars of computer graphics. During the rendering of a frame, the graphics driver parlays *draw* and *dispatch* API calls to sequences of data movement and shader program executions that run on the GPU. Over the years as GPUs have evolved toward general programmability, shader programs have grown substantially in complexity and can contain large and irregular control flow graphs (CFGs), hundreds of registers, and many thousands of instructions.

Therefore, profiling a single shader in its *entirety*, let alone all of the hundreds of shaders used to render a single game frame, could drop frame rates by unacceptable, and unplayable proportions. For online profiling, we adopt a sparse random profiling approach.

3.1. Key Concepts

We use a three-level hierarchical random sampler as shown in Figure 2 to reduce profiling overheads. The *frame-level sampler* randomly samples from a biased distribution to determine whether to enter a *profiling window*, which is a short period of gameplay where profiling occurs. Outside a profiling window, our framework incurs undetectable overheads. Inside a profiling window, an *API-level sampler* uniformly randomly chooses which API calls to sample. Finally, a *shader-level sampler* uniformly randomly selects one or more *point*s in a shader program to instrument from the set of eligible measurement positions. For example, with basic block based instrumentation, each basic block represents a legal instrumentation *point*. The CFG in Figure 4 has five instrumentation *point*s, one for each basic block.

If invoked enough times, our hierarchical profiler statistically guarantees coverage across all API calls and all *point*s within shaders bound to these calls. The driver controls the frame- and API-level samplers, and the backend compiler controls the shader-level sampler. We now discuss the driver and compiler support in turn, and end this section by discussing how a production system might aggregate piecemeal profiles.

3.2. Driver Support

The graphics driver orchestrates *Cooperative PGO*. The driver tracks Direct3D *present* calls to determine when frames get rendered. This serves as an extremely low overhead heartbeat to determine when to enter and exit *profiling windows*. On a *present* call, we effectively flip a biased coin to determine whether to enter a profiling window. To improve efficiency, our prototype does not sample from a Bernoulli distribution on every *present* call, but instead samples (much less frequently) from a geometric distribution to determine how many *present* calls will occur before the next "heads" [Lib04]. Once inside a profiling window, *Cooperative PGO* registers for additional callbacks to track API calls for the API-level sampler. At the end of a profiling window, set to 1,024 frames in this paper, the driver removes these callbacks.

We make the frame-level sampler pick windows of several frames rather than a single frame at a time in order to amortize shader compilation overheads over a large enough region. As discussed in Section 3.3 below, instrumentation involves spawning asynchronous shader compilations, most of which will not finish within the lifetime of a single frame. Profiling windows increase the likelihood that instrumented shaders will be used before the driver garbage collects them. Furthermore, frequent changes to the driver's callback dispatch table could introduce stutter.

Figure 3 shows the operation of the API-level sampler. The driver does not sample if it detects that the CPU's ability to queue work lags the GPU's ability to render the work. If the CPU is the bottleneck, improving the runtime performance of GPU-side shader execution is unimportant. Furthermore, as Figure 3 shows,

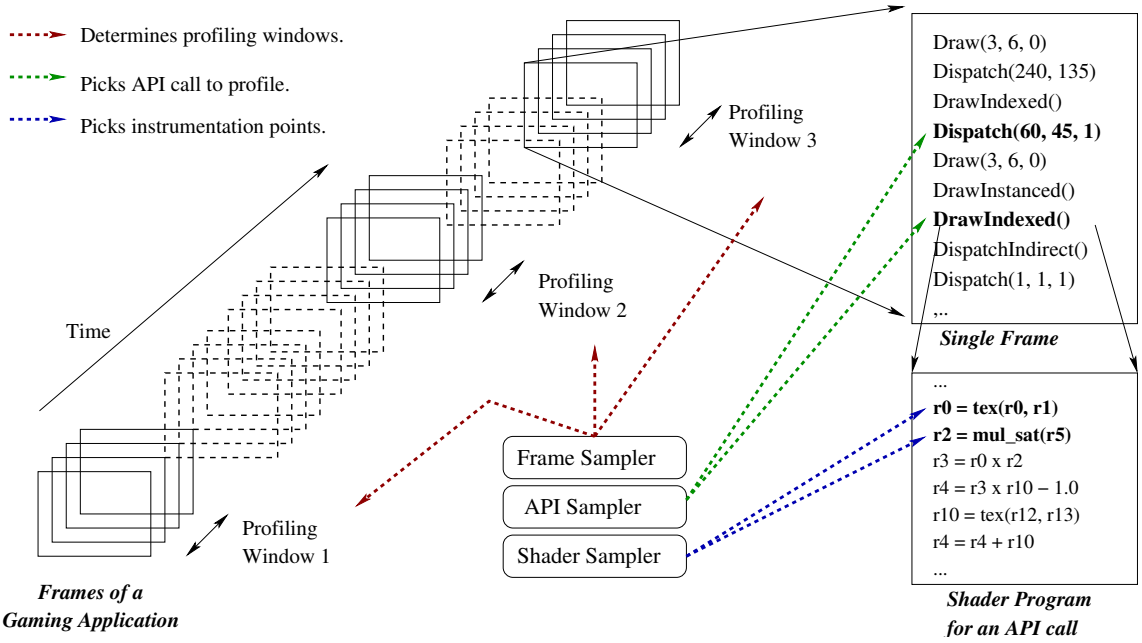


Figure 2: *Hierarchical random sampling in a single client of the* Cooperative PGO *system.*

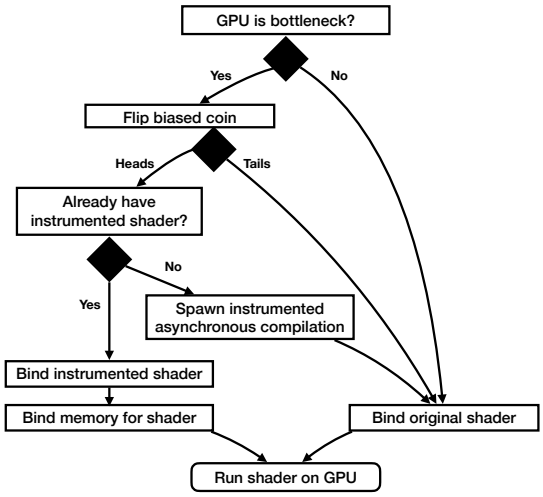


Figure 3: *The driver's API-level sampler. Most* draw *and* dispatch *calls use uninstrumented shaders. Occasionally the driver will spawn non-blocking asynchronous compiles to instrument shaders for profiling.*

instrumentation involves *recompiling* shaders on CPU-side compilation threads, which will additionally burden the CPU and cause frame rates to drop.

However, if the driver determines that the GPU is the bottleneck, then the API-level sampler elects to profile with probability p when binding *draw* or *dispatch* calls to execute on the GPU. Assuming sparse random profiling (e.g., $p < 0.01$), the driver will almost always bind original, *uninstrumented* shaders, which will execute on the GPU with no overhead. This sampler also performs a coin toss by sampling from a geometric distribution [Lib04]. Occasionally the biased coin toss will come up "heads" and the driver will initiate sampled profiling.

The driver will first check to see if it already has an instrumented shader variant for the affected *pixel* or *compute* shader. (Our work does not consider *vertex*, *geometry*, or *tessellation* shaders because they are rarely performance bottlenecks.) If the driver has not generated an instrumented variant, it will spawn an asynchronous instrumented compilation of the shader. Importantly, the driver does not block the current draw or dispatch call waiting for compilation to finish, but will instead use the original, uninstrumented shaders. Some future sample will use the instrumented shader when compilation finishes. If the driver has an instrumented shader variant, then instead of binding the original shader, the driver will perform these additional steps:

- If the driver has not already done so, the driver allocates memory into which the instrumented shader can write its profiling information during execution.
- The driver binds the *instrumented* shader that will, when it executes on the GPU, record profiling information in GPU memory.

The driver transfers profiling data in GPU memory to system memory or disk at regular intervals. At the end of a user session, this data will be packaged and shipped to a centralized server.

3.3. Compiler Support

As is common for PGO, our *Cooperative PGO* system relies on the compiler to insert instrumentation code during compilation such that when the instrumented shader runs on the GPU it will record profiling information into memory that the driver binds to the shader. When the driver decides to instrument a shader, it uses dedicated compilation threads to asynchronously compile.

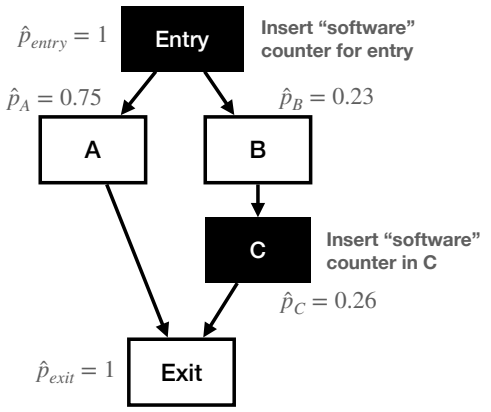


Figure 4: *Example shader CFG. Our approach sparsely samples the space of* point*s. In this example, block C is randomly sampled causing the compiler to instrument a "software" counter in C and another normalizing counter in the entry.*

Typical PGO instrumentation is intrusive and has substantial runtime overheads. Instead, we sparsely instrument shaders as we show in Figure 4. In this simple example we show how we can sparsely randomly instrument basic blocks with "software" counters to estimate basic block frequencies relative to the shader's entry. At run-time, the driver passes a pointer to an array of counters that it allocated to record the profile statistics. The compiler, therefore instruments the shader by adding fast L2 atomic increment instructions [Mic18a], appropriately indexed into the counter array, to a randomly chosen block and the entry block. In Figure 4 the chosen block is C, and we can estimate its frequency, $\hat{p}_C$ by normalizing the value in C's counter by the value in the entry block's counter. The number of *point*s that the system measures in one batch is controllable, and we call it the batch size, b. For traditional PGO, the batch size is the entire space of *point*s, but *Cooperative PGO* enables batch sizes as small as one.

Normalized counter values are a key element of *Cooperative PGO* because samples or piecemeal profiles of the same shader program may correspond to different game resolution settings or different scenes. Thus, absolute counter values in isolation, such as the execution count of a basic block or zero-value counts, must be normalized to add meaning in a *Cooperative PGO* system. We normalize counter values with respect to the number of invocations of the shader program, which is simply the execution count of the program entry basic block, and is tracked for all targeted PGOs.

A *Cooperative PGO* system experiences sampling errors commensurate with the number of samples recorded. In Figure 4 blocks B and C *should* have the same execution frequency since B falls through to C, but the sampling process can cause small inconsistencies. We do not fully characterize the accuracy of the profiles compared to ground truth, in part because prior art has characterized the accuracy of sample-based profilers [AR01], and because for the profile-guided optimizations we consider in this paper, accuracy is not paramount. For instance, basic block reordering focuses on moving *cold* blocks to the end of a program; it is not important exactly how cold they are, just that they are cold.

3.4. Aggregator

To construct a whole program profile, an aggregator assembles the discrete piecemeal profiles into a data structure representing the behavior of the entire program. The aggregator runs as a service on a centralized server and communicates with the distributed clients participating in piecemeal profiling. The aggregator receives piecemeal profiles from various clients in a gaming cloud and accumulates these profiles against the corresponding game's profile database.

In our prototype, the aggregator continually accumulates incoming piecemeal profiles in their raw form, which just includes raw counter values, metadata to indicate the profile type, the *point*s in the space that the client profiled, and a normalization counter that the aggregator uses to normalize the raw counter values. The aggregator maintains a database of every *point* it has ever received from a client. In our prototype, an administrator explicitly requests that the aggregator generate a curated *PGO database* that a client's PGOs use during compilation. We currently only populate the PGO database for shaders in which we have gathered at least one sample for every *point* in the shader's space. Where more than one sample exists for a *point*, the aggregator normalizes each sample, then averages the normalized samples. In our experiments the hottest shaders were well-covered, with several samples per point.

In an actual deployment, with an abundance of raw profiles with which to work, an aggregator would only populate the PGO database with profiles for shaders in which it has reasonable confidence in each *point*'s estimated value, for example by using solutions to the *polling problem* [BT02]. Additionally, it could afford to only maintain a sliding window of piecemeal profiles, discarding the oldest profiles to better capture changing trends in gameplay. Our prototype's PGO databases contain exactly one value per *point* (e.g., a PGO database of basic block frequencies contains one floating point number per block, which contains the block's estimated frequency). However, an aggregator could easily construct *histograms* of normalized values for each *point* for PGOs designed to consider profile distributions [HNL*13]. The aggregator could also compute age-weighted averages, confidence intervals, variance, and more for each *point* in the space.

4. Case Studies

This section describes three profile-guided optimizations that we evaluate in the context of *Cooperative PGO*.

4.1. Case Study I: Using Basic Block Frequencies

The canonical PGOs, and those which are available in most commercial compilers, leverage dynamic basic block frequency information to make smarter compilation decisions. We modified four compiler optimizations to use basic block frequency information:

- Predication: We modified our baseline predication heuristic to not predicate blocks that are infrequently executed ($< 1\%$).
- Loop unrolling: We modified our loop unroller to forgo unrolling cold loop bodies.
- Block layout: We modified our final code generator to reorder basic blocks such that cold blocks are moved to the end of the shader.

```
b0 = expensive0()
b1 = expensive1()
b2 = expensive2()
r0 = TEX // likely zero
r4 = r0 * b0
r5 = r4 * b1
pixout = r5 * b2
```

(a) *Original code.*

```
r0 = TEX
if (r0 == 0.0) {
    pixout = 0.0
} else {
    b0 = expensive0()
    b1 = expensive1()
    b2 = expensive2()
    r4 = r0 * b0
    r5 = r4 * b1
    pixout = r5 * b2
}
```

(b) *Transformed code.*

Figure 5: *Zero-value specialization concept.*

- Register allocation: We modified the register allocator to give allocation preference to hot basic blocks.

While basic block PGOs attempt to optimize various aspects of control flow, the next PGO we present optimizes data dependencies and introduces additional control flow to shader programs.

4.2. Case Study II: Zero-Value Specialization

Rangan et al. demonstrated that zero-value specialization can improve the performance of graphics applications and present a fast-slow versioning transform called *Zeroploit* [RSU*20], which is illustrated in Figure 5. In Figure 5a, if the texture operation is likely to return zero, then because `r0` feeds a multiply chain, we see that `r4`, `r5`, and `pixout` are also likely to be zero. Depending on the particulars of the "expensive" functions and the likelihood that `r0` is zero, it may be profitable to transform the code into that shown in Figure 5b. If `r0` is frequently zero, this specialized function will forgo executing the expensive functions and will simply write a zero into the `pixout` register. However, since we are not guaranteed that `r0` will always be zero, the transformation includes a slow fallback path. In this example, we call `r0` the versioning variable as it determines whether to execute the optimized or unoptimized version of the code.

While this transformation is not IEEE 754 compliant [Mic19a], game developers typically allow for such IEEE-unsafe floating point optimizations to opportunistically squeeze out additional performance as well as to ensure that NaN values do not leak into render targets. For example, setting the *refactoringAllowed* [Mic18c] global flag and dropping the *precise* storage class specifier for variable declarations [Mic18f] in Microsoft's high-level shading language (HLSL) explicitly permits the reassociation and optimization of floating point operations.

Stephenson and Rangan recently described an automatic compiler technique for *Zeroploit* called *PGZ* [SR21], which we use as our second case-study. Below, we provide a high-level overview of *PGZ* and discuss how *PGZ* can benefit from a *Cooperative PGO* framework.

4.2.1. The *PGZ* algorithm

Algorithm 1 presents the general approach of *PGZ* [SR21], which we have faithfully implemented in our prototype. *PGZ*'s first task, EnumerateCandidates, is to enumerate the set of possible versioning variables in a shader, referred to as *candidates*. Any write of a virtual register is a candidate. In Figure 5a the set of candidates is $\{b0, b1, b2, r0, r4, r5, pixout\}$.

Algorithm 1: *PGZ.*

Input: *shader*, the shader program to be transformed.
Input: *profile*, the associated value profile.
Output: *xformed*, the transformed shader.
$N_{transformed} \leftarrow 0$
$cands \leftarrow$ EnumerateCandidates$(shader, profile)$
repeat
  foreach $c \in cands$ **do**
    $Vr_{guard} \leftarrow$ CandidateVr(c)
    $pzero \leftarrow$ CandidatePZero(c)
    $\{\mathbb{C}, \mathbb{D}\} \leftarrow$ CandRemovalSets(Vr_{guard})
    $score[c] \leftarrow$ EstimateSavings$(\mathbb{C}, \mathbb{D}, pzero)$
  end
  if $\exists\, s \in score > T$ **then**
    $c \leftarrow i$ **where** $score[i] = max(score)$
    Transform(c)
    $cands = cands - c$
    $N_{transformed} \leftarrow N_{transformed} + 1$
  else
    Return
  end
until $N_{transformed} > 2$

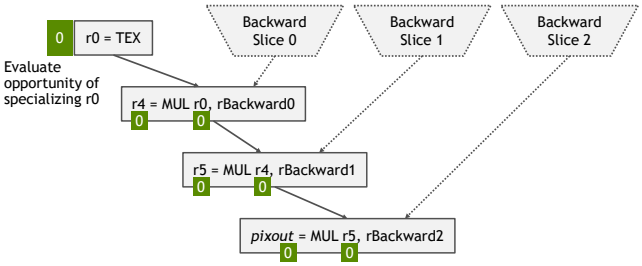


Figure 6: *Evaluating a candidate. Constant propagation and folding can convert some instructions to constant literal moves. A subsequent dead code elimination pass removes backward slices of computation that are no longer needed.*

With a set of candidates to consider, the algorithm determines the benefit of each candidate without actually transforming the code. Most of the complexity of *PGZ* is contained in CandRemovalSets, which propagates a candidate's zero value, then applies dead code elimination to remove unneeded instructions. Figure 6 illustrates the approach on a simple example, which shows CandRemovalSets' evaluation of variable `r0`. Constant propagation causes all subsequent instructions to evaluate to zero, and the subsequent dead code elimination analysis marks the backward slices of computation as dead since they are not required to produce constant literals (shown with dotted lines in the figure).

After evaluating CandRemovalSets, Algorithm 1 estimates the effectiveness were it to transform the given candidate. The EstimateSavings function (beyond the scope of this paper) examines the set of instructions that would be converted to constants

or eliminated, along with the likelihood p_{zero} that the candidate is zero to estimate the dynamic worth of applying the transformation. Candidates with p_{zero} less than a threshold of 0.32 are ignored by *PGZ*'s heuristic. As recommended in the *PGZ* paper, we pass the versioning variable through a *vote.all* operation [NVI21b] to ensure that the dynamic branching condition is warp-convergent. This process ensures that the zero-specialized fast path is taken only if all threads are in agreement. Otherwise the default slow path is executed dynamically. For more details, we refer the reader to the *PGZ* paper [SR21].

4.2.2. Profiling Requirements

PGZ relies on a profile that contains P_{zero} for all of a shader's candidates. In addition, the EstimateSavings function uses dynamic basic block weights to estimate the runtime impact of the transformation. To profile for P_{zero} we associate two counters for every candidate: one tracks the number of writes to the candidate in total, and the other tracks the number of writes to the candidate in which *all* threads of the warp wrote zero. While the instrumentation required to profile for zero-value frequencies is straightforward, some shaders have hundreds to *thousands* of candidates. In this case, the overhead of online profiling would be prohibitive, making *PGZ* an excellent fit for a *Cooperative PGO*-based implementation.

4.3. Case Study III: A-Z Testing

A-Z testing is our name for a common adaptive optimization strategy that dynamically measures the runtime of N code variants, and then chooses the fastest variant. This is a straightforward, well-studied approach that has found success in a number of different settings, from static code compilation to just-in-time code generation [DR97, AR01, LAHC06, PCL11, LMnJC20]. The only difference between A-Z testing and prior research is that A-Z testing, by virtue of running within a *Cooperative PGO* system, can consider a huge number of code variants.

We tested 24 different compiler flag settings for this PGO. These flags control a range of scheduling parameters, register allocation settings, and disabling of compiler passes such as predication, loop invariant code motion, and loop unrolling. We did not test combinations of these flags since our goal here is not to discover the maximum benefit from this PGO, but to prove that *Cooperative PGO* in a cloud gaming system makes this PGO viable. As Section 6 shows, the single-flag-at-a-time approach demonstrates that this PGO can produce tangible performance benefits. When deployed in a production gaming cloud, we expect that the variant testing PGO can afford to test flag combinations commensurate with the popularity of a game. The greater the popularity, the more the combinations that can be tested in the profiling phase, at the specified sampling rate. A machine learning technique could potentially guide a sophisticated search through the space of knob settings [AKC*18].

The profiling requirement for this PGO is simply the ability to measure whole shader execution times. This measurement can be performed either by using API-level timestamp queries [Mic18e], which our prototype does, or by reading the GPU clock with assembly instructions [NVI21b] at the beginning and the end of a shader program, obtaining their difference to get shader latency, and then averaging the latencies across all warps of a shader program.

Table 1: *Gaming applications evaluated in this paper.*

Application	Short	APICs	dx11	dx12
Ashes of the Singularity - Escalation	Ashes	3	✓	
Deus Ex Mankind Divided	DXMD	1		✓
Final Fantasy XV	FFXV	1	✓	
Metro Exodus	Metro MetDemo	1 demo	✓ ✓	
PlayerUnknown's Battlegrounds	PUBG	3	✓	
Horizon Zero Dawn	HZD	1		✓
WatchDogs Legion	Watch	1		✓
Red Dead Redemption 2	RDR	1		✓
Serious Sam 4	SS	3	✓	

5. Methodology

This section describes how we emulate *Cooperative PGO*. Because our candidate PGOs focus on improving the performance of shader programs running on a GPU, we consider games that are GPU-limited. Table 1 lists the gaming applications evaluated in this paper. Using an internal frame-capture tool, similar to publicly available tools like Renderdoc [Kar21] or Nsight [NVI21a], one or more random single-frames are captured from a built-in benchmark or from actual gameplay of each gaming application, depending on whether the game has a built-in benchmark. These single-frame captures, called APICs, contain all the information needed to replay a game frame (i.e., the API sequence, including all relevant state, as well as shader programs used in individual calls). For applications represented by more than one APIC, we made sure to capture each APIC from visually different scenes. The APICs were captured at either 1440p or 4k. APIC-based experimentation enables controlled piecemeal profile aggregation and repeatable benchmarking, which allowed us to perform in-depth analysis and debugging. The APICs we use were captured by NVIDIA's Quality Assurance team from popular games. Such QA-captured APICs are routinely used to understand and improve production GPU driver performance, and speedups on these APICs regularly translate well to gameplay.

We evaluate our candidate PGOs on a total of 15 APICs spanning 9 applications as shown in Table 1. We hyphenate an application's short name with the APIC number to uniquely refer to an APIC (e.g., `SS-3` to refer to the third APIC of Serious Sam 4). For Metro Exodus, in addition to the one APIC, we also evaluate our candidate PGOs on the built-in demo (distinguished with short name `MetDemo`). Running an APIC involves replaying a single frame in a loop, for the desired frame count, whereas the demo is a multi-frame DirectX 11 application that runs for more than 8000 frames.

We perform our experiments on an NVIDIA GeForce RTX™ 2080 GPU, locked to base clock settings of 1515 MHz for the GPU core and production DRAM frequency settings. Full specification of this GPU can be found in TechPowerUp [Tec18]. We compare the final rendered image against a golden reference image to ensure that our transforms are functionally correct. We measure whole-system frames-per-second (FPS) speedups, which includes GPU and CPU execution time using accurate, in-house profiling tools. These runs use production settings for driver and compiler optimizations, which include classical optimizations such as constant folding, dead code elimination, loop unrolling, and instruction scheduling, in addition to various machine-specific optimizations.

We implemented a piecemeal profiler and our candidate PGOs as a research prototype within a recent branch of NVIDIA's GeForce Game Ready driver. At a high level, the baseline compilation flow

operates in two phases. First, a proprietary driver frontend converts input shader program code in high-level assembly (e.g. Microsoft's DirectX ByteCode (DXBC) [Mic18d] or DirectX Intermediate Language (DXIL) [Mic19b]) to a driver-internal intermediate representation (IR1) and performs API-dependent code transformations and peephole optimizations. Second, a proprietary backend optimizing compiler takes the IR1 output from the driver, translates it to a compiler-internal intermediate representation (IR2), applies classical and machine-specific optimizations, and finally compiles optimized IR2 programs to the binaries that ultimately execute on the GPU's SMs. Our prototype's code instrumentation and PGOs operate at the IR2 level in early passes of a modified version of the production backend compiler. Our prototype uses a hash of a shader's IR1 output from the driver along with driver and compiler knobs to associate shader variants with profile data.

In a *Cooperative PGO* system, aggregated profiles will evolve over time, possibly exposing new optimization opportunities and/or new information about previously transformed shaders. For example, for *PGZ*, new *point*s in a program might be identified as likely zero over time as gamers traverse different levels, scenes, and settings of a game. A question arises about how one might profile and re-optimize shaders that have already been specialized by *Cooperative PGO*. Our PGO and instrumentation facilities, which we implemented in the backend compiler, sidestep this issue by always using the *unspecialized* IR1 output from the driver's frontend compiler as the starting point for transformations and profiling. To specialize a shader the driver frontend passes an immutable IR1-representation of the shader code to the backend compiler along with that shader's aggregated profile data, and the end result of the compilation is a specialized binary that cannot be further transformed nor serve as input to subsequent compiler passes. Later, if the driver is in a *profiling window* and the API Sampler chooses to instrument the same shader, the driver will again pass the shader's immutable IR1 representation to the backend compiler, which will perform the instrumentation *without performing PGO specialization*. This approach fits naturally with the production driver's implementation and does not preclude any developer, driver, or compiler optimizations.

We could not deploy our infrastructure on a live gaming cloud because it is in the research stage. Instead, we emulate *Cooperative PGO* on a single GPU system as follows. We collect thousands of piecemeal profiles for each APIC and the one demo in our test suite by sparsely sampling with a large enough batch size that profiles converge reasonably fast (1 to 2 days). Next, in an offline pass, we aggregate these piecemeal profiles and dump the aggregated profiles to the disk. This offline approach enables controlled aggregation of profiles across APICs and allows us to perform interesting cross-validation experiments, which we present in Section 6.5.

Our candidate PGOs use the aggregated profiles from disk to perform the respective PGOs in our JIT compiler in dedicated runs, which are used for FPS measurements to gauge the performance of our PGOs. These runs still have very low-frequency sparse random sampling enabled to mimic the negligible piecemeal profiling overhead we expect to see in the client nodes of a real world *Cooperative PGO* system.

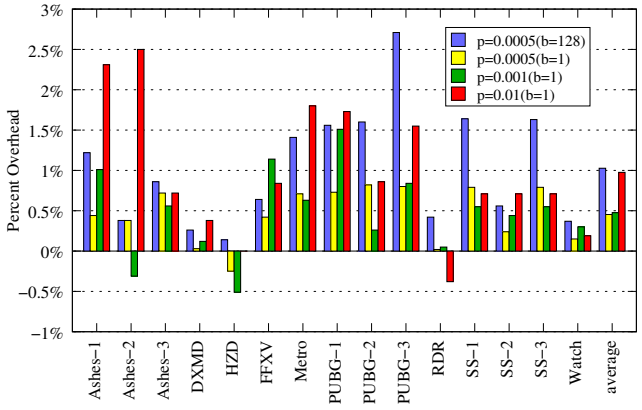


Figure 7: *Profiling overhead during a profiling window.*

6. Results

This section presents results from our *Cooperative PGO* prototype. We first quantify the overhead of piecemeal profiling and the convergence rates for our three candidate PGOs, and then present performance speedups for the candidate PGOs using the profile information from their corresponding *Cooperative PGO* systems.

6.1. Profiling Overhead

Recall from Section 3.1 that in our *Cooperative PGO* system, profiling occurs only during sporadic profiling windows of 1,024 consecutive frames, and then gets turned off completely. When a *Cooperative PGO* system is not in a profiling window, there is no overhead. In this section, we present the overheads that gamers would see during a profiling window. There are two knobs that we use to control shader instrumentation, and both directly affect profiling overheads. The first is the sampling probability p, which sets the probability that our biased coin comes up heads when performing API-level sampling on *draw* or *dispatch* calls. The smaller p is, the less likely that the system will instrument a shader. The second knob is the batch size b, which controls how many points in the space we instrument, and therefore measure in one batch. In Figure 4, because $b = 1$ we measure one point in the space, the frequency of execution of block C. However, with $b \geq 5$, we measure the frequencies of all blocks in the example CFG in one shot.

Figure 7 shows the overhead of our prototype during a 10 second profiling window of game rendering. For each APIC on the X-axis, the figure shows the average frame-rate slowdown as a percentage of the uninstrumented baseline. Each bar corresponds to a given p and b, as shown in the legend. The average frame-rate slowdown is quite small for sparse sampling, usually well under 1%, and we do not see an increase in stutters. Since even these low overheads will offset potential gains from any PGO, we recommend profiling only during sporadic profiling windows. By adjusting the probability of the frame sampler entering a profiling window, an engineer can drive the already low overhead in Figure 7 to negligible levels.

6.2. Profiling Coverage and Accuracy

With sparse random sampling of a space, we can use solutions to the *coupon collector's problem* to estimate how many windows we

Table 2: *Sampling statistics for zero-value profiling.*

Game	Shaders per run		Shaders	#Points	E[W] $b=1$	
	p=0.001	p=0.01			p=0.001	p=0.01
Ashes-1	44	67	154/185	40,504	10,298	6,763
Ashes-2	57	74	128/177	35,306	6,844	5,272
Ashes-3	48	68	117/157	28,273	6,377	4,502
DXMD	191	472	465/495	75,651	4,678	1,893
HZD	0	0	286/302	119,690	-	-
FFXV	121	290	419/459	103,765	10,400	4,339
Metro	73	136	184/195	34,743	5,251	2,819
PUBG-1	84	157	241/255	41,765	5,577	2,984
PUBG-2	107	182	251/268	47,471	5,033	2,959
PUBG-3	102	189	268/282	51,338	5,750	3,103
RDR	95	176	179/196	73,286	9,087	4,905
SS-1	46	81	114/129	21,901	5,033	2,858
SS-2	82	141	213/252	59,288	8,364	4,864
SS-3	83	135	191/231	53,651	7,413	4,557
Watch	60	277	627/636	250,909	54,406	11,785

expect to sample (with replacement) before we have covered the entire space [MR95]. If, for example during a profiling window, we measure a single, random basic block's normalized frequency, we would like to know how many windows we expect to observe before we have an estimate for all N basic block frequencies in the shader's CFG. The expected number of windows, W, for any N is:

$$E[W] = N\sum_{i=1}^{N}\frac{1}{i}, \tag{1}$$

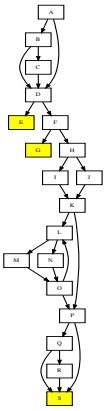
which grows as $\Theta(N\log N)$. Thus, for a CFG with 25 basic blocks, we can expect to have to sample about 95 times to have a measurement for each basic block. We will of course need to sample each point in the space multiple times so that we can gain statistical confidence in our estimates. Assuming that the system samples the shader at least once in $\frac{1}{4}$ of the windows, we expect to require $100\times 4\times 95 = 38{,}000$ windows to have sampled each basic block in the shader's CFG 100 times.

We have focused our discussion on basic block execution weight profiling because basic block PGOs are commonplace. However, the profiling space for basic block profiling is substantially smaller than that of zero-value profiling, which the *PGZ* PGO uses. Table 2 shows the sampling statistics for our system on several game scenes for zero-value profiling. The "Shaders per run" column shows how many shaders our prototype samples during a single 1,024 frame window with sampling probabilities of 0.001 and 0.01.

Our experiments perform heavy sampling, with $p = 0.02$ and $b = 128$, which allows us to more quickly cover the profiling space without a community of users. We were able to completely profile most of the shaders used to render the given scenes, which the "Shaders" column shows as a fraction of *completely* sampled shaders over all sampled shaders. As the data show, a given shader's space for *PGZ*'s zero-value profiling is large. For example, we completely profiled 128 shaders in `Ashes-2`, which combined for 35,306 points, yielding an average of 276 points per shader. The "E[W]" column shows the expected number of 1,024 cycle sampling windows we would need to perform to get the same level of coverage for a given p when $b = 1$. The expected number of windows is small compared to the number of frames for which these games are played, and thus a production *Cooperative PGO* system would have no problem achieving complete profile coverage.

For context, at the time of this writing over 900,000 gamers were *actively* playing the top played game *Counter-Strike: Global Offensive* on Steam [Ste20]. At a nominal 30 FPS, a *Cooperative PGO* system can effectively sample from among a population of 27 million FPS (i.e. $900{,}000\times 30$ FPS). The average number of active gamers per game across the top 100 games on Steam is upwards of 50,000, which implies an effective population of 1.5 million FPS ($50{,}000\times 30$ FPS). With such large populations to sample from, *Cooperative PGO* can comfortably collect the required number of samples within a short period of time, while minimizing profiling overheads with low values of p. Concretely, applying the above to `Watch`, the application in Table 2 with the largest profiling space, the wall-clock time needed in a *Cooperative PGO* system to profile ~55,000 windows is $\frac{55{,}000\times 1{,}024}{1.5\text{ million FPS}}$, which equals 37.54 seconds. The above assumes profiling is always on so that when one window ends, the next immediately begins. If the frame sampler were to pick profiling windows with a 0.1 probability, the wall-clock time needed for convergence would grow by 10x, which would still amount to a modest 6.23 minutes. Different statistical techniques, based on the variance in the data, could be used to determine how many samples are required to ensure a specified confidence level.

Table 3: *Convergence of basic block frequencies for a compute shader in* `MetDemo` *with* $p = 0.05$ *and* $b = 2$.



PMP	45	100	500	1000	2000	2492	GT
SPB	1-8	7-12	49-54	99-106	205-214	251-265	-
BB							
A	1	1	1	1	1	1	1
B	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0
D	1	1	1	1	1	1	1
E	0	0	0	0	0	0	0
F	1	1	1	1	1	1	1
G	0	0.03	0.07	0.07	0.07	0.06	0.07
H	1	0.98	0.99	0.97	0.97	0.97	0.97
I	0.99	0.95	0.95	0.95	0.95	0.95	0.96
J	0.41	0.49	0.49	0.48	0.48	0.48	0.49
K	0.95	0.98	0.97	0.97	0.97	0.97	0.97
L	1.52	1.85	2.12	1.89	1.95	1.95	1.91
M	0.42	0.36	0.39	0.41	0.40	0.41	0.45
N	1.63	1.41	1.17	1.37	1.35	1.38	1.57
0	2.06	2.22	2.17	2.00	2.03	1.98	1.91
P	0.92	0.95	0.97	0.96	0.96	0.96	0.97
Q	0.67	0.76	0.84	0.83	0.85	0.85	0.81
R	0.60	0.73	0.85	0.83	0.84	0.84	0.81
S	0.99	0.98	0.98	0.97	0.97	0.97	0.97

For basic block profiling, our applications average around 17 blocks per shader, while for for A-Z testing there is a constant, 24 knob settings per shader. Though these profiling spaces are smaller than *PGZ*'s, they are still too large for traditional adaptive optimization with over 5,000 and 6,700 points per scene respectively. Low profiling overheads coupled with modest profile collection times make *Cooperative PGO* an attractive proposition for distributed gaming systems and gaming clouds.

6.3. Case Study: Profile Convergence in `MetDemo`

`MetDemo` is a multi-frame, built-in demo from the application `Metro Exodus`. To illustrate how piecemeal profiling works across frames and scenes of a gaming application, we show in Table 3 the control flow graph, with basic blocks *A* through *S*, of a compute shader from `MetDemo` and how its BB execution weights get aggregated and change over time. The top row, titled PMP, lists the number of piecemeal profiles collected (with $p = 0.05$ and $b = 2$) over time with "GT" representing ground truth, obtained by aggregating BB execution weights across the entire demo. The SPB

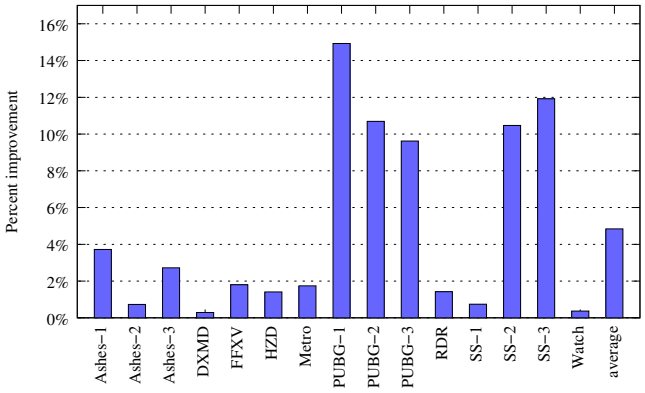


Figure 8: PGZ *isolated single-frame performance gains.*

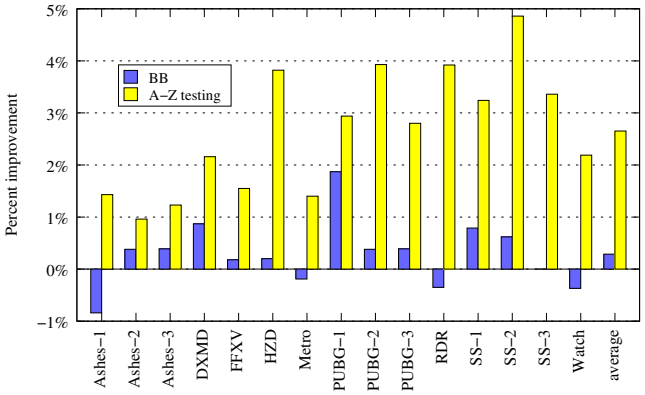


Figure 9: *BB PGOs and A-Z testing isolated single-frame performance gains.*

row shows how many samples per block, as a range, each block has received. After accumulating 45 piecemeal profiles, the aggregator has finally seen at least one sample for each of the 19 blocks. As time progresses from left to right, more piecemeal profiles are collected and aggregated from different frames, causing execution weights of individual basic blocks to evolve before they stabilize at values that are close to ground truth. This simple example illustrates how sparse random sampling eventually converges to the ground truth, which prior work has also demonstrated [AR01].

6.4. Single-Frame Testing

To test the soundness of our profiling data and PGOs, we perform isolated single-frame testing. We first repeatedly run each of the APICs in Table 1 with sparse sampling enabled ($p = 0.02$, and $b = 128$) to collect frame-specific piecemeal profiles. We then run our aggregator to create frame-specific PGO databases and test the speedup we obtain on each frame with its custom PGO database. Profiling consumes at least a day per application.

Figure 8 shows the results of this experiment for *PGZ*. The *PGZ* PGO performs well for `PUBG` and `SS`, achieving gains of 15% and 12%, respectively. Key shaders in those APICs contain *PGZ* opportunities that eliminate large portions of computation when likely-zero variables are dynamically zero. Most other applications achieve lesser, but real speedups, and demonstrate that *Cooperative PGO* is viable.

Figure 9 shows the results for our basic block PGOs and A-Z testing. We implemented basic block PGOs because they are staples in the literature. Unfortunately, the PGOs we implemented provide little benefit for typical shader programs, in part because pixel shaders, which often dominate GPU utilization tend to have straightforward control flow (where single basic block shaders are not uncommon). Except for `PUBG-1`, which achieves a nearly 2% gain, we do not see an overall performance advantage. A-Z testing, on the other hand performs well, achieving a nearly 3% average gain. Hot shaders in these applications show an affinity to one or more of the knob settings with which we performed this experiment, further reinforcing prior art that studies application sensitivities to compiler settings [KKOW00, AKC*18].

Table 4: PGZ *transformed candidates.*

Game	P1	P2	P3	P1+P2	P1+P3	P2+P3	P1+P2+P3
Ashes-1	**61**	13	12	**60**	**57**	13	**59**
Ashes-2	12	**62**	12	**62**	12	**62**	**62**
Ashes-3	12	13	**56**	13	**58**	**56**	**57**
PUBG-1	**43**	12	12	**46**	**46**	15	**49**
PUBG-2	12	**56**	8	**59**	17	**59**	**62**
PUBG-3	6	8	**56**	8	**60**	**61**	**59**
SS-1	**9**	0	0	**9**	**9**	0	**9**
SS-2	1	**45**	4	**47**	5	**46**	**46**
SS-3	1	4	**39**	5	**40**	**39**	**40**

6.5. Multi-Frame Cross-Validation Experiments

The single-frame tests we described above demonstrate that the *piecemeal profiling* component of a *Cooperative PGO* system works. We now test whether the PGOs with which we evaluate *Cooperative PGO* can generalize across game scenes. We take a traditional approach to showing generalization: we individually profile APICs as we did above, and then we apply one APIC's PGO database to another APIC from the same game. We study cross-validation on `Ashes`, `PUBG`, and `SS`, which have 3 APICs each.

Table 4 investigates how many transformations *PGZ* performs for each scene using a variety of PGO databases, which is a proxy for the quality of the PGO database. For each application, we create three scene-specific PGO databases by exclusively training on each of the three varied scenes we collected for the application. The `P1` column shows how many transformations *PGZ* performs when we use a PGO database using profiles exclusively from the first scene of each application. For example, the `P1` column for the `Ashes-2` application uses the PGO database created with `Ashes-1` profiles to compile `Ashes-2`. The `P2` and `P3` columns represent the PGO database using profiles from the second and third scenes for each application, respectively; and `P1+P2`, `P2+P3`, `P1+P3`, and `P1+P2+P3` refer to different combinations of aggregated PGO databases from the associated scenes of the corresponding games.

Take `SS-1` as a concrete example: *PGZ* transforms 9 candidates using the PGO database gathered exclusively from `SS-1`, but it transforms none when using `P2` and `P3`. Encouragingly, with the exception of `Ashes-1` the data shows that combining piecemeal profiles across frames provides *better* profiling coverage in a *Coop-*

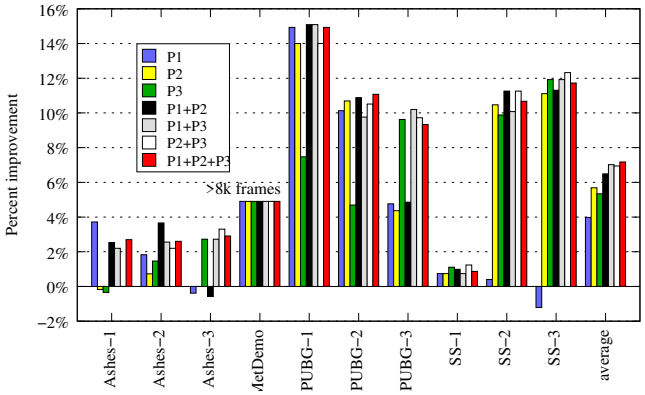


Figure 10: *Frame-rate speedup for* PGZ *on an in-game demo and with cross validation across APICs.*

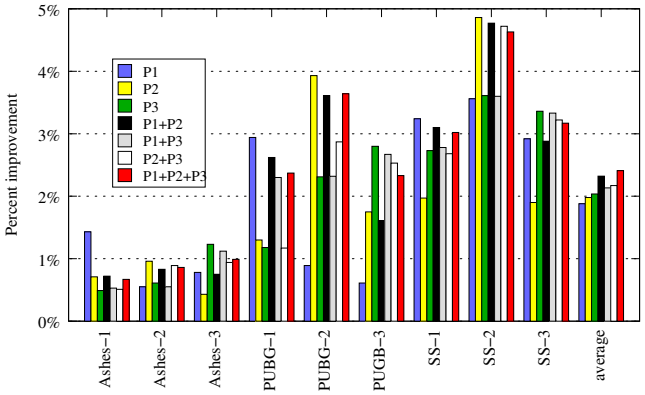


Figure 11: *Frame-rate speedup for A-Z testing with cross-validation across APICs.*

erative PGO system. In fact, the `P1+P2+P3` PGO database, generally identifies the most *PGZ* opportunities across all of the scenes.

Figure 10 shows the frame-level *PGZ* speedups from applying various PGO databases to our APICs. We generally expect the APIC-specific PGO databases to perform at, or near the peak, which is mostly true for these APICs. The red bars show that the combined `P1+P2+P3` PGO database performs well across all of these applications, and is the best database on average. The performance correlates well with the data in Table 4 and lends credence to the idea that aggregating profiling data across varied scenes can improve profiling coverage. In addition to performing the cross validation studies on APICs, we thoroughly profiled the `MetDemo` in-game benchmark, which consists of over 8,000 frames, for two days on a single machine. As Figure 10 shows, using an aggregated PGO database, our *PGZ* PGO recorded a 4.9% FPS speedup on the full demo. We did not evaluate `MetDemo` on BB and A-Z PGOs.

Figure 11 shows the performance of the A-Z PGO using the same cross validation setup we described above. Here too, we see that the red bar representing the `P1+P2+P3` database performs the best on average and also generalizes well across databases. This result is intuitive since many game scenes share the same important shaders, and many compiler knobs improve *static* shader properties (e.g., instruction grouping and scheduling) that are largely data-independent.

7. Towards a Fully-Distributed *Cooperative PGO* System

Our single-node emulation allowed us to demonstrate that *Cooperative PGO* can enable interesting optimizations that require profiles too expensive to collect fully at runtime. This section outlines the tasks required to deploy *Cooperative PGO* in an actual cloud, which range from straightforward engineering to investigating open questions and addressing privacy concerns.

7.1. Open Questions

One problem that could afflict *Cooperative PGO* is *profile dilution*. As we discussed in Section 6, `Ashes−1` discovered fewer *PGZ* opportunities with the combined PGO database than it did with the scene-specific PGO database. For eleven other candidates in `Ashes−1`, several zero-value probabilities dropped in the combined `P1+P2+P3` PGO database, when compared to the `P1` PGO database. Though the zero-value probabilities still triggered *PGZ*'s threshold-based heuristic, further *dilution* by adding in profiles from yet other scenes, could eventually cause probabilities to drop below the threshold and mask opportunities.

While a deeper exploration of the profile dilution problem and its solutions are beyond the scope of this paper, a *Cooperative PGO* system could tabulate *distributions* of profile values from the piecemeal profiles, including joint distributions with large enough communities. Enhanced PGOs that consider distributions of data for each *point*, such as presented by Homescu et al. [HNL*13], could better protect against dilution. Another alternative is to define an application programming interface (API) for the game developer to hint to a *Cooperative PGO* system about scene changes or other drastic differences, which can then be used to perform scene-specific clustered aggregation. An API-based approach could similarly suit the adaptive optimization frameworks in modern drivers.

7.2. Engineering Details

Recall from Section 3 that a *Cooperative PGO* system has three main components: a driver that orchestrates piecemeal profiling and interacts with a central server to obtain aggregated profiles; a compiler that performs the instrumentation for piecemeal profiling and implements PGOs; and an aggregator that runs on a central server and creates useful whole program profiles by accumulating per-shader piecemeal profiles. In a cloud deployment, the driver and compiler changes will be included in the user-mode driver (UMD) installations on individual nodes of the cloud.

The aggregator, in its simplest form, will run as a daemon on a high-availability central server that is accessible, over a private network, to the UMD instances on the individual nodes of the cloud. The central server will be walled off from all external traffic except for clients in the same cloud, i.e., the clients terminate TCP connections to gamers. Best practices in systems engineering will be used to set up low overhead connections between the UMD instances and the aggregator daemon, and to provide quality of service (QoS) guarantees on the central server. The latter may necessitate dropping piecemeal profiling packets from individual cloud

nodes to avoid overwhelming the central server and causing denial-of-service (DoS) attacks. Hierarchical or distributed aggregator services may also be considered.

Since *Cooperative PGO*'s ultimate objective is to improve gameplay performance, we must ensure that in addition to compilations for instrumented shaders, network and disk activity on the client nodes occur off the critical path of gameplay and do not affect the user experience. At the end of a gaming session on a client node, piecemeal profiles collected during gameplay and copied to system memory or disk-based circular buffers, will be encrypted and transmitted to the server. Symmetrically, at the start of a gaming session a UMD node will issue a pull request to the central server for the aggregated profile data for a given game. After obtaining the profiling data as response from the server, the UMD will trigger PGO compilations as appropriate.

Cooperative PGO's components, especially UMD clients, must be resilient to error conditions such as unreliable network transport. If the aggregator daemon becomes unreachable, clients should continue to operate without PGO enabled or with stale local PGO databases. As alluded to in Section 3.4, we expect to implement suitable replacement policies in the aggregated profile database so that the database size does not grow too large and cause problems during database updates. Although a *Cooperative PGO* system is naturally resilient against stray, lost, or malformed network packets carrying piecemeal profiling information, we plan to fully encrypt network traffic to safeguard against rogue game clients intercepting and modifying profiles and PGO databases.

7.3. Privacy and Consent

We must consider the privacy and consent ramifications of *Cooperative PGO*. We plan to copy NVIDIA's model for Geforce Experience, which attains end user consent to collect and use telemetry data at install time [Cor21]. In a cloud-based model, we instead seek consent when users create accounts, or perhaps before every session. An alternative incentive-based model is to only enable PGO compilations for users who have opted to enable profiling in *Cooperative PGO*. We will additionally take several actions to preserve privacy. First, all profiling data collected by a *Cooperative PGO* client is anonymous. The "tag" associated with profile data is a combined 192-bit hash of shader code, compiler, and driver knobs, and contains no user information whatsoever. Second, by design, individual piecemeal profiles reveal little about a program's behavior. Only when an aggregator combines a large corpus of piecemeal profiles do the program's execution trends emerge. Third, *Cooperative PGO* exclusively profiles the *pixel* and *compute* shaders in games. By not profiling host-side code we drastically reduce a user's privacy exposure. Finally, the piecemeal profiles never leave the fire-walled cloud and all communication between clients and the aggregator is encrypted. Advances in homomorphic encryption could practically eliminate a gamer's risk by allowing the aggregator to operate on fully encrypted data [WH12].

8. Conclusion

To address the shortcomings of adaptive optimizations and traditional profile-guided optimizations, we propose *Cooperative PGO*. The optimizations we presented as case studies in this paper rely on profiling information that cannot be collected online using traditional approaches. By crowdsourcing PGO profiling we simultaneously minimize profiling overheads and provide ample profiling coverage. We provide a proof-of-concept *piecemeal profiler* and explore potential PGOs that improve the performance of popular highly-optimized Direct3D games. Even with piecemeal profiling enabled, our implementation of the *PGZ* transformation yielded impressive gains of up to 15% over a highly-tuned production baseline. To the best of our knowledge we are the first to:

- Propose and evaluate crowdsourcing for PGO.
- Propose and evaluate crowdsourcing in a JIT system.
- Employ a hierarchical sampling approach that dramatically reduces overheads and makes online profiling and PGO practical for interactive gaming.

Cloud gaming, thanks to its high-performance demands and the possibility of crowdsourcing, is tailor-made for *Cooperative PGO*.

Acknowledgements

We thank our anonymous reviewers and Emmett Kilgariff for their useful feedback. Discussions with Marc Blackstein lead to the *Cooperative PGO* idea. Eric Werness, Jerry Zheng, Lee Hendrickson, Sanket Misal, and Christos Angelopoulos helped brainstorm driver and compiler implementation details. Many thanks to the Applied Architecture team for their support during this work.

References

[AFG*05] Arnold M., Fink S. J., Grove D., Hind M., Sweeney P. F.: A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE 93*, 2 (2005), 449–466. 1

[AKC*18] Ashouri A. H., Killian W., Cavazos J., Palermo G., Silvano C.: A Survey on Compiler Autotuning Using Machine Learning. *ACM Computing Surveys 51*, 5 (Sept. 2018). 7, 10

[AR01] Arnold M., Ryder B. G.: A Framework for Reducing the Cost of Instrumented Code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2001), pp. 168–179. 2, 3, 5, 7, 10

[BT02] Bertsekas D., Tsitsiklis J.: *Introduction to Probability*. Athena Scientific Books. Athena Scientific, 2002. 5

[CFE97] Calder B., Feller P., Eustace A.: Value Profiling. In *International Symposium on Microarchitecture (MICRO)* (1997), pp. 259–269. 2

[Chr20] Christoff M.: *Chrome Just Got Faster with Profile Guided Optimization*, August 2020. URL: https://blog.chromium.org/2020/08/chrome-just-got-faster-with-profile.html. 2

[CL99] Cohn R., Lowney P. G.: Feedback Directed Optimization in Compaq's Compilation Tools for Alpha. In *ACM Workshop on Feedback-Directed Optimization* (1999). 2

[CMH*13] Cho H. K., Moseley T., Hank R., Bruening D., Mahlke S.: Instant Profiling: Instrumentation Sampling for Profiling Datacenter Applications. In *International Symposium on Code Generation and Optimization (CGO)* (2013), pp. 1–10. 2

[Cor21] Corporation N.: *License for Customer use of GeForce Experience Software*, May 2021. URL: https://www.nvidia.com/en-us/geforce/geforce-experience/license/. 12

[D'A20] D'ANASTASIO C.: An Infrastructure Arms Race Is Fueling the Future of Gaming. *Wired* (2020). URL: https://www.wired.com/story/cloud-gaming-infrastructure-arms-race/. 2

[DR97] DINIZ P. C., RINARD M. C.: Dynamic Feedback: An Effective Technique for Adaptive Computing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (1997), Association for Computing Machinery, pp. 71–84. 2, 7

[Fre08] THE FREE SOFTWARE FOUNDATION: *PDO - GCC Wiki*, 2008. URL: https://gcc.gnu.org/wiki/PDO. 2

[FT11] FURSIN G., TEMAM O.: Collective Optimization: A Practical Collaborative Approach. *ACM Transactions on Architecture and Code Optimization (TACO) 7*, 4 (Dec. 2011). 3

[HNL*13] HOMESCU A., NEISIUS S., LARSEN P., BRUNTHALER S., FRANZ M.: Profile-Guided Automated Software Diversity. In *International Symposium on Code Generation and Optimization (CGO)* (2013), pp. 1–11. 5, 11

[JTLL10] JIN G., THAKUR A. V., LIBLIT B., LU S.: Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2010), pp. 241–255. 3

[Kar21] KARLSSON B.: *Renderdoc v1.12*, January 2021. URL: https://renderdoc.org/docs/index.html. 7

[KBR17] KESSENICH J., BALDWIN D., ROST R.: *The OpenGL® Shading Language*, May 2017. URL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf. 2

[KKOW00] KISUKI T., KNIJNENBURG P., O'BOYLE M., WIJSHOFF H.: Iterative Compilation in Program Optimization. In *Compilers for Parallel Computers* (2000), pp. 35–44. 10

[LAHC06] LAU J., ARNOLD M., HIND M., CALDER B.: Online Performance Auditing: Using Hot Optimizations without Getting Burned. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2006), pp. 239–251. 2, 7

[Lib04] LIBLIT B. R.: *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004. 2, 3, 4

[LMnJC20] LAZCANO R., MADROÑAL D., JUAREZ E., CLAUSS P.: Runtime Multi-Versioning and Specialization inside a Memoized Speculative Loop Optimizer. In *International Conference on Compiler Construction* (2020), pp. 96–107. 2, 7

[LNZ*05] LIBLIT B., NAIK M., ZHENG A. X., AIKEN A., JORDAN M. I.: Scalable Statistical Bug Isolation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2005), pp. 15–26. 3

[LP20] LEOBAS G. V., PEREIRA F. M. Q.: Semiring Optimizations: Dynamic Elision of Expressions with Identity and Absorbing Elements. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Nov. 2020). 2

[Mic18a] MICROSOFT CORPORATION: *Atomic Iadd*, May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/atomic-iadd--sm5---asm-. 5

[Mic18b] MICROSOFT CORPORATION: *High Level Shading Language*, May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl. 2

[Mic18c] MICROSOFT CORPORATION: *Shader Model 4 Assembly (DirectX HLSL) - dcl_globalFlags*, May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dcl-globalflags. 6

[Mic18d] MICROSOFT CORPORATION: *Shader Model 5 Assembly (DirectX HLSL)*, May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/shader-model-5-assembly--directx-hlsl-. 8

[Mic18e] MICROSOFT CORPORATION: *Timing*, May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/timing. 7

[Mic18f] MICROSOFT CORPORATION: *Variable Syntax*, May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-variable-syntax. 6

[Mic19a] MICROPROCESSOR STANDARDS COMMITTEE: *754-2019 - IEEE Standard for Floating-Point Arithmetic*, July 2019. URL: https://ieeexplore.ieee.org/servlet/opac?punumber=8766227. 6

[Mic19b] MICROSOFT CORPORATION: *DirectX Intermediate Language*, 2019. URL: https://github.com/Microsoft/DirectXShaderCompiler/blob/master/docs/DXIL.rst. 8

[Mic19c] MICROSOFT CORPORATION: *Profile-Guided Optimizations*, 2019. URL: https://github.com/MicrosoftDocs/cpp-docs/blob/master/docs/build/profile-guided-optimizations.md. 2

[MR95] MOTWANI R., RAGHAVAN P.: *Randomized Algorithms*. Cambridge University Press, 1995. 9

[MWD00] MUTH R., WATTERSON S. A., DEBRAY S. K.: Code Specialization Based on Value Profiles. In *International Symposium on Static Analysis (SAS)* (2000), pp. 340–359. 2

[NVI21a] NVIDIA CORPORATION: *Nsight 2021.1*, January 2021. URL: https://developer.nvidia.com/nsight-graphics. 7

[NVI21b] NVIDIA CORPORATION: *Parallel Thread Execution ISA: Application Guide*, March 2021. URL: https://docs.nvidia.com/cuda/pdf/ptx_isa_7.2.pdf. 7

[PCL11] PRADELLE B., CLAUSS P., LOECHNER V.: Adaptive Runtime Selection of Parallel Schedules in the Polytope Model. In *High Performance Computing Symposium* (2011), pp. 81–88. 2, 7

[RSU*20] RANGAN R., STEPHENSON M. W., UKARANDE A., MURTHY S., AGARWAL V., BLACKSTEIN M.: *Zeroploit*: Exploiting Zero Valued Operands in Interactive Gaming Applications. *ACM Transactions on Architecture and Code Optimization (TACO) 17*, 3 (Aug. 2020). 2, 6

[SR21] STEPHENSON M., RANGAN R.: PGZ: Automatic Zero-Value Code Specialization. In *Proceedings of the International Conference on Compiler Construction (CC)* (2021), p. 36–46. 2, 6, 7

[SRYVH10] STEPHENSON M. W., RANGAN R., YASHCHIN E., VAN HENSBERGEN E.: Statistically Regulating Program Behavior via Mainstream Computing. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (2010), p. 238–247. 3

[Ste20] Steam. https://store.steampowered.com/stats/Steam-Game-and-Player-Statistics, November 2020. Accessed: 2020-11-15. 9

[Tec18] TECHPOWERUP: *NVIDIA Geforce RTX 2080*, September 2018. URL: https://www.techpowerup.com/gpu-specs/geforce-rtx-2080.c3224. 7

[WCL17] WEN S., CHABBI M., LIU X.: REDSPY: Exploring Value Locality in Software. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017), pp. 47–61. 2

[WH12] WU D., HAVEN J.: Using Homomorphic Encryption for Large Scale Statistical Analysis. *FHE-SI-Report, Univ. Stanford, Stanford, CA, USA, Tech. Rep. TR-dwu4* (2012). 12

[YYL*20] YOU X., YANG H., LUAN Z., QIAN D., LIU X.: ZeroSpy: Exploring Software Inefficiency with Redundant Zeros. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2020). 2

[ZHMC*20] ZHOU K., HAO Y., MELLOR-CRUMMEY J., MENG X., LIU X.: GVProf: A Value Profiler for GPU-Based Clusters. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2020). 2