PGZ: Automatic Zero-Value Code Specialization

Mark Stephenson
mstephenson@nvidia.com
NVIDIA
Austin, TX, USA

Ram Rangan
rrangan@nvidia.com
NVIDIA
Bangalore, Karnataka, India

Abstract
In prior work we proposed Zeroploit, a transform that duplicates code, specializes one path assuming certain key program operands, called versioning variables, are zero, and leaves the other path unspecialized. Dynamically, depending on the versioning variable’s value, either the specialized fast path or the default slow path will execute. We evaluated Zeroploit with hand-optimized codes in that work.

In this paper, we present PGZ, a completely automated, profile-guided compiler approach for Zeroploit. Our compiler automatically determines which versioning variables or combinations thereof are profitable, and determines the code region to duplicate and specialize. PGZ’s heuristic takes operand zero value probabilities as input and it then uses classical techniques such as constant folding and dead-code elimination to estimate the potential savings of specializing a versioning variable. PGZ transforms profitable candidates, yielding an average speedup of 21.2% for targeted shader programs, and an average frame-rate speedup of 4.4% across a collection of modern gaming applications on an NVIDIA GeForce RTX™ 2080 GPU.


Keywords: profile guided optimization, value specialization, GPUs, shader programs, gaming applications

1 Introduction
Graphics processing units (GPUs) have enjoyed consistent generation-over-generation performance growth since their inception, thanks in large part to 2D CMOS scaling. However, as GPU performance growth fueled by Moore’s Law comes to an end, the gaming industry is in urgent need of novel architectural and software solutions to deliver similar performance growth to sustain innovations in our quest for photo-realistic real-time rendering.

To that end, we previously introduced Zeroploit, a template for exploiting dynamically zero valued operands in shader programs of gaming applications [27]. The Zeroploit approach uses the knowledge that one of the input operands of a multiply (or a similar) operation is mostly zero dynamically to rearrange the code to avoid computing the other source operand whenever the former is zero, since anything multiplied by a zero results in a zero. Though this is not safe as per IEEE 754 rules [6] since the other source operand could evaluate to a floating point NaN, −∞ or +∞, game developers typically allow for such IEEE-unsafe floating point optimizations 1 to opportunistically squeeze out additional performance as well as to ensure that NaN values do not leak into render targets. Zeroploit leverages this developer-granted permission to transform a code region into a zero-specialized fast path and an unspecialized slow path. We previously applied Zeroploit manually to graphics shader programs, both in terms of opportunity detection as well as the code-versioning transform itself. In this paper, we strive for an automatic compiler solution for the Zeroploit optimization for shader programs of gaming applications.

Figure 1 illustrates Zeroploit. It shows the two basic requirements of the Zeroploit transform, namely, the identification of a versioning variable and the identification of a versioning scope. In its basic form, a versioning variable is an operand that a value profiler identifies as having a high probability of being zero dynamically (e.g., the output of a saturating arithmetic operation). The versioning scope is the region of the input code that gets duplicated to create the specialized fast path and unspecialized slow path code versions.

An automatic compiler solution to Zeroploit must solve three key challenges. First, given that shader programs typically have fat and complex data flow graphs (DFGs) and the probability of their operands being dynamically zero

1 In Microsoft’s high-level shading language (HLSL), this is achieved by explicitly setting the refactoringAllowed[10] global flag and dropping the precise storage class specifier for variable declarations [11].
We present techniques to effectively address challenges arising from mixed vector-scalar ISAs. PGZ's transform relies on SIMT-wide vote [13] or an equivalent operation to ensure that versioning branches are dynamically convergent. We describe the PGZ compiler pipeline in detail and show how the above solutions fit together within this framework.

We evaluate PGZ on a suite of modern, heavily-optimized gaming applications and show that the automatic approach presented in this paper can provide an average speedup of 21.2% for targeted shaders on an NVIDIA GeForce RTX™ 2080 GPU, which translates to an average frame-rate speedup of 4.4%. We show that PGZ’s compile-time overhead is suitable for just-in-time environments such as GPU drivers generating code for interactive gaming applications.

2 Background

This section provides a high-level introduction to three topics: graphics programs, the execution model in NVIDIA GPUs, and profile-guided optimizations.

2.1 Graphics Programs

Modern gaming applications are most commonly developed in Direct3D 11 [8], Direct3D 12 [9], OpenGL [16], and Vulkan [17] APIs. Without exception, all graphics APIs use a two-level hierarchy to convey work from the CPU to the GPU: an API layer and a shader program layer. The API layer is used to setup GPU state, bind resources such as constant buffers, textures (input data), render targets (output data), or other read-write resources, and bind shader programs for use in GPU work calls. A GPU work call typically causes one or more threads of a corresponding bound shader program to be run.

A typical API sequence interleaves various state setup commands with GPU work calls. Several such commands can be in flight at any given point in the GPU. State setup commands take very little time and most of the GPU time is spent in GPU work calls. The hundreds or thousands of API calls that make up a frame can exhibit vastly different performance characteristics. An API call can be limited by the performance of fixed function units, CPU-GPU data transfer latencies, or the performance of programmable shaders, whose performance can in turn be limited by memory bandwidth, instruction issue rate, or latency of memory loads. PGZ is applicable only to the portion of a gaming application’s frame time that is dominated by programmable shaders, specifically pixel and compute shaders, which are the two most expensive shader types.

2.2 Execution Model in NVIDIA GPUs

Using NVIDIA’s terminology, shader programs execute on programmable processing cores called streaming multiprocessors (SMs). Threads of shader programs are grouped into convenient physical entities called warps. A warp can span at most 32 threads. Instruction fetch, decode, and scheduling in the SM happens at a warp granularity, while operand fetch, execution in functional units, and register writeback
happen at a per-thread granularity. SM execution is most efficient when handling full warps, since all its functional units are fully utilized. The SM implements a single-instruction, multiple threads (SIMT) execution model [13] whereby each thread of a warp maintains its own program counter (PC). If threads of a warp diverge, i.e., execute at different PCs, SM datapath utilization suffers. To mitigate divergence the SM uses joint hardware-software mechanisms to make diverged threads re-converge at well-defined “synchronization” points in the program. We will build on the above background when discussing PGZ’s heuristics later in the paper.

2.3 Profile-Guided Optimizations
Profile-guided optimizations (PGOs) have historically focused on collecting control flow profiles to determine the execution weights of various blocks of code. Compiler passes such as inlining, register allocation, predication, and loop unrolling make use of accurate information about execution weights to generate suitably optimized code. For example, a compiler can arrange code based on profile feedback to improve instruction cache performance by packing frequently taken blocks closer to each other. Such execution weight based PGOs have proven effective in a variety of programs, from general purpose programs [5] to Web browsers [3].

PGOs typically involve compiling and executing code in two different ways. First, the baseline code needs to be instrumented for profile collection and profiles collected. Next, a second compilation may then use the collected profile information to more intelligently generate code. At a high level, the PGZ optimization presented in this paper uses a similar dual compilation approach. PGZ differs from typical PGOs in its use of value profiles, instead of control flow profiles, to optimize code. Section 6 provides a comparison against prior value specialization work.

3 Zeroploit
We provide background information on Zeroploit in this section. Before doing so, we recall relevant terminology from the Zeroploit paper [27]. A zero-propagator is an instruction that produces zero as output due to one or more of its source operands being zero (e.g., a multiply operation), while a zero-originator is an instruction that produces zero as output from non-zero inputs, due to its semantics (e.g., a saturating arithmetic operation). A useful zero-originator is a zero-originator whose output is consumed by a zero-propagator.

A typical Zeroploit opportunity has the following three main ingredients:

1. A program variable produced by a useful zero-originator and by definition, consumed by a zero-propagator. This variable (or its equivalent) will become the versioning variable in the transformed code, based on whose dynamic value, execution will jump to either a fast path or a slow path.
2. The other operand of the zero-propagator is expensive to compute.
3. The expression computing the versioning variable is available early. At a minimum, this expression must be independent of the other operand. In general, if the backward slices of the versioning variable and the other operand intersect farther up a shader’s program dependency graph (or do not intersect at all), the better it is for specialization as that allows for larger versioning scopes.

We refer to the example in Figure 1 as we describe the steps of the Zeroploit transform below. The code in Figure 1a, which is assumed to satisfy all three prerequisites, can be Zeroploit-transformed to the code in Figure 1b as follows. First, Zeroploit identifies a set of operations that will be affected by either forward or backward slice (including recursive backward slice) specialization based on the versioning variable being zero. The region of code from the first affected operation to the last forms the versioning scope. Next, it duplicates operations in the versioning scope. One of the copies or versions, which will become the specialized fast path, is prefixed with an explicit assignment of zero to the versioning variable to enable the compiler back-end to easily notice the specialization opportunity in that scope and apply classical optimizations such as constant folding, constant propagation, and dead code elimination to instructions in that path. The other version will serve as the default, unspecialized, slow path. Finally, a conditional branch predicated on the versioning variable being equal to zero is added to dynamically steer execution to the fast path or the slow path. Figure 1c shows the final optimized code after the fast path has been specialized by a compiler back-end.

An automatic Zeroploit solution for gaming applications running on SIMT GPUs must be able to evaluate, rank, and choose from among several potential versioning variables, deal with challenges posed by mixed scalar-vector ISAs when determining versioning variables and scopes, and ensure SIMT warp divergence penalties do not negate specialization benefits. The next section describes our solution to the challenges above.

4 PGZ
PGZ automatically transforms shader programs to profitably exploit likely zero operands. This section discusses how our compiler identifies candidate versioning variables, our zero-value profiler, and the compiler analyses and transformations PGZ performs. In the remainder of the paper when we state that the compiler specializes on a versioning variable, we implicitly mean that the compiler specializes a region of code in which the variable is guaranteed to be zero, and it guards that region with a runtime check that invokes either the specialized region, or the default fallback region. Without loss of generality we assume the compiler’s intermediate representation (IR) is in static single assignment (SSA) form.
Figure 2. PGZ flow. PGZ runs early in our prototype’s compiler flow as (a) indicates, before many of our compiler’s significant passes run. As (b) shows, after enumerating the potential candidates, PGZ runs in one of two modes: instrumentation, or PGO.

Figure 2(a) shows where PGZ runs with respect to other compiler phases. Our profiling pass, as well as the PGZ analyses and transformations run early in our backend compiler’s flow, and as a result the IR that PGZ sees is unoptimized. However, by running PGZ after classical optimizations like dead code elimination and copy propagation, we ensure PGZ operates on code without any static redundancy, which reduces the number of potential operands it needs to profile and evaluate. We choose to run PGZ earlier than advanced backend optimizations because loop optimizations such as unrolling and software pipelining can increase the number of potential versioning variables our optimization has to consider. As with most compiler optimizations we recognize that PGZ is sensitive to phase orders. Future work will consider alternate phase orders.

4.1 Identifying Candidate Versioning Variables
Regardless of PGZ’s ordering with respect to other compiler passes, PGZ’s first task is to enumerate the set of possible versioning variables in a shader, which we henceforth refer to as candidates. We call them candidates because at this point in PGZ’s flow the compiler does not know the potential upside to specialization on any of the versioning variables. The candidate enumeration step of Figure 2(b) proceeds as follows:

1. Find scalar candidates: PGZ makes the distinction between scalar candidates and vector candidates. Scalar candidates are trivial to enumerate since they are merely the virtual register writes in a program. To reduce the optimization search space PGZ does not consider as candidates constant literal moves because traditional compiler transformations such as constant propagation and folding will likely optimize these away; nor does PGZ consider register-to-register moves as candidates because they simply rename the move’s source operand and are therefore redundant. All other register writes are potential scalar candidates. In SSA form there is a one-to-one correspondence between candidates and virtual registers.

2. Find vector candidates: In addition to scalar candidates, PGZ groups subsets of scalar candidates together to form vector candidates. Some of PGZ’s biggest gains come from jointly specializing multiple scalar candidates in the IR, which is also supported by prior observations [27]. We have seen cases where jointly specializing on the conjunction of two or more scalar candidates exposes specialization opportunities, whereas specializing on any scalar candidate in isolation exposes none. Blindly considering arbitrary combinations of candidates is combinatorial in the number of versioning variables and is therefore not tractable. To gain insights into what combinations PGZ should consider we manually inspected dozens of shader programs and found that many specializable combinations of variables are explicitly combined by the programmer into short vectors, typically into RGBA or XYZW components. Manual inspection of shader program code led to the insight that some specialization opportunities require whole vectors to be zero. So while PGZ does not consider packing arbitrary combinations of candidates, we found that packing candidates that are elements of a programmer-specified vector into a vector candidate is sufficient to expose additional specialization opportunities.

A challenge we face when attempting to pack candidates into their programmer-specified vectors is that short vector information does not propagate from the programming language to the compiler’s backend IR. In fact, for DirectX 12, the explicit vectors in the high level shader language (HLSL) are lost before backend compilation begins. To reconstruct programmer-specified vectors we leverage prior art on compiling for short, single instruction multiple data (SIMD) instruction sets. In particular, for this work we implemented portions of Larsen and Amarasinghe’s “SLP” algorithm for packing independent and isomorphic instructions into SIMD superwords [21]. We note that for our purposes since we are not actually packing candidates into wide registers nor scheduling instructions to execute in SIMD fashion, a best-effort implementation that forms natural groups of candidates suffices. PGZ creates a new set of vector candidates from the superword-level vectors formed by applying SLP [21].

3. Enumerate candidates: PGZ enumerates the union of scalar candidates (S) and vector candidates (V). That is, PGZ assigns each candidate with an identifier that it uses to associate candidates with runtime profiles.

Formalizing our discussion of candidates, each scalar candidate $s \in S$ corresponds to exactly one SSA virtual register definition $or_s$; and a virtual register $or$ associates with at most...
one scalar candidate, $s$. Virtual registers can also correspond to a vector candidate $v \in V$ and a scalar candidate $s \in S$. Each vector candidate represents a set of virtual registers, e.g., $v_r = \{v_{rx}, v_{ry}, v_{rz}, v_{rw}\}$ where $x, y, z, w \in S$ and are distinct. We limit the number of virtual registers associated with a vector candidate to four.

As Figure 2(b) shows, after candidate enumeration PGZ can take on one of two distinct paths. On one path PGZ instruments the shader to collect runtime profiles, and on the other path PGZ ingests a runtime profile and attempts to specialize the shader for likely zeros. We first discuss program instrumentation for profiling.

### 4.2 Instrumenting for Profiling

For each candidate enumerated above, our profiler records the likelihood that the candidate is zero at runtime. We presented a general value profiling framework in [27] that builds upon the ideas presented in [1]. However, because we did not see significant opportunities to leverage values other than zero, instead of reusing our general value profiler, we designed a simple zero-value profiler that discovers the likelihood that a candidate is zero. Our profiler instruments each scalar candidate $s \in S$ to record both the total number of writes, $writes_s$, to the candidate’s associated virtual register $v_r$ and the number of those writes that were zero, $zero_s$. Our zero-value profiler leverages NVIDIA’s global memory atomic increment operations for updating counters.

For each scalar candidate $s$, we can easily estimate the likelihood, $P(v_r = 0)$, that the associated versioning variable is zero according to $P(v_r = 0) \approx \frac{\text{zero}_s}{\text{writes}_s}$. Vector candidates, on the other hand require a different approach. Ideally we would estimate the likelihood that all elements of the vector are jointly zero, e.g., $P(v_r = 0) = P(v_{rx} = v_{ry} = v_{rz} = v_{rw} = 0)$. A profiler could determine the joint likelihood by only incrementing $writes_s$ after all elements of the vector are written, and by only incrementing $zero_s$ if all elements of the vector are zero. Alternatively, and the approach we use in our prototype, is to estimate $P(v_r = 0)$ assuming independence among the individual candidates, i.e.,

$$P(v_{rx} = 0) \cdot P(v_{ry} = 0) \cdot P(v_{rz} = 0) \cdot P(v_{rw} = 0)$$

The final profiling decision we discuss relates specifically to SIMT execution. If the versioning variable is not uniformly zero across the threads of a warp, should we still consider specializing code on that variable? PGZ only considers warp-uniform zero specialization primarily because it is the conservative choice. One of the primary metrics GPU programmers use to gauge the performance of their code is SIMT efficiency, i.e., how many threads in a warp are simultaneously active on average. Were PGZ to introduce divergent branches, where the guarding runtime specialization check causes the warp to serially execute the specialized region and the fallback region, it would decrease SIMT efficiency and quite likely reduce performance. For this reason, PGZ’s profiler uses the `vote.all` flavor of NVIDIA’s `vote` [13] instruction to ensure that $v_r$ is zero for all threads in the warp before incrementing candidate $c$’s zero counter, $zero_c$.

### 4.3 Profile-Guided Zero Specialization

We now discuss the steps our prototype performs to specialize a shader. As shown in Figure 2(b), when the driver supplies the backend compiler with a valid runtime profile, PGZ attempts to specialize a shader to exploit likely zeros. The basic flow of our profile-guided specialization is to iteratively estimate each candidate’s performance potential and then transform the best candidate if it is above a given threshold. We now describe each step of this flow.

#### 4.3.1 Filtering Candidates

As we soon discuss, each step of the estimate-transform loop shown in Figure 2(b) is expensive. We can significantly improve the compile-time overhead by filtering out candidates where $P(v_r = 0) < P_{\text{thresh}}$. In Section 5 we show how $P_{\text{thresh}}$ affects compile time, but for all other results we present, $P_{\text{thresh}} = 0.32$. For vector candidates, we remove any element from the candidate if its zero likelihood is less than $P_{\text{thresh}}$. We can also skip the evaluation of any zero-propagator candidates $zp$ with $P(v_{zp} = 0)$ if one of its zero-propagating inputs is generated by a candidate $c$ with $P(v_{rc} = 0) = 0$. This follows because in this scenario, specializing for candidate $c$ will also specialize candidate $zp$. Eliminating zero-propagator candidates in this fashion through a transitivity analysis helps whittle down the list of candidates further to just root zero-originators.

#### 4.3.2 Evaluating Candidates

The PGZ transformation adds control flow, which not only bloats code, but also has the potential to negatively impact scheduling by splitting independent long latency instructions across basic blocks and artificially serializing them. Therefore, we need to balance the deleterious effects of PGZ’s code transformation with its potential upside. With a set of candidates to consider, PGZ estimates the benefit of each candidate. Our algorithm greedily selects the candidate with the best estimated opportunity, and transforms the shader to specialize for the candidate’s associated variable(s). While the actual specialization transformation is straightforward, estimating a candidate’s benefits quickly is more difficult. Note that shaders commonly have hundreds of candidates, so we must limit the compile-time complexity of opportunity estimation.

Figure 3 illustrates PGZ’s approach on a simple example. At a high-level PGZ estimates benefits by first finding the set of instructions that become specialized constants in the

---

1. It is still beneficial to introduce divergent branches when the shader is limited by texture throughput and specialization removes texture operations [27]. However, such scenarios are difficult for tools to identify without hardware performance monitor feedback.

2. The `vote.all` instruction returns `True` if a Boolean predicate is `True` across all active threads of a SIMT warp and `False` otherwise.
forward slice. In the figure, which shows the evaluation of variable \( r_0 \), constant propagation causes all subsequent instructions to evaluate to zero. PGZ then runs a dead code elimination analysis to remove the backward slices of computation that are no longer required (shown with dotted lines in the figure). While constant propagation and dead code elimination form PGZ’s foundation, candidate estimation performs these supporting analyses:

1. **Versioning check hoisting**: This step aims to determine legal, “earlier” locations in the program to which to hoist the computation needed to perform the candidate’s versioning check. Hoisting the versioning check can increase the scope of the computation’s backward slices. While we could have implemented a sophisticated code motion approach (e.g., loop invariant code motion, partial redundancy elimination, or unification), our prototype performs simple intra-block hoisting of the candidate’s versioning check. This limitation trades off missing some specialization opportunities for improved compile-time speed.

2. **Region identification**: This step identifies the set of basic blocks, \( R \), in which any specialization is legal for a given candidate, and it depends on the location of the versioning check. The set is formally defined as \( \{ b \mid b \in G \text{ and } v \in \text{Dom}(b) \} \), where \( G \) represents the set of basic blocks in the control flow graph, \( v \) is the block with the versioning check, and \( \text{Dom} \) is the function that returns the dominators of the given block. We can copy the blocks to a new set \( S \leftarrow R \), and insert an if-then-else hammock that branches either to \( R \) or \( S \), depending on the outcome of the versioning check.\(^5\) We will specialize the instructions in \( S \).

3. **Constant propagation**: Within \( S \) we perform a standard forward constant propagation analysis. When propagating constants, if the definition of a source operand reaches from a block \( b \notin S \) we conservatively assume the operand is not constant (i.e., it is \( \top \)). This analysis discovers the set of instructions for which all destination results will provably return constants.

\(^5\)We perform all of the analyses for opportunity estimation without actually transforming the shader. This improves the efficiency of our optimization.

4. **Dead code elimination**: Source registers are not needed to produce a statically known constant, and therefore newly discovered constant expressions in \( S \) might eliminate additional expressions in the backward slice. The following dataflow equations, which are slight modifications to live variable analysis, show how the backward analysis proceeds at the instruction level.

\[
\begin{align*}
\text{Kill}_i &= \text{Defs}(i) \\
\text{Gen}_i &= \begin{cases} 
\emptyset & \forall d \in \text{Defs}(i) \text{ or } d \text{ is constant} \\
\text{Uses}(i) & \text{otherwise} 
\end{cases} \\
\end{align*}
\]

If all of an instruction’s destination registers (i.e., the LHS of the instruction) are marked as constants then Equation 2 does not generate any upward exposed flows. Furthermore, if an instruction’s definition is not live at the point of definition, then we can effectively remove the instruction and forgo its upward exposed flows. Of course, this only applies to instructions that do not change global state such as branches and stores. In addition, we logically treat fused-multiply-add instructions as a multiply followed by an add, which allows us to remove backward slices of the “other” multiplicand, when one of the multiplicands is zero.

At the block level, Equation 3 is exactly the same as it is for live variable analysis. However, we slightly modify the live variable analysis flow in Equation 4 so that we need only consider computing flows in the region \( S \). This relies on a pre-computed set of “live in” virtual registers for the original shader, \( LIn \).

\[
\begin{align*}
\text{In}_n &= (\text{Out}_n - \text{Kill}_n) \cup \text{Gen}_n \\
\text{Out}_n &= \bigcup_{s \in \text{succ}(n)} \begin{cases} 
\text{In}_s & s \in S \\
\text{LIn}_s & s \notin S 
\end{cases} 
\end{align*}
\]

5. **Region refinement**: Remember that the versioning variable’s basic block dominates \( S \). However, constant propagation and dead code elimination may have found specialization opportunities in only a subset of \( S \)’s basic blocks. This optional step attempts to limit the scope of the versioning variable to reduce the number of new basic blocks specialization introduces.

**Heuristic**: After we have run the analyses mentioned above we estimate the candidate’s opportunity. The heuristic we use to determine a candidate’s effectiveness follows:

\[
T_{PGZ} = P(\text{sr} = 0) \cdot T_S + (1 - P(\text{sr} = 0)) \cdot T_R + T_{\text{check}} \\
T_S = T_{PGZ} - T_R
\]

\(T_{\text{check}}\) is an estimate for the number of cycles spent performing the versioning check, \( T_R \) estimates how many dynamic cycles are spent in \( R \), and is based on the basic block frequencies (from the profile) as well as the mixture of instructions in \( R \). Likewise, \( T_S \) is a cycle estimate using basic block frequency information as well as the set of instructions remaining in \( S \). Intuitively, \( T_S \) is the number of cycles...
saved via specialization. Initially we transformed any candidate if $T_\delta$ exceeded 25 cycles, but that approach failed for a couple of reasons. First, it did not take into account the speedup of specializing the candidate. For example, if $R$ is large and expensive to execute, then saving 25 cycles is probably not worth the negative effects of PGZ’s transformation. In addition, we want to forgo transforming candidates that introduce too many additional basic blocks. Finally, we give an extra boost to candidates where specialization removes memory operations:

$$su = \frac{T_R - (T_S + T_{\text{check}})}{T_R}$$

$$su_{\text{thresh}} = \frac{P(R = 0) \cdot \text{MemKilled}}{|S|},$$

where $|S|$ is the number of blocks in $S$, and MemKilled is the number of memory operations in $R$ that are not in $S$. We will not transform any candidate unless $T_\delta > 25$ and $su < su_{\text{thresh}}$. These refinements to our heuristic avoid transforming some empirically negative candidates. When discussing results in Section 5, we show that this heuristic effectively identifies several outstanding opportunities, but we also show that there are still cases where our heuristic fails. Future work will consider alternative heuristics.

4.3.3 Performing the Transformation. Figure 4 demonstrates the PGZ transformation with a simple example. In the example, we specialized the CFG in Figure 4a for executions where R8 is zero. While the transformation is conceptually simple, this example lets us recap some interesting trade-offs. As during the evaluation phase, we determine the scope of the candidate’s versioning variable. In our example, the grayed basic block is the candidate’s scope, since this block is the only block in the CFG that the candidate dominates. Our transformation creates the region $S$ (the grayed block in Figure 4b), and sets all occurrences of R8 in $S$ to zero. It inserts a versioning check that either branches to $S$ or the original region $R$. PGZ passes the versioning check predicate through the vote.all operation [13], thus making the dependent branch warp-convergent. This avoids SIMT warp-divergence penalties in the transformed code. Finally, PGZ invokes constant folding and dead code elimination.

This example makes clear that the PGZ transformation increases the size and complexity of a shader’s CFG. In addition, the runtime versioning check introduces overheads. In this example, depending on $P(R = 0)$, the added complexity may be worthwhile because specialization removed an expensive texture lookup (TEX). Note however, that specialization also delays the execution of the “TEX R0 . . .” instruction. In the original CFG in (a), the two texture operations could potentially execute concurrently, whereas specialization defers the execution of a texture operation when $R8 \neq 0$.

Figure 4a also demonstrates different scoping alternatives. The Zeroploit approach, which we remind the reader is not automated, hoists the versioning variable as “early” as possible to expose additional opportunities. In this example Zeroploit would hoist the versioning variable to the grayed block’s dominator, where the subsequent analysis would determine that R2 is zero everywhere when R8 is zero.

4.3.4 Managing Compile-Time. Our prototype does not consider inter-block code motion to increase a candidate’s scope, and we made this decision primarily in the name of...
Table 1. Gaming applications evaluated in this paper.

<table>
<thead>
<tr>
<th>Application</th>
<th>Short</th>
<th>APICs</th>
<th>dx11</th>
<th>dx12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ashes of the Singularity - Escalation</td>
<td>Ashes</td>
<td>3</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deus Ex Mankind Divided</td>
<td>DXMD</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Final Fantasy XV</td>
<td>FFXV</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Metro Exodus</td>
<td>Metro</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PlayerUnknown’s Battlegrounds</td>
<td>PUBG</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Horizon Zero Dawn</td>
<td>HZD</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WatchDogs Legion</td>
<td>Watch</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Serious Sam Legion</td>
<td>SS</td>
<td>3</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Partial redundancy elimination and related techniques are expensive enough that some compiler engineers are reluctant to apply them once [20], let alone potentially hundreds of times, once per candidate.

Later, in Section 5, we show the correlation between compile-time and the number of candidates PGZ considers and the number of transforms it performs (up to xform_limit in Figure 2). We can reduce compilation time by better candidate filtering, including increasing the "probability of zero" threshold for a candidate.

5 Evaluation

This section describes our experimental methodology, presents runtime speedups, compile-time overheads, and concludes with a discussion of insights regarding the results.

5.1 Methodology

We implemented PGZ as a research prototype on top of a recent branch of NVIDIA’s GeForce Game Ready driver. Value profiling is performed in offline runs, while just-in-time compilation based on these value profiles happens online.

Table 1 lists the gaming applications evaluated in this paper. Using an internal frame-capture tool, similar to publicly available tools like Renderdoc [19] and Nsight [12], one or more random single-frames are captured from a built-in benchmark run 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. both the API sequence (including all relevant state) as well as shader programs used in individual calls. For applications for which we use more than one APIC, we made sure they were from visually different scenes. We evaluate PGZ on a total of 14 APICs spanning 8 applications as shown in Table 1. We hyphenate an application’s short name with the APIC number to uniquely refer to an APIC in this section.

We prefer APIC replays to full-game runs as the former do not require user interaction, allow for fine-grained timing of targeted calls, and enable deep dives into targeted shaders, all of which are important for evaluating a research prototype.

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 [26]. We compare the final rendered image against a golden reference image to ensure that our transforms are functionally correct. We measure GPU-only time using accurate in-house profiling tools, after locking GPU clocks and power state to production active game-play settings. These runs use production settings for driver and compiler optimizations, which include classical optimizations like constant folding, dead code elimination, loop unrolling, instruction scheduling, etc., in addition to various machine-specific optimizations.

5.2 Shader Program Performance

PGZ’s speedups over baseline execution at the targeted region and full-frame levels are shown in Figure 5a and Figure 5b, respectively. We define a “targeted region” of an application as the set of shader programs that PGZ’s heuristic chooses for transformation. We calculate targeted region speedup as the sum of the baseline time for all chosen shaders divided by the sum of the execution times for the same shaders in a PGZ-transformed run. Both graphs show performance speedup of PGZ without SLP, with default PGZ, and PGZ with an “oracle” heuristic. The latter is calculated by simply ignoring all heuristic-induced slowdowns. These results demonstrate that SLP, which we use to form vector candidates, is very useful in APICs from PUBG and SS. Overall, PGZ’s heuristics perform almost as good as the oracle.

Next, we compare PGZ to manual Zeroploit. We obtained manual Zeroploit-ed shaders (reused from [27] or created anew) for all applications except HZD, which was cumbersome to hand-optimize. Since PGZ targets more shaders per APIC than Zeroploit, we do not compare targeted regions and
compare only full-frame performance in Figure 5b. We see that for most applications PGZ’s performance very closely tracks that of manual Zeroploit. For PUBG, PGZ is able to successfully exploit Zeroploit opportunities in more shaders than was possible manually. In SS, the manual optimization outperforms PGZ by combining versioning variables cleverly using integer comparison operations, a trick that we have not yet incorporated in the automatic algorithm since we found this to be a one-off occurrence. Overall, despite imposing a heavy programming cost, manual Zeroploit shows an average frame speedup of 5.5%, while PGZ achieves most of that speedup automatically.

Since our goal in this paper is to demonstrate an automatic compiler technique for Zeroploit and not to explore its effectiveness across scenes of gaming applications, we use the same APICs for profiling and testing. However, in the few applications for which we were able to obtain multiple APICs corresponding to different scenes, we can see that PGZ is able to extract Zeroploit benefit effectively across scenes. For example, in Figure 5a, notice similar upsides among PUBG APICs, among Ashes APICs, and in APICs 2 and 3 of SS. This augurs well for a dynamic adaptive compilation system based on PGZ, which we plan to explore in the future.

At the full-frame level, we see average PGZ speedups of 4.4% and 4.5% with default and oracle heuristics, while for targeted regions PGZ achieves average speedups of 21.2% and 22.2%, with default and oracle heuristics, respectively. These graphs show that the default PGZ heuristics effectively target Zeroploit opportunities in most applications.

5.3 Compilation Statistics

Although PGZ is reserved for compiling hot shaders, it does run in the context of a just-in-time compiler where compilation time is a consideration. To determine the overhead of our compiler analyses and transformations we collected the compile-time slowdown over the driver’s baseline compilation for each shader that PGZ managed to transform. We focus on the transformed shaders to guarantee that the evaluate-transform loop executes at least once. We use a proprietary tool to measure and record the Direct3D driver’s per-shader compile times.

On a 3.4GHz Intel Core i7-6800K CPU with 32 GB of RAM, the geometric mean compile-time slowdown was 1.57× across all of our applications. If we increase the “probability of zero” threshold \( P_{\text{thresh}} \) for candidates that PGZ considers from 0.32 to 0.9, the mean slowdown decreases slightly to 1.45×. If, on the other hand, we limit the number of iterations of the evaluate-transform loop to one, we see the mean slowdown decrease to 1.35×. Not surprisingly, these results show a clear correlation between the number of candidates evaluated and the resultant overhead. Of note, modern drivers carefully choose when to asynchronously compile optimized shaders and will not waste CPU cycles compiling additional shaders unless performance is limited by the GPU. In such cases, where the CPU is not limiting performance, spending additional time optimizing shaders can be prudent.

5.4 Discussion

5.4.1 Heuristic Performance. Figure 6 shows how PGZ’s heuristics performed at the individual shader level. Here, we plot individual shader speedups on the y-axis against both an unweighted abscissa (blue line) and a weighted abscissa (magenta points). The unweighted abscissa is simply a monotonically increasing shader identifier, whereas the weighted abscissa is calculated based on a given shader’s normalized frame-time contribution in its baseline execution. We can see that PGZ is positive or neutral in most shaders. On about 20% of the shaders, it produces slowdowns as can be seen on the right end of blue line. However, as the steep fall of the cluster of magenta points at the right end shows, these slowdowns predominantly occur in shaders with negligible frame-time contributions. This implies PGZ’s heuristics work well in practice on shader programs that matter.

PGZ currently runs early in the compiler flow, before loop unrolling, scheduling, and register allocation. Through inspection of a few of the negative examples in Figure 6 we found that PGZ’s code-versioning transform at times interacts pathologically with these downstream passes. In particular, PGZ’s code expansion often affects the number of registers a shader uses, which can severely impact a shader’s occupancy and performance [25]. In addition, we have seen examples where PGZ’s code expansion inhibits unrolling otherwise amenable loops. Finally, by splitting basic blocks to insert versioning checks, PGZ’s transformation can also interfere with the compiler’s code motion optimizations that hoist or sink memory requests.

One noteworthy pathological interaction with the compiler’s code motion optimizations comes from a regressing shader in PUBG. PGZ does not account for the early-exit behavior of warp-uniform discard operations [23] and hoists a few texture operations above a warp-uniform discard operation that was taken 89% of the time. This causes additional texture traffic than the baseline code, resulting in a slowdown. Many of the instructions that PGZ assumed it eliminated had
in fact been dynamically discarded. One solution to remedy this issue, which future work will consider, is to profile for the likelihood of a discard and to use that information to more accurately compute the dynamic execution estimates used in Equations 5 and 6.

5.4.2 Effectiveness of PGZ in PUBG. PGZ progressively optimizes an important shader in PUBG over three iterations of the evaluate-transform loop. In the first iteration, the best candidate removes 30 multiplies and several multiply accumulate and add instructions. The candidates that PGZ transforms on the second and third iteration score poorly on the first iteration; the first transformation exposes opportunities that did not exist. On the second iteration a vector candidate removes dozens of additional math operations. Again, the final and most influential candidate that is transformed scores poorly on the second iteration. On the final iteration of the evaluate-transform loop PGZ removes a large swath of instructions, including dozens of memory operations.

6 Related Work

We previously showed Zeroploit, a profile-guided code transform for forward and backward slice specialization based on zero values, to be effective in gaming applications with manually optimized codes [27]. In this paper, we highlight the various challenges that need to be solved in order to automate Zeroploit, and present and evaluate a full compiler solution for it.

Value-dependent code specialization has been previously automated in partial evaluation systems for functional and imperative language programs [7, 18], general purpose programs [2, 15, 24, 28], embedded software [4], Java just-in-time compilers [14, 29], etc. Unlike aforementioned value-specialization efforts, which specialize for generic values and thus perform only forward-slice specialization, PGZ achieves both forward and backward-slice specialization by specializing for just zeros. For example, Muth et al.’s code-versioning approach for generic value specialization used heuristics that estimate savings from specializing forward slices [24]. Grant et al. used an annotations-based system without heuristics to target dynamic zero candidates at function scopes in the DyC dynamic compilation system [15]. In contrast, PGZ estimates forward and backward slice specialization benefits by using expected savings from constant propagation and folding as well as dead-code elimination.

Recently, Leobas and Pereira identified the type of opportunity targeted by Zeroploit as the mathematical notion of semirings. As a proof-of-concept semiring optimization, they automated a profile-driven transform to target silent stores and evaluated it on CPU programs in the LLVM test suite [22]. They also presented a novel online profiler that works well for detecting silent stores in hot loops. PGZ differs from semiring optimization in a few key ways. First, PGZ’s focus on gaming applications means it can take advantage of developer-granted permission to perform IEEE-754-unsafe operations to optimize floating point code. Second, PGZ uses a cost-benefit analysis based on constant folding and dead code elimination to evaluate, rank, and select from among several versioning variables per shader program, whereas the semiring silent store optimization did not have to use any heuristics and was applied to all stores detected by their profiler as being likely silent. Third, their loop-iteration sampling based profiler cannot be directly applied to graphics programs, since the majority of shader programs do not have loops, and the ones that do have low-tripcount loops. Therefore, PGZ uses a profiling strategy more suited to gaming applications - instruction granular zero-value profiling, sampled over several frames. And fourth, we describe the PGZ compiler pipeline in full detail, complete with dataflow equations, which complements Leobas and Pereira’s theoretical proofs for semiring optimizations, thus enabling provably correct practical application of Zeroploit and similar semiring optimizations.

Finally, unlike prior work, PGZ, when applied to gaming applications, has to contend with the SIMT execution model of NVIDIA and similar GPUs. PGZ accomplishes this by using NVIDIA’s warp-wide vote instruction [13] in both its zero-value-profile as well as its code transformer.

7 Conclusion

We previously proposed Zeroploit, a code-versioning transform that leverages likely-zero operands in shader programs of gaming applications to opportunistically execute zero-specialized code dynamically, and evaluated it with hand-optimized codes. In this paper, we advanced the above line of work with PGZ, which uses novel compile-time heuristics to automatically identify opportunities and transform code for Zeroploit. PGZ’s performance based on single-frame testing is encouraging and compares favorably with our prior labor-intensive, manual Zeroploit approach. With its default heuristics, PGZ achieves an average frame-rate improvement of 4.4% and an average targeted region speedup of 21.2% over a heavily optimized production driver, across a suite of modern gaming applications.

Future work will study PGZ’s effectiveness, both as a PGO based on offline profiling as well as an adaptive compilation strategy based on continuous online profiling, in multi-frame scenarios such as demos and game-play.

Acknowledgments

We thank the anonymous reviewers and Emmett Kilgariff for their useful feedback. Many thanks to Aditya Ukarande, Deep Shekhar, and Shyam Murthy for the manual Zeroploit-ed shaders. Finally, we thank NVIDIA’s Devtech, QA, and DX driver performance teams for providing the application traces used in this research.
References


