Performance Analysis for PDF Files with Transparency

Dave Eberly
dave.eberly@artifex.com
February 16, 2010

This report is about performance issues with PDF files that use transparency. A list of files was generated by Ray Johnston. Each file in the list was processed using the command line

```
gswin32c -r300 -q -sDEVICE=ppmraw -o nul: -Z: -dLastPage=1 <filename>
```

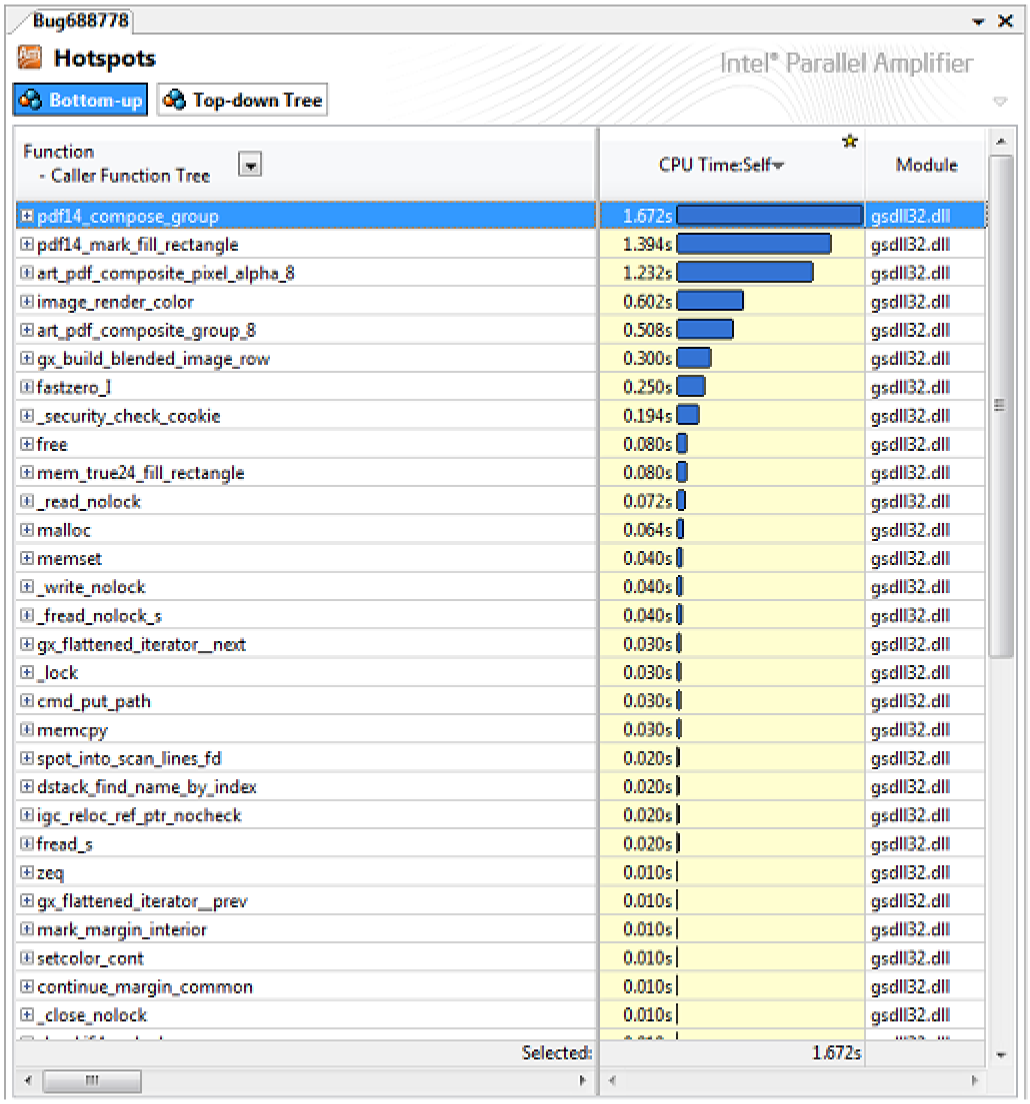
The transparency was then ignored using the command line

```
gswin32c -r300 -q -sDEVICE=ppmraw -o nul: -Z: -dLastPage=1 -d NOTRANSPARENCY <filename>
```

The ratio of time drawn with transparency to time drawn without transparency was computed. The spreadsheet `pdf_trans_performance.xls` contains the timings, sorted by column A that contains the ratio.

The source code was compiled using Microsoft Visual Studio 2005 and profiled on an Intel PC using Intel's Parallel Amplifier for timing. Additional information about problems at the assembly language level was obtained using Intel's VTune. The results might vary slightly for different compilers and different CPUs, but the conclusions are generally the same.

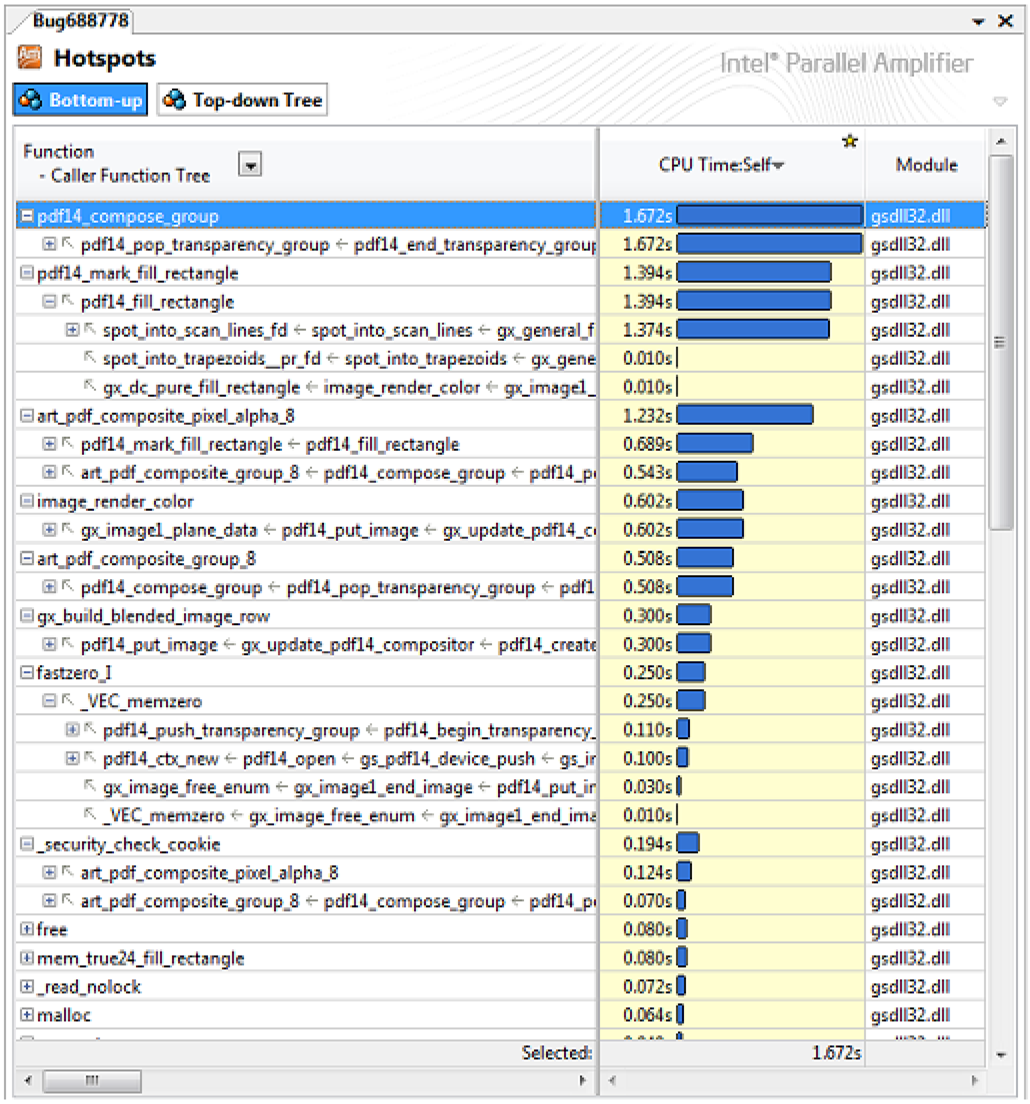
A sample PDF that illustrates most of the performance bottlenecks is `Bug688778.pdf`. Figure 1 shows the summary of hotspots.



Function - Caller Function Tree	CPU Time:Self	Module
pdf14_compose_group	1.672s	gsdll32.dll
pdf14_mark_fill_rectangle	1.394s	gsdll32.dll
art_pdf_composite_pixel_alpha_8	1.232s	gsdll32.dll
image_render_color	0.602s	gsdll32.dll
art_pdf_composite_group_8	0.508s	gsdll32.dll
gx_build_blended_image_row	0.300s	gsdll32.dll
fastzero_I	0.250s	gsdll32.dll
_security_check_cookie	0.194s	gsdll32.dll
free	0.080s	gsdll32.dll
mem_true24_fill_rectangle	0.080s	gsdll32.dll
_read_nolock	0.072s	gsdll32.dll
malloc	0.064s	gsdll32.dll
memset	0.040s	gsdll32.dll
_write_nolock	0.040s	gsdll32.dll
_fread_nolock_s	0.040s	gsdll32.dll
gx_flattened_iterator__next	0.030s	gsdll32.dll
_lock	0.030s	gsdll32.dll
cmd_put_path	0.030s	gsdll32.dll
memcpy	0.030s	gsdll32.dll
spot_into_scan_lines_fd	0.020s	gsdll32.dll
dstack_find_name_by_index	0.020s	gsdll32.dll
igc_reloc_ref_ptr_nocheck	0.020s	gsdll32.dll
fread_s	0.020s	gsdll32.dll
zeq	0.010s	gsdll32.dll
gx_flattened_iterator__prev	0.010s	gsdll32.dll
mark_margin_interior	0.010s	gsdll32.dll
setcolor_cont	0.010s	gsdll32.dll
continue_margin_common	0.010s	gsdll32.dll
_close_nolock	0.010s	gsdll32.dll
Selected:	1.672s	

Figure 1. The summary of hotspots for Bug688778.pdf.

Figure 2 shows an expanded tree control that indicates the call stacks leading to the hotspot.



Function - Caller Function Tree	CPU Time:Self	Module
⊟ pdf14_compose_group	1.672s	gsdll32.dll
⊞ pdf14_pop_transparency_group ← pdf14_end_transparency_group	1.672s	gsdll32.dll
⊟ pdf14_mark_fill_rectangle	1.394s	gsdll32.dll
⊟ pdf14_fill_rectangle	1.394s	gsdll32.dll
⊞ spot_into_scan_lines_fd ← spot_into_scan_lines ← gx_general_f	1.374s	gsdll32.dll
spot_into_trapezoids__pr_fd ← spot_into_trapezoids ← gx_gene	0.010s	gsdll32.dll
gx_dc_pure_fill_rectangle ← image_render_color ← gx_image1_	0.010s	gsdll32.dll
⊟ art_pdf_composite_pixel_alpha_8	1.232s	gsdll32.dll
⊞ pdf14_mark_fill_rectangle ← pdf14_fill_rectangle	0.689s	gsdll32.dll
⊞ art_pdf_composite_group_8 ← pdf14_compose_group ← pdf14_p	0.543s	gsdll32.dll
⊟ image_render_color	0.602s	gsdll32.dll
⊞ gx_image1_plane_data ← pdf14_put_image ← gx_update_pdf14_c	0.602s	gsdll32.dll
⊟ art_pdf_composite_group_8	0.508s	gsdll32.dll
⊞ pdf14_compose_group ← pdf14_pop_transparency_group ← pdf1	0.508s	gsdll32.dll
⊟ gx_build_blended_image_row	0.300s	gsdll32.dll
⊞ pdf14_put_image ← gx_update_pdf14_compositor ← pdf14_create	0.300s	gsdll32.dll
⊟ fastzero_I	0.250s	gsdll32.dll
⊟ _VEC_memzero	0.250s	gsdll32.dll
⊞ pdf14_push_transparency_group ← pdf14_begin_transparency_	0.110s	gsdll32.dll
⊞ pdf14_ctx_new ← pdf14_open ← gs_pdf14_device_push ← gs_ir	0.100s	gsdll32.dll
gx_image_free_enum ← gx_image1_end_image ← pdf14_put_ir	0.030s	gsdll32.dll
_VEC_memzero ← gx_image_free_enum ← gx_image1_end_ima	0.010s	gsdll32.dll
⊟ _security_check_cookie	0.194s	gsdll32.dll
⊞ art_pdf_composite_pixel_alpha_8	0.124s	gsdll32.dll
⊞ art_pdf_composite_group_8 ← pdf14_compose_group ← pdf14_p	0.070s	gsdll32.dll
⊞ free	0.080s	gsdll32.dll
⊞ mem_true24_fill_rectangle	0.080s	gsdll32.dll
⊞ _read_nolock	0.072s	gsdll32.dll
⊞ malloc	0.064s	gsdll32.dll
Selected:	1.672s	

Figure 2. The call stacks of hotspots for Bug688778.pdf.

The top eight functions tend to show up in many of the profiles of the PDF files. The performance problems are classified as *data cache misses*, *per-pixel function call overhead*, *byte-channel processing overhead*, and *memory clearing*. There is also a performance issue with conversion from RGB or CMYK to gray scale, but that is easily remedied.

1 Data Cache Misses

The main bottleneck is data cache misses when converting between planar format and chunky format. In the profile of Figure 1, these occur in the calls to functions `pdf14_compose_group`, `pdf14_mark_fill_rectangle`, and `gx_build_blended_image_row`.

The function `pdf14_compose_group` has lines of code such as the following, all occuring in the `i`-loops within the double loop that processes all pixels.

```
/* conversion from planar to chunky */
for (i = 0; i < n_chan; ++i)
{
    tos_pixel[i] = tos_ptr[x + i * tos_planestride];
    nos_pixel[i] = nos_ptr[x + i * nos_planestride];
}

/* conversion from chunky to planar */
for (i = 0; i < n_chan; ++i)
{
    nos_ptr[x + i * nos_planestride] = nos_pixel[i];
}
```

When the strides are large, the memory fetches for `tos_ptr` and `nos_ptr` cause cache misses, which leads to a lot of stalls waiting for cache to be loaded from main memory.

In `pdf14_mark_fill_rectangle`, similar code occurs

```
/* conversion from planar to chunky */
for (k = 0; k < n_chan; ++k)
{
    dst[k] = dst_ptr[k * planestride];
}

/* conversion from chunky to planar */
for (k = 0; k < n_chan; ++k)
{
    dst_ptr[k * planestride] = dst[k];
}
```

And in `gx_build_blended_image_row`, there are lines of the form

```
comp = buf_ptr[x + planestride * comp_num];
```

In all cases, the large strides cause a lot of data cache misses. The recommended action is to avoid storing data in planar format and using only the chunky format for cache coherency.

2 Per-Pixel Function Call Overhead

The work performed inside the double *xy*-loop when processing image data is referred to as *per-pixel costs.* Minimizing the per-pixel costs will lead to better performance. The `art_pdf_*` functions compute the output pixel values given a set of inputs. The inputs contain information that leads to selection of the correct pixel drawing code. Moreover, the selection sometimes involves calling other `art_pdf_*` functions. For example, `art_pdf_composite_group_8` calls the function `art_pdf_composite_pixel_alpha_8` which in turn has the potential to call `art_blend_pixel_8`. This latter function contains a `switch` statement with 17 cases.

The function call overhead and selection code is relatively large, thus causing the `art_pdf_*` funtions to show up in the list of top offenders for computation time. The recommend action is to factor these functions into smaller lightweight functions and to move the selection code outside the double loop. This is much the way that pixel shaders are managed when running on a GPU. In fact, such a factorization will make it easier to integrate GPU support into the product.

The function `_security_check_cookie` is specific to Microsoft Visual Studio. This is code inserted to detect buffer overruns in local arrays in functions. The two `art_pdf_*` functions have such arrays, which causes the code injection. The buffer overrun detection is enabled by default (option `\GS`), but can be turned off. However, it is probably better to avoid having the local arrays in the first place (if possible). In my experiments with `pdf14_compose_group`, I had written a pixel shader that was called for a specific data set and did not have the local array. The function overhead was reduced greatly.

3 Byte-Channel Processing Overhead

The `i`-loops in `pdf14_compose_group` (and elsewhere) are designed to be general, handling whatever number of channels the incoming images might have. However, processing byte channels one at a time incurs the cost of shifting and masking by the CPU to extract the correct 8-bit quantity from a 32-bit value.

For the common case of four channels, it is better to process all four channels in parallel, a prerequisite being the avoidance of conversions from planar format to chunky format. In the case of additive color, data is copied from `tos_ptr` to `tos_pixel`. The pointers are typecast as 32-bit integer types and the copy is a single 32-bit copy. In the case of subtractive color, where the first three channels must be subtracted from 255, the parallel subtraction is accomplished by

```
    tos_pixel32bit[i] = 0x00FFFFFF ^ tos_ptr32bit[i];
```

for a little-endian machine. This maps (c_0, c_1, c_2, c_3) to $(255 - c_0, 255 - c_1, 255 - c_2, c_3)$. The mask for big endian is `0xFFFFFF00`. Experiments showed that this has a significant impact on the performance.

The function `image_render_color` also shows up on several profiles. Its performance problem is also a result of byte-channel processing. The following lines of code are problematic and cause a “blocked store forward” and a “partial register stall”. These are both expensive penalties.

```
    next.v[0] = psrc[0];
    next.v[1] = psrc[1];
    next.v[2] = psrc[2];
    psrc += 3;
    if (posture != image_skewed && next.all[0] == run.all[0])
```

The data type `color_samples` is a union with field `v` an array of bytes and with field `all` an array of integers, which leads to the problem. The compiler tries to optimize the register loading. The address `psrc` is loaded into register EAX. The assembly generated for the three accesses of the `v` field involves loading `psrc[]` values (from EAX) into the partial registers BL, CL, and DL. The assembly code for the if-line has a comparison of `posture` to `image_skewed` whose result is tested later in the assembly. There are copies of BL, CL, and DL to the stack locations for `next.v[]`, completing the assignments `next.v[*] = psrc[*]`. Assembly code then occurs to set up for the comparison of `next.all[0]` to `run.all[0]`. The first instruction loads the address `next` into the ESI register, which causes the stall because of the three preceding accesses to the partial registers (which involve `next`). The remaining code: EAX is incremented by 3, so a copy from EAX back to the stack location for `psrc` occurs. The comparison result for `posture` and `image_skewed` is then tested to see whether to jump. `run.all[0]` is compared to ESI (contains `next.all[0]`) to see whether to jump.

Using four comparisons

```
   (next.v[0] == run.v[0] && next.v[1] == run.v[1] && ...)
```

avoids the penalty, but then the costs are in copying BL, CL, and DL to `next` immediately after BL, CL, and DL are loaded. The following also avoids the penalty (code for little endian):

```
    next.all[0] = (*(bits32*)psrc & 0x00FFFFFF) | (next.all[0] & 0xFF000000);
    psrc += 3;
    if (posture != image_skewed && next.all[0] == run.all[0])
```

with some expense in updating `next.all[0]`, but the operations do not involve partial registers. This performed the best of the three cases. It is not clear that there is an optimal solution for this example, but the idea is that the byte-channel processing is problematic.

4 Memory Clearing

The `fastzero_I` function is called when there is a `memset(*,0,*)` call. The assembly shows that the zeroing is implemented using MMX instructions (for speed). The function shows up on the profile only because the memory clearing occurs many times. In particular, memory is cleared on each call to `pdf14_ctx_new` and to `pdf14_push_transparency_group`. It is not clear to me whether the memory clears are need (I suspect they are). The only alternative that I can think of is related to fast depth-buffer clears, but it is not clear that such methods apply here.

5 Conversion of Colors to Gray Scale

Sometimes `Smask_Luminosity_Mapping` shows up on the profile. It is not one of the top offenders, but once the planar-chunky conversions are eliminated, this function has the opportunity to be noticeable in the profile. One place where the performance is an issue is

```
float temp = (0.30 * src[x + mask_R_offset] +
              0.59 * src[x + mask_G_offset] +
              0.11 * src[x + mask_B_offset]);
temp = temp * (1.0 / 255.0 );
dstptr[x] = float_color_to_byte_color(temp);
```

The `src[]` values are byte-valued, so the result `temp` is guaranteed to be in the interval $[0, 255]$. The mapping of `temp` to $[0, 1]$ occurs and then the `float_color_to_byte_color` macro clamps the value to $[0, 1]$, followed by a multiplication by 255. Firstly, the floating-point operations are relatively expensive. Secondly, the clamping requires floating-point comparisons, which are also relatively expensive.

An alternative is to use integer arithmetic.

```
/* Avoid floating-point arithmetic using the following
   approximations for the color weights:
       5033165/16777216 = 0.300000011
       9898557/16777216 = 0.589999973
       1845494/16777216 = 0.110000014
*/
unsigned int r = (unsigned int)src[x + mask_R_offset];
unsigned int g = (unsigned int)src[x + mask_G_offset];
unsigned int b = (unsigned int)src[x + mask_B_offset];
unsigned int result = (5033165*r + 9898557*g + 1845494*b) >> 24;
dstptr[x] = (byte)result;
```

Although fast, sometimes the result is off by 1 (in either direction). Of the 256^3 possible combinations of (r, g, b), 75100 are off by 1 (0.4% of the cases). The boundary cases are important to get right. It is the case that $(0, 0, 0)$ is mapped to 0 and $(255, 255, 255)$ is mapped to 255.

The other place where the performance is an issue is

```
float temp = (0.30 * ( 0xff - src[x + mask_C_offset]) +
              0.59 * ( 0xff - src[x + mask_M_offset]) +
              0.11 * ( 0xff - src[x + mask_Y_offset]) ) *
                     ( 0xff - src[x + mask_K_offset]);
temp = temp * (1.0 / 65025.0 );
dstptr[x] = float_color_to_byte_color(temp);
```

Using the same rational approximations to the coefficients, some arithmetic leads to the following fast code.

```
unsigned int c = (unsigned int)src[x + mask_C_offset];
```

```
    unsigned int m = (unsigned int)src[x + mask_M_offset];
    unsigned int y = (unsigned int)src[x + mask_Y_offset];
    unsigned int k = (unsigned int)src[x + mask_K_offset];
    unsigned int tmp0 = (4278190080 - (5033165*c + 9898557*m + 1845494*y)) >> 24;
    unsigned int tmp1 = 16777216 - 65793*k;
    unsigned int result = (tmp0*tmp1) >>  24;
    dstptr[x] = (byte)result;
```

The results are also off by 1 (in either direction), but for a greater fraction of the cases. Of the 256^4 possibilities, 1029544184 are off by 1 (24% of the cases). The boundary cases are correct: $(0, 0, 0, 0)$ is mapped to 255 and $(255, 255, 255, 255)$ is mapped to zero.

In both suggested code fragments, the integer arithmetic is designed so that there is no overflow.