Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders

Georgios Keramidas, Chrysa Kokkala, Iakovos Stamoulis

Think Silicon Ltd.

Patras, GR26500, Greece

***Abstract*—Redundancy lies at the heart of graphical applications. However, as we demonstrate in this work harvesting this high degree of redundancy is not an easy task. Our experimental findings reveal that simply memoizing the outcomes of single instructions or a set of instructions is not able to pay off due to the high-precision calculations required in modern graphics APIs (e.g., OpenGL|ES 3.0). To this end, we propose clumsy value cache (VC), a hardware memoization mechanism that is explicitly managed by special machine-level instructions (part of the target GPU ISA). A unique characteristic of VC is that it is able to perform partial matches i.e., reducing the arithmetic precision (accuracy) of the input parameters, thus increasing significantly the volume of successful value reuses. To eliminate the error introduced by partial matches and consequently the impact on the quality of the rendered images, i) we systematically examine the precision tolerance of a large set of instructions in modern OpenGL fragment shaders and ii) we devise and optimize various run-time, feedback-directed policies to control the interplay between accuracy and image quality maximizing the value reuse benefits at the same time. The proposed mechanism is evaluated in a cycle-accurate OpenGL simulator and our results indicate that our approach reduces the number of executed instructions by 13.5% with negligible (non-visible) impact in the quality of the rendered images.**

I. Introduction

Exploiting the property of redundancy in graphical applications has attracted the attention of the architectural community in the recent years. The Transaction Elimination technique [2] compares consecutive framebuffer instances and performs partial updates of entire frame tiles. Parallel Frame Rendering processes multiple frames in parallel to overlap the texture accesses [3] and avoid redundant computations [4] in GPUs. Those works try to exploit inter-frame redundancy, they require significant hardware support, they can be applied only in GPUs with tile-based deferred rendering, and they reduce the responsiveness of the system.

In this work, we outline our proposal namely *Value Cache* (VC). VC is a variable precision memoization mechanism that focuses on eliminating the intra-frame redundant arithmetic calculations. In general, software or hardware memoization is a widely studied technique. The idea is to store the results of previous (costly) calculations in a dedicated storage area in order to avoid re-calculations. Conceptually, this dedicated storage area functions as a lookup table which internally maps between a set of output results (i.e., the data itself) and a set of input parameters (i.e., identifier(s) of the stored data). Once the data is stored in VC, it may be accessed and retrieved while the step-by-step calculation from the initial source input parameters is bypassed i.e., a number of instructions may be eliminated; not executed.

However, while exploiting the concept of redundancy seems promising, our experimental findings showcase that by simply memoizing the outcomes of single instructions or a set of instructions is not able to provide performance or power benefits. This is mainly due to the limited opportunities of successful value reuses (i.e., value cache hits) which eventually stem from the high-precision calculations required by modern graphics APIs (e.g., OpenGL|ES 3.0).

To overcome this issue, we introduce the concept of approximate value reuses. In particular, the proposed VC mechanism is equipped with the ability of performing partial matches i.e., reducing the arithmetic precision (accuracy) of the input parameters. Obviously, this is an effective way to significantly ameliorate the volume of VC hits. However, by relying on partial matches, we may introduce errors in the arithmetic calculations and eventually reduce the quality of the rendered images or animations.

In the context of this work, we chose to apply our clumsy VC memoization mechanism in the fragment shading operations of a typical graphics processing pipeline. Fragment shaders typically consist of complex arithmetic operations that may incorporate the geometric and appearance descriptions of the rendered objects and the environment. The target of the fragment programs is to compute the final color value of a pixel. The key insight is that the color values generated by the fragment shaders will be interpreted by the human senses, which are not perfect, thus it is possible to introduce small and controllable errors during the fragment shading operations, if such an approach will result in power savings.

The contributions of this work are as follows:

- We perform a classification of all fragment shaders' instructions in order to characterize their potential for memorization and their impact on the quality of the rendered images when we deliberately ignore a few low order bits.
- We develop a methodology to automatically insert the VC instructions in selected code regions (groups of instructions) of the target fragment shaders.
- We statistically evaluate the effectiveness of our methodology with respect to accuracy and image quality using profiling.
- We devise and optimize various run-time, feedback-directed policies to control the interplay between accuracy and quality maximizing the value reuse benefits at the same time.

II. Observations

In order to understand the potential of value reuses in OpenGL applications, we perform an extensive analysis of various fragment programs. In this section, we summarize the main conclusions of our analysis with respect to redundancy and accuracy.

Value reuses are not equal between instructions. As a first step, we evaluate the value reuse potential in a per-instruction basis for various fragment shaders. Omitting at this point the simulation details, Fig. 1 depicts our gathered statistics averaged across all fragment programs/games. The vertical axis in the graph shows the VC hits i.e., the percentage of the redundant operations that are captured by our memoization mechanism. The x-axis shows the studied instructions categorized as vector or scalar instructions. For clarity reasons, the reported statistics correspond to the averaged behavior of each instruction across all fragment programs/games. There are two stacked bars attached to each instruction depicting the percentages for an 8-entry and 32-entry VC respectively. In all cases, the blue parts of the bars show the measured statistics when only the concept of redundancy is assumed (full 32-bits matches), while each additional bar segment on top illustrates the extra benefits when 8, 16, and 20 bits are ignored during the comparison between a new input and the VC stored entries (the concept of accuracy is exploited).

As it can be seen from Fig. 1, the number of redundant operations captured by VC differs significantly among the instructions. If we concentrate only in the blue parts of the bars (redundancy), the two-input vector instructions (MUL, ADD, and SGE) exhibit a promising behavior. For example, more that 24% hit ratio is reported by MUL and more than 50% by SGE (vector greater or equal comparison). On the other hand, the three-input vector instructions (MAD and DP3/vector dot-product), as expected, show a poor value reuse potential (less than 1% hit ratio in DP3). Exactly the same behavior appears in the power consuming scalar instructions (EX2/exponential, LG2/logarithmic, and RSQ/reciprocal square root). Thereby, it becomes obvious that relying only on the concept of redundancy results in diminishing benefits, invalidating the usage of a memoization mechanism in graphics applications.

However, the situation changes radically when the concept of accuracy is introduced. By forcing the VC mechanism to move from full (32-bit) matches to partial matches during the VC comparison process, a remarkable ramp up in the hit percentages can be observed. As it is evident from Fig. 1, compared to full matches (0-bits in the graph), the average additional increase in VC hit ratios is 23.5% and 56.3% when 16-bits (green parts of the bars) and 20-bits (purple parts) respectively are ignored in the VC comparison process. On the contrary, by ignoring only 8-bits produces meager extra benefits (the red parts of the bars are not visible in the graph in most of the cases). The end result is that aggressive reductions in the precision of the VC comparison operations must be enforced in order to exploit the potential of value reuses in graphics applications. Of course, the latter trend will have a negative impact in the quality of the generated frames. This impact is examined in the rest of this section. Note that during the VC comparisons, we always keep the sign bit and the full exponent of the corresponding 32-bit floating point numbers and we reduce the bits of mantissa. For example, when we ignore 16-bits, mantissa is handled as a 7-bit vector (instead of 23-bits as in the regular case).

Finally, as Fig. 1 also indicates, by comparing the left and the right bar in every pair of bars, it is clear that increasing the size of the value cache barely produces better results. This is a promising result leading to the fact that a frugal value cache memory array is required (the only exception is the TEX instruction which shows an almost linear improvement in the reported hits when VC size increases). The impact of VC size is further analyzed in the next paragraph.

Value reuse potential in code segments does not exist.

A straightforward way to increase the pay offs from the value reuse mechanism is to increase the size of the reused blocks. By doing this, a VC hit will be able to eliminate a larger number of executed instructions. For example, if we assume that the VC stores the inputs/outputs of a block consisting of five instructions, then a single VC hit will skip the execution of five instructions arising the achieved performance and power benefits from value reuses. However, by increasing the size of reused blocks (called VC blocks hereafter), we also increase the number of inputs and outputs that must be stored in the VC storage area. This, in turn, significantly lowers the possibility to experience a VC hit, since a larger number of input arguments must be compared and matched. Thus, in order to maximize the resulting benefits, we must find a correct balance between the following contradictory trends: large VC blocks increase the benefits reported by VC hits, but large VC blocks lead also to smaller VC hit ratios. The latter trend can be also seen in Fig. 2. The graph in this figure shows the VC hit ratios (y-axis) for various VC block sizes (x-axis) assuming that only the concept of redundancy is employed (32-bit matches). In all cases, the statistics presented in Fig. 2 are averaged numbers across various block sizes arbitrarily selected from the studied fragment programs.

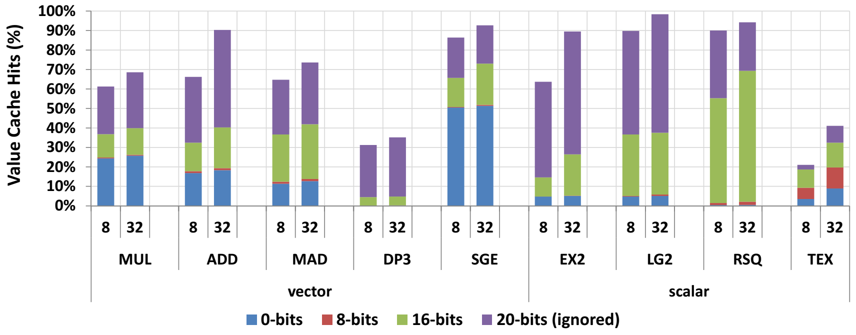


Fig. 1. Potential of redundant operations for each instruction in the studied fragment shaders.

Each bar in Fig. 2 represents a different VC size starting from a 32-entries VC (left bar) up to 32K-entries VC (right bar). In fact, the 32K case corresponds to infinity since no changes have been observed with larger VC sizes. As we can see from the graph, VC block sizes larger than 2 instructions almost eliminate the potential of value reuses even with unrealistic VC sizes. Hence, the only viable way to increase the number of VC hits without resorting to costly reuse hardware is to strongly rely in the output precision of graphics applications (i.e., the concept of accuracy) which is the subject of the next paragraph.

Not all instructions in fragment shaders must be equally precise. To understand the impact of changing the precision in fragment shaders, we perform a set of experiments varying the precision of the various operations of the studied fragment shaders. More specifically, for each target instruction, we statically reduce (before the execution) its precision in steps of 4-bits and we measure the impact in the quality of the rendered images. Note that during those experiments our VC mechanism is disabled. Our analysis reveals that the fragment shaders instructions can be divided into two main categories: arithmetic operations and texture fetches. Each category exhibits very different sensitivities to precision reduction. Fig. 3 demonstrates the errors appearing when the precision reduction is applied in all the instructions of the target programs (image in the middle), while the right icon depicts the case in which reduced precision is forced only in the arithmetic calculations (texture fetches are still done in full precision). In both cases, the reduction in precision is performed by employing the half precision floating point format (16-bits).

As we can see, when the reduced precision is applied only in the arithmetic calculations, no perceptive differences in the generated frame are introduced. In general (not shown in the figure), arithmetic imprecisions manifest themselves in the computation of color values in two ways: they gently darken the scene as LSBs are dropped and smooth color gradients can appear blocky as nearby values are quantized to the same result.

On the contrary, as the right bottom icon of Fig. 3 illustrates, texture fetches are much more sensitive to variations in input precisions. This was expected since texture coordinates are effectively indices into an array. Using slightly incorrect indices to index an array can lead to results that are very wrong, correct, or anywhere in between. The behavior is dependent on parameters such as the frequency of the texture data, size of the texture, and type of the texture filtering algorithm. Reduced precision texture coordinates will lead to neighboring pixels fetching the same texel. In some pathological cases, texture coordinates for entire

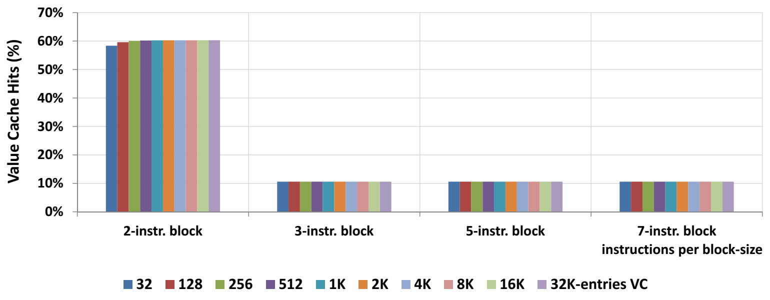


Fig. 2. Potential from value reuses for various VC block sizes.

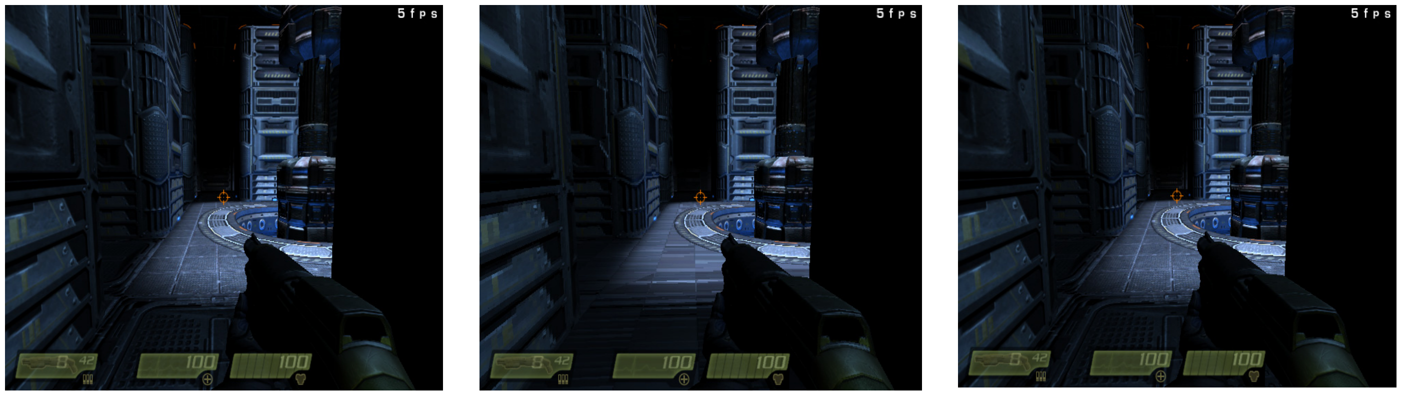


Fig. 3. Image in left is the reference frame produced by full-precision computations (32 bits) throughout the fragment shader. Image in the middle shows the result of reducing the precision of texture coordinates to 16 bits. Image in right shows the result of reducing the precision of color computations to 16 bits. There are no perceptive differences in the latter case.

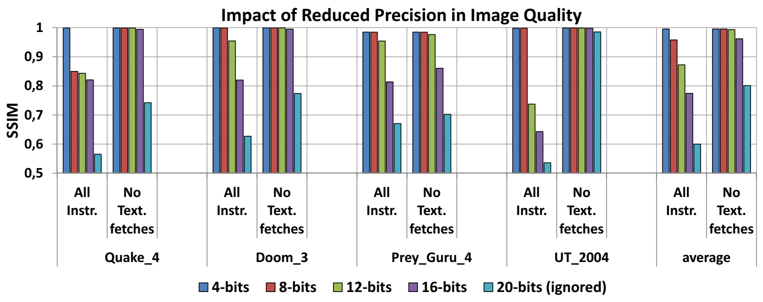


Fig. 4. Impact of reduced precision.

triangles may collapse to the same value when using a slightly reduced precision, giving the triangle a single color. This effect is visible in the ground floor in Fig. 3

Furthermore, Fig. 4 quantifies the impact of reduced precision for the two instruction categories and for the four benchmarks that we consider in this work. The y-axis shows the effect in the image quality using the SSIM metric [5]. There are two groups of bars attached in each game. The first group shows the impact in the image quality when the precision reduction is forced in all the pixel shader instructions, whereas the second group depicts the same impact when only the arithmetic calculations are performed with reduced precision. As we can see, the same behavior appears in all cases. The precision of the arithmetic calculations can be aggressively reduced (dropping up to 12 bits) without introducing perceptive errors in the images (actually this can go up to 16-bits in all cases except Prey_Guru_4). On the other side, only a 4-bit precision reduction in texture fetches can be allowed. Further reducing the precision of texture fetches will result in significant and intolerable errors. Therefore, it becomes apparent that the control over the precision of the two instruction groups (arithmetic calculations and instruction fetches) must be done independently.

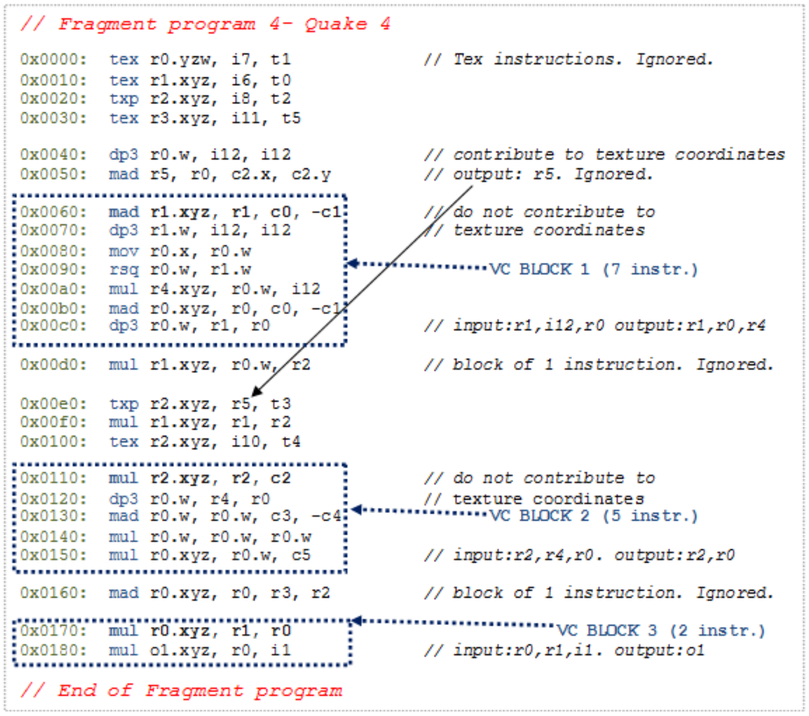
III. VC Block Selection Policy

Taking into account the above observations, we now present our methodology to identify the code blocks (sets of instructions) that are more suitable for reuse. In essence, the proposed memoization mechanism is block based and it is controlled by special machine-level instructions (part of the GPU ISA). In general, the primary operations performed by VC are the AddEntries and the LookupEntries instructions. AddEntries places new results in the value cache and LookupEntries retrieves one or more entries from the value cache, in case of a VC hit, or produces misses if there is no corresponding entry for the sought input parameters.

Our goal in this section is to devise an appropriate VC block selection policy. In other words, given a fragment shader, our methodology should output the suitable blocks of instructions for value reuses. Those VC blocks will be annotated with the above VC instructions i.e., the LookupEntries will be inserted at the start of the selected VC block and the AddEntries at the end.

Our VC selection methodology is applied in the control/data flow graph (CDFG) of the target fragment programs and its operation is as follows:

- **step 1:** parse the target fragment shader in a top-down fashion.
- **step 2:** seek for code regions with no more than three input and three output parameters (registers). The reasoning behind this choice will be explained below.
- **step 3:** if a code block with no more than three input and three output parameters is found, the code parsing is stopped. If the just traversed block contains at least two instructions, the block is selected for value reuse. The process continues starting from the next instruction after the block.



```
// Fragment program 4- Quake 4

0x0000:  tex r0.yzw, i7, t1           // Tex instructions. Ignored.
0x0010:  tex r1.xyz, i6, t0
0x0020:  txp r2.xyz, i8, t2
0x0030:  tex r3.xyz, i11, t5

0x0040:  dp3 r0.w, i12, i12           // contribute to texture coordinates
0x0050:  mad r5, r0, c2.x, c2.y       // output: r5. Ignored.

0x0060:  mad r1.xyz, r1, c0, -c1      // do not contribute to
0x0070:  dp3 r1.w, i12, i12           // texture coordinates
0x0080:  mov r0.x, r0.w
0x0090:  rsq r0.w, r1.w               VC BLOCK 1 (7 instr.)
0x00a0:  mul r4.xyz, r0.w, i12
0x00b0:  mad r0.xyz, r0, c0, -c1
0x00c0:  dp3 r0.w, r1, r0             // input:r1,i12,r0 output:r1,r0,r4

0x00d0:  mul r1.xyz, r0.w, r2         // block of 1 instruction. Ignored.

0x00e0:  txp r2.xyz, r5, t3
0x00f0:  mul r1.xyz, r1, r2
0x0100:  tex r2.xyz, i10, t4

0x0110:  mul r2.xyz, r2, c2           // do not contribute to
0x0120:  dp3 r0.w, r4, r0             // texture coordinates
0x0130:  mad r0.w, r0.w, c3, -c4      VC BLOCK 2 (5 instr.)
0x0140:  mul r0.w, r0.w, r0.w
0x0150:  mul r0.xyz, r0.w, c5         // input:r2,r4,r0. output:r2,r0

0x0160:  mad r0.xyz, r0, r3, r2       // block of 1 instruction. Ignored.

0x0170:  mul r0.xyz, r1, r0           VC BLOCK 3 (2 instr.)
0x0180:  mul o1.xyz, r0, i1           // input:r0,r1,i1. output:o1

// End of Fragment program
```

Fig. 5. VC Block selection policy (notations: r: register file, i: input register file, o: output register file, c: constant register file, t: texture unit).

	frame10	Frame50	frame100	frame150	Frame200	Frame250	Frame300
Quake_4	60%	60%	60%	60%	60%	60%	60%
Doom_3	54%	54%	57%	57%	55%	55%	54%
Prey_Guru_4	53%	54%	54%	54%	55%	55%	55%
UT_2004	67%	64%	68%	64%	72%	76%	73%
Average code coverage: 58,7%							

Fig. 6. Code coverage of the proposed VC block selection policy.

- **step 4:** if a TEX instruction is reached, the code parsing is stopped. Again, if the just traversed block contains at least two instructions, the block is selected for value reuse. The process continues starting from the next instruction after the TEX instruction.
- **step 5:** if an arithmetic instruction affects (calculates) a texture address (i.e., a TEX instruction is dependent to this instruction), the code parsing is also stopped. Again, if the just traversed block contains at least two instructions, the block is selected for value reuse. The process continues starting from the next instruction after the block.

To further explain the proposed policy, an example code is illustrated in Fig. 5. As indicated by the blue boxes in the code, our policy manages to find three VC blocks consisting of 7, 5, and 2 instructions respectively.

The proposed VC block selection process is straightforward and it can be easily incorporated in the back-end of a typical compiler. As part of this work, this methodology has been integrated in the Attila simulator [1] as an extra trace parser and the Attila ISA has been appropriately extended to include the new VC instructions.

A key point of our selection policy is the maximum number of input and output instructions that are allowed in each VC block. Of course, blocks with less than three input/output registers can still be eligible VC blocks. First, if we limit the number of inputs/outputs we will end up with small VC blocks diminishing the performance and power gains of value reuses (for the same reason 1-instruction blocks are not candidates for VC blocks). In contrast, if we increase the allowed input/output registers, then i) the number of VC hits will be significantly reduced (as shown in the previous section), and ii) VC hardware will become unacceptably costly. We experimentally found that the three inputs/three outputs policy strives the best balance between the two contradictory trends. In addition, modern mobile GPUs already contain hardware support for reading three registers simultaneously due to the existence of instructions like MAD.

Finally, the effectiveness, in terms of code coverage, of the proposed VC block selection policy can be seen in the array presented in Fig. 6. Code coverage is defined as the percentage of instructions that are assigned to VC blocks normalized to the total number of instructions. As Fig. 6 illustrates the code coverage ranges from 53% (Prey_Guru_4/frame_10) to 73% (UT_2004/frame_200), while the averaged coverage across all the studied games/frames is 58.7%.

IV. VC Practical Implementation

The proposed memoization mechanism is implemented as a new functional unit (called Value Cache Functional Unit or VCFU) that is controlled by special machine-level instructions (part of the GPU ISA). As mentioned, each selected VC block will be surrounded by two assembly-level instructions namely the AddEntries and the LookupEntries instructions. Assuming a VC block consisting of N instructions, a VC hit means that only the LookupEntries instruction is executed; the VC block is skipped. The end result is that the number of the executed instructions is reduced by N-1 when a VC hit occurs. On the contrary, in a VC miss, the number of executed instructions is increased by 2 (N+2). Therefore, if not enough VC hit rate is achieved or code blocks are small, the VC mechanism can obtain negative returns. In Section VI and Section VII, we present our approach to increase the VC hit ratio by controlling, in a static or dynamic way, the precision of the VC inputs.

V. Simulation Infrastructure

As noted, the proposed mechanism is applied in the OpenGL fragment shaders and it is evaluated using four OpenGL games (Doom 3, Quake 4, Unreal Tournament 2004, and Prey Guru 4).

The evaluation is performed in the Attila simulator [1]. Attila is a architectural-level, cycle accurate simulator capable to work at the OpenGL level.

VI. Profiling Results: Value Reuse Benefits vs QoS

As noted in the previous section, the VC approach introduces more instructions in the target fragment programs. As a result, the usage of VC may obtain negative returns by increasing the number of executing instructions if not enough VC hit ratio is reported. In order to increase the VC hit ratio, we enforce an aggressive reduction in the precision of the VC lookup operations. This reduction is performed by ignoring a controlled number of bits during the comparison between a new VC input and the VC stored entries (i.e., the concept of accuracy is exploited). Our effort in this section is to study the relation between the quality of output frames and VC performance when the reduction in the precision is performed in a static manner. Fig. 7 presents the results of this analysis.

Fig. 7 (top graph) shows the measured VC hit ratio as averaged values over all VC blocks of all studied fragment programs/games. The graph in the middle depicts the numbers of instructions saved by our proposal i.e., the percentage of the executed instructions when the value cache is activated normalized to the initial number of instructions. In addition, the bottom graph of Fig. 7 illustrates the losses in image quality reported in each case. In all figures, the horizontal axis contains five groups of bars. The four leftmost groups correspond to the four studied OpenGL games. The rightmost group of bars (tagged as "low_complexity") is a collection of frames with poor graphics contents e.g., game menus, simple animations etc. The frames comprising the latter category have been arbitrarily selected from the four OpenGL games. Finally, there are eight bars in each group corresponding to different precisions. The leftmost bar tagged as "0-bits" show the results when only the concept of redundancy is exploited (full 32-bits VC matches). Every bar in the right side of "0-bits" bars corresponds to a different precision starting from the case in which 4-bits are ignored during the VC comparison process. The number of ignored bits ranges from 4 bits up to 24-bits. In the latter case, the mantissa part of the floating point numbers is actually not involved in the VC comparison operations. Note that in all cases, the same number of ignored bits is applied in all the selected VC blocks.

Several interesting conclusions can be drawn from the graphs presented in Fig. 7. First, if we concentrate on the "0-bits" bars, it is evident that a memoization mechanism, that cannot take advantage of the tolerance in graphics applications, fails to produce value reuse benefits. The value cache hit ratios range from 21.6% in UT2004 to 61.4% in the "low_complexity" category (36.8% on average). In essence, this number represents the percentage of the redundant operations in the under evaluation fragment shaders. However, the graph in the middle shows that even if our memoization technique is able to capture those redundant operations, the end result is that the number of the executed instructions is actually increased in all studied games. The only exception is the "low_complexity" category in which 12.3% less executed instructions are observed. As a result, it is evident that the only viable way to experience benefits from value reuses is to aggressively reduce the precision of the input data.

The negative returns from memoization continue up to the "16-bits" case. If we further reduce the precision, the value cache mechanism starts to pay off. The "20-bits" case is actually an inflection point above which the savings in the number of the executed instructions start to appear. More specifically, the saved instructions in the "20-bits" case are 8.5% (on average). In fact, the "20-bits" case is an appealing design point not only due to the instruction savings, but also due to the fact that non-noticeable errors are introduced in the output frames (the losses in the image quality are well below 0.8% in all cases). Furthermore, if a more aggressive reduction in precision is applied, the percentage of the saved instructions remarkably ameliorates, but the image quality decreases to unacceptable levels. However, this trend is not uniform across all benchmarks. For example, in the "22-bit" case,

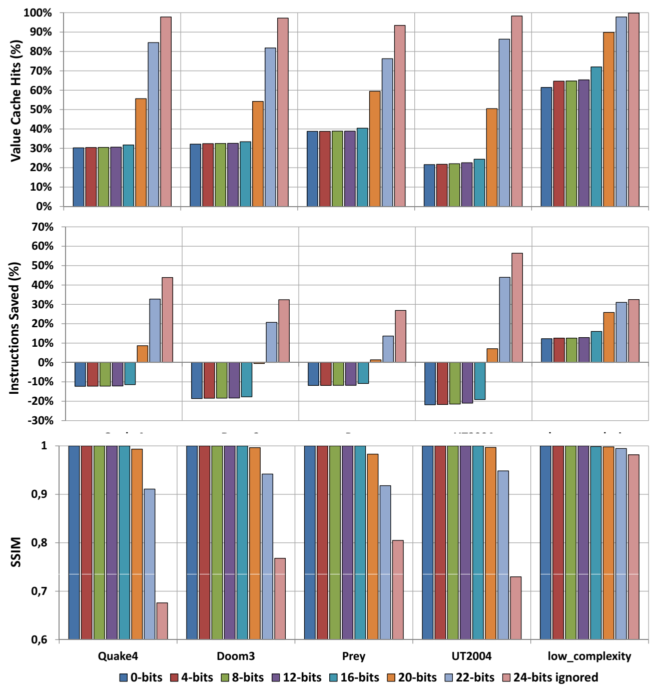


Fig. 7. VC hit ratio (top), percentage of saved executed instructions (middle), and impact in image quality (bottom) for the static policy for various precision levels.

the instruction savings are 20.7% in Doom3 and 44% in UT2004, while in both games the image losses are below 6%. Consequently, this can be a suitable design point. On the other hand, as it was expected, the "low_complexity" frames exhibit a great tolerance in precision (see SSIM graph) even if the precision is reduced by 24-bits.

VII. Dynamic Value Cache Mechanism

The main drawback of a VC mechanism with a statically defined precision is its strong reliance on application profiling and the inability of the technique to adapt to the variability of the graphics contents being used as inputs. To address that, we introduce a hardware-level dynamic technique (called Dynamic Value Cache or DVC) to choose the optimal VC operating precision. At the highest level, DVC is a closed-loop method by which the resulting errors can be monitored as VC precision is reduced at run-time. When a predefined number of errors occurs, the precision of VC is increased to avoid continued errors. In the rest of this section, we briefly describe the proposed closed loop mechanism, we experimentally define the control parameters of the feedback algorithm, and we present the overall reuse benefits achieved by DVC.

Design parameters of the feedback mechanism. The design parameters of the proposed close loop method are: the sense interval, the sampling period, the error threshold, and the number of allowed errors. To monitor VC performance, the execution time is divided into fixed length intervals, called sense intervals. In our setup, the sense intervals are measured in terms of executed fragment program instances. For example, if the sense interval is set to 2500, every 2500 executed fragment programs a new selection in the precision of the VC operations will be performed.

The remaining three parameters are used to account for the number of errors occurred during a sense interval. The allowed errors are determined by an experimentally defined threshold. However, a critical question is how to measure the errors introduced by the reduced precision VC operations. To do this, the hardware must compute for the same inputs both the result produced by a successful VC lookup (VC hit) as well as the result of the normal fragment program execution (the latter is implemented by enforcing the VC to produce misses based on a

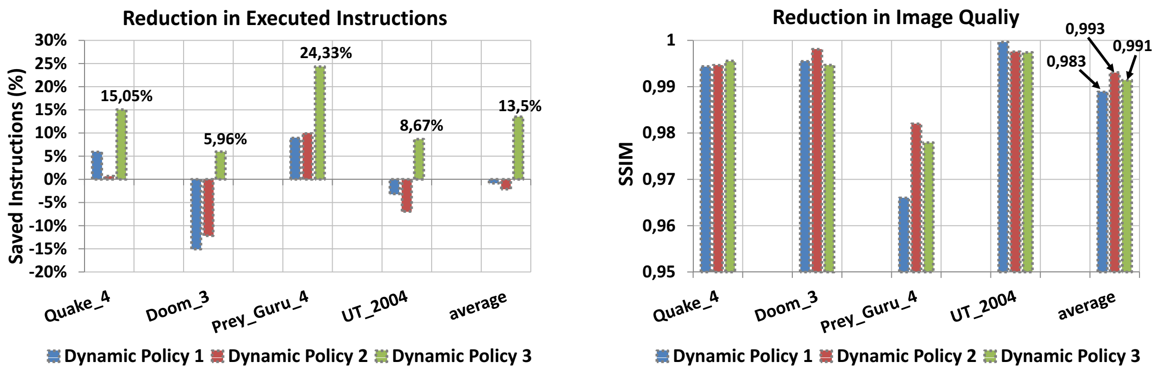


Fig. 8. Average instruction savings (left) and average impact in image quality (right) for the three dynamic policies.

VC visible flag bit). The two results are compared (using a simple 32-bit subtraction) and if the result of the comparison is larger than a preset threshold (error threshold), an error is reported. This process is enabled by the fact that every fragment program has a single output point i.e., the end result of each fragment program is always assigned to a single register.

An obvious shortcoming of this dual-mode execution is its large overheads. To overcome this, we investigate various sampling techniques. We explore different sampling rates and sampling patterns. Our results reveal a very promising characteristic: by sparsely sampling every n-th generated fragment program outputs performs nearly as well as denser random sampling. In the rest of this section, we statically set the sampling period to 100 meaning that one every 100 fragment program outputs is selected for error monitoring. In addition, we assume that the API driver is equipped with extra control logic to implement this dual-mode execution. The overheads (extra instructions) of the dual-mode executions are taken into account in our experimental results.

Adaptation strategy. Apart from the error monitoring logic, we must also determine how to change the operating precision. Towards this direction, we examine two main strategies. The first strategy, called lock/unlock strategy, is related to the learning phase of our adaptation policy. Starting from the highest possible precision (full precision), the lock strategy continuously reduces the precision of the VC operations in every sense interval, if the feedback module indicates that the sought errors (if any) are below the predefined thresholds. This process is finished when the errors are above the tolerance thresholds. At this point, the precision is increased by one and it is locked in this new value. After this learning phase, the closed loop mechanism is disabled until the end of the current frame and the remaining VC operations are executed with the new precision. On the other hand, in the unlock strategy, there is no learning phase. The close loop module continuously monitors the resulting errors and the VC precision is increased/decreased by one unit (one bit) in each sense interval depending on the outcome of the feedback module. Obviously, the unlock mechanism requires a more tight control over the generated errors.

The second strategy, called local or global strategy, aims to define if the precision control will be applied in a per-fragment program basis (called local) or globally among the fragment programs (all the VC operations of all the fragments will be enforced to dictate to the same precision level). An obvious benefit of the per-fragment precision control is that the selected precision will be tailed to the special characteristics of each fragment program. On the other hand, longer learning phases are required if the locking strategy is employed (a learn phase for each fragment program). In addition, the per-fragment program precision control requires an extra table to store the last operating precision of each program. However, this is not considered as an important drawback, since the number of the simultaneously running fragment programs is limited (typically below five in our setup).

Fine tuning the feedback mechanism. In order to fine tune the feedback mechanism, we perform a series of experiments varying the length of the sense interval, the error threshold, and the number of allowed errors. Obviously, the selection of the configuration must be done as a compromise between the VC performance and the image quality degradation. In particular, we use the following criterion in both strategies: we opt the configuration that exhibits the highest VC hit ratio without reducing the image quality by more than 2% i.e., the output errors are below noticeable levels. Due to lack of space, the results of this analysis are omitted.

Evaluation of various dynamic policies. Based on the parameters extracted by our fine turing step, we devise and evaluate three different policies to control the operating precision of the dynamic value cache mechanism. Those policies are: i) *Policy 1:* lock strategy and local strategy, ii) *Policy 2:* unlock strategy and local strategy, and iii) *Policy 3:* unlock strategy and global strategy.

The left graph in Fig. 8 shows the number of saved instructions achieved by each policy, while the right graph depicts the impact in the image quality reported by each technique. In both graphs, the statistics are presented in a per game basis (averaged among all fragment programs in each game). As Fig. 8 indicates, policy 3 is the best performing policy achieving a reduction in the number of executed instructions of 15.1%, 6%, 24.3%, and 8.7% in Quake_4, Doom_3, Prey_Guru_4, and UT_2004 respectively. The average instruction savings are 13.5%. The other two policies offer diminishing or even negatives returns from value reuse. Moreover, policy 3 manages not only to reduce the number of executed instructions, but to do so with a negligible impact in image quality (well below 2% in each game and 0.8% on average).

VIII. Conclusions

In this work, we present the Value Cache mechanism. VC targets to remove the redundant, complex arithmetic operations in OpenGL graphics applications. Our results show that by employing a typical hardware mechanism to exploit the redundant operations leads to meager benefits even if impractically large memoization tables are used. To overcome this issue, we heavily rely on the concept of reducing the accuracy of the value memoization comparisons in a dynamic fashion. Overall, our approach manages to reduce the number of executing instructions in modern fragment shaders by 13.5% with a negligible loss in the quality of the rendered images.

References

[1] Attila framework: http://attila.ac.upc.edu/

[2] ARM, Transaction elimination. Available: http://www.arm.com/products/multimedia/mali-technologies/transaction-elimination.php

[3] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Parallel frame rendering: Trading responsiveness for energy on a mobile gpu. Proc. of Intl. Conference on Parallel Architectures and Compilation Techniques, 2013.

[4] J-M. Arnau, J-M. Parcerisa, and P. Xekalakis. Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization. Proc. of Intl. Symposium on Computer Architecture, 2014.

[5] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. IEEE Transactions on Image Processing, 2004.