Analysis of Bug 689606

Dave Eberly
dave.eberly@artifex.com
February 4, 2010

The bug refers to the following code block in function `pdf14_compose_group` in file `gs/base/gxblend1.c`.

```
for (y = y0; y < y1; ++y) {
    for (x = 0; x < width; ++x) {
        byte pix_alpha = alpha;
        if (additive) {
            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];
            }
        }
        <other code>
```

Comment 5 of the bug report provides profile information from Shark. Given that some time has passed since the comment was added, the experiment was repeated and the first four entries of the time-profile summary are listed next.

```
- 75.6%, pdf14_compose_group, gxps
- 14.4%, art_pdf_composite_group_8, gxps
- 4.4%, __bzero, libSystem.B.dylib
- 3.6%, Smask_Luminosity_Mapping, gxps
```

The profile source code views contain the percentage of time relative to total function time, listed at each line of the code (see Figures 1 and 2). The amount of time spent copying inside the `i`-loop is quite large. Switching to event sampling and trapping DTLB misses (data cache misses), it is observed that the copy from `nos_ptr` to `nos_pixel` involves a large number of cache misses and is the cause of the poor performance. `nos_planestride` is 484704 and `n_chan` is 4, so the `i`-loop makes four large jumps each iteration. If the function were called infrequently, then the cache misses would not be noticeable. However, for the XPS file at hand, there are nearly 2000 calls to `pdf14_compose_group`, each involving a 2×2 block (`width` and `y1-y0` are both 2).

The same experiment was performed on an Intel Core2 Quad Core running Microsoft Windows 7, using the gxps program compiled with Microsoft Visual Studio 2005, and profiling with Intel's VTune (same type of profiler as Shark). The performance is also poor for the very same reason.

The Intel Macintosh version had an additional concern that Shark mentioned. The address calculation for `nos_ptr` involves a multiplication. Shark suggests optimization `-O2` rather than `-Os` (the latter is what Apple recommends for the PowerPC chip). On the Intel Windows version, the compiler does a good job of optimizing the address calculation (no multiply). I modified the code to use as much incrementing of pointers as possible, as shown in the next code.

```
for (y = y0; y < y1; ++y) {
    byte* tosBase = tos_ptr;
    byte* nosBase = nos_ptr;
    for (x = 0; x < width; ++x) {
        byte pix_alpha = alpha;
        byte* tosSrc = tosBase;
        byte* nosSrc = nosBase;
        byte* tosTrg = tos_pixel;
```

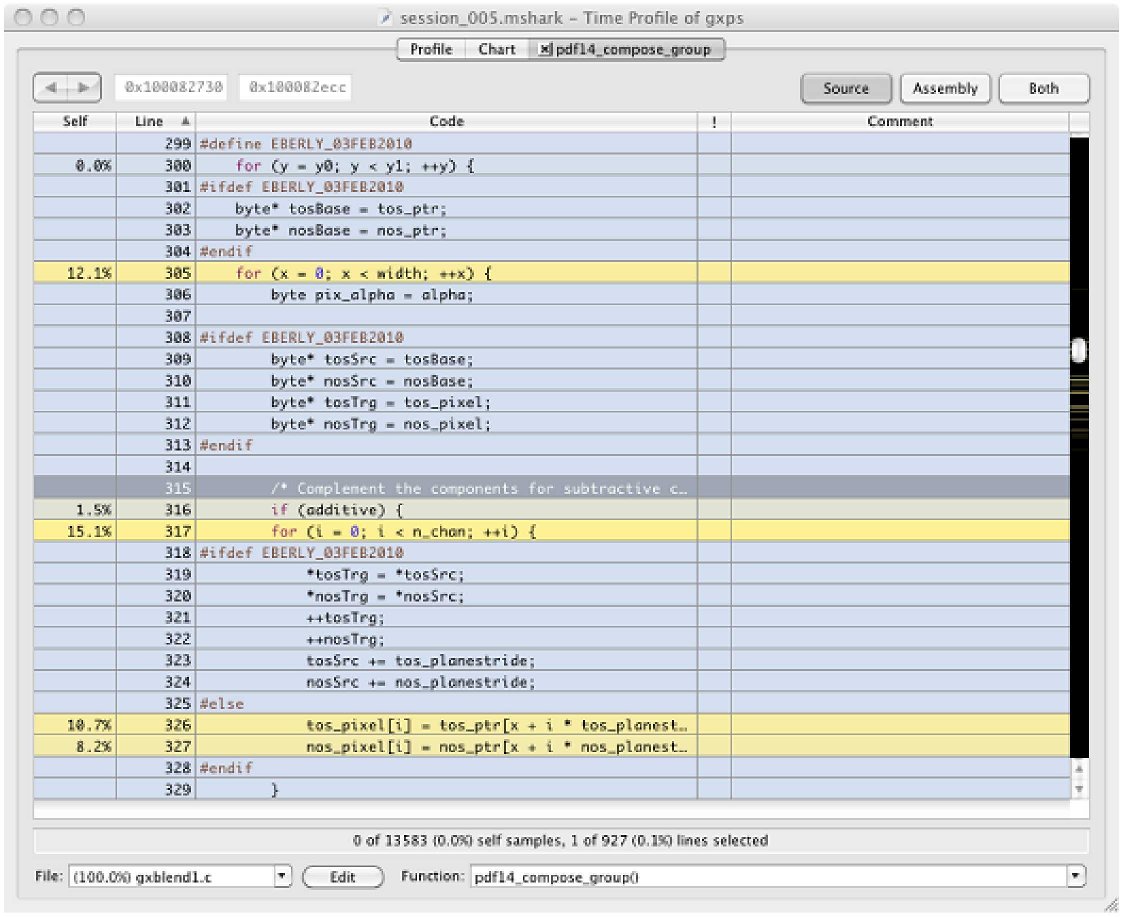
```
byte* nosTrg = nos_pixel;
if (additive) {
    for (i = 0; i < n_chan; ++i) {
         *tosTrg = *tosSrc;
         *nosTrg = *nosSrc;
         ++tosTrg;
         ++nosTrg;
         tosSrc += tos_planestride;
         nosSrc += nos_planestride;
    }
}
<other code>
```

This helped improve performance regarding address calculations, but of course the cache-miss bottleneck cannot be removed in this manner. The first four entries of the time-profile summary became

```
- 70.8%, pdf14_compose_group, gxps
- 18.2%, art_pdf_composite_group_8, gxps
- 5.0%, __bzero, libSystem.B.dylib
- 3.9%, Smask_Luminosity_Mapping, gxps
```

which shows a reduction in the percent of time spent in the function `pdf14_compose_group`.

For comparison, Figure 1 shows a screen capture from the profiler with the current code enabled.



Self	Line	Code	!	Comment
	299	`#define EBERLY_03FEB2010`		
0.0%	300	`    for (y = y0; y < y1; ++y) {`		
	301	`#ifdef EBERLY_03FEB2010`		
	302	`    byte* tosBase = tos_ptr;`		
	303	`    byte* nosBase = nos_ptr;`		
	304	`#endif`		
12.1%	305	`    for (x = 0; x < width; ++x) {`		
	306	`        byte pix_alpha = alpha;`		
	307			
	308	`#ifdef EBERLY_03FEB2010`		
	309	`        byte* tosSrc = tosBase;`		
	310	`        byte* nosSrc = nosBase;`		
	311	`        byte* tosTrg = tos_pixel;`		
	312	`        byte* nosTrg = nos_pixel;`		
	313	`#endif`		
	314			
	315	`        /* Complement the components for subtractive c…`		
1.5%	316	`        if (additive) {`		
15.1%	317	`        for (i = 0; i < n_chan; ++i) {`		
	318	`#ifdef EBERLY_03FEB2010`		
	319	`            *tosTrg = *tosSrc;`		
	320	`            *nosTrg = *nosSrc;`		
	321	`            ++tosTrg;`		
	322	`            ++nosTrg;`		
	323	`            tosSrc += tos_planestride;`		
	324	`            nosSrc += nos_planestride;`		
	325	`#else`		
10.7%	326	`            tos_pixel[i] = tos_ptr[x + i * tos_planest…`		
8.2%	327	`            nos_pixel[i] = nos_ptr[x + i * nos_planest…`		
	328	`#endif`		
	329	`        }`		

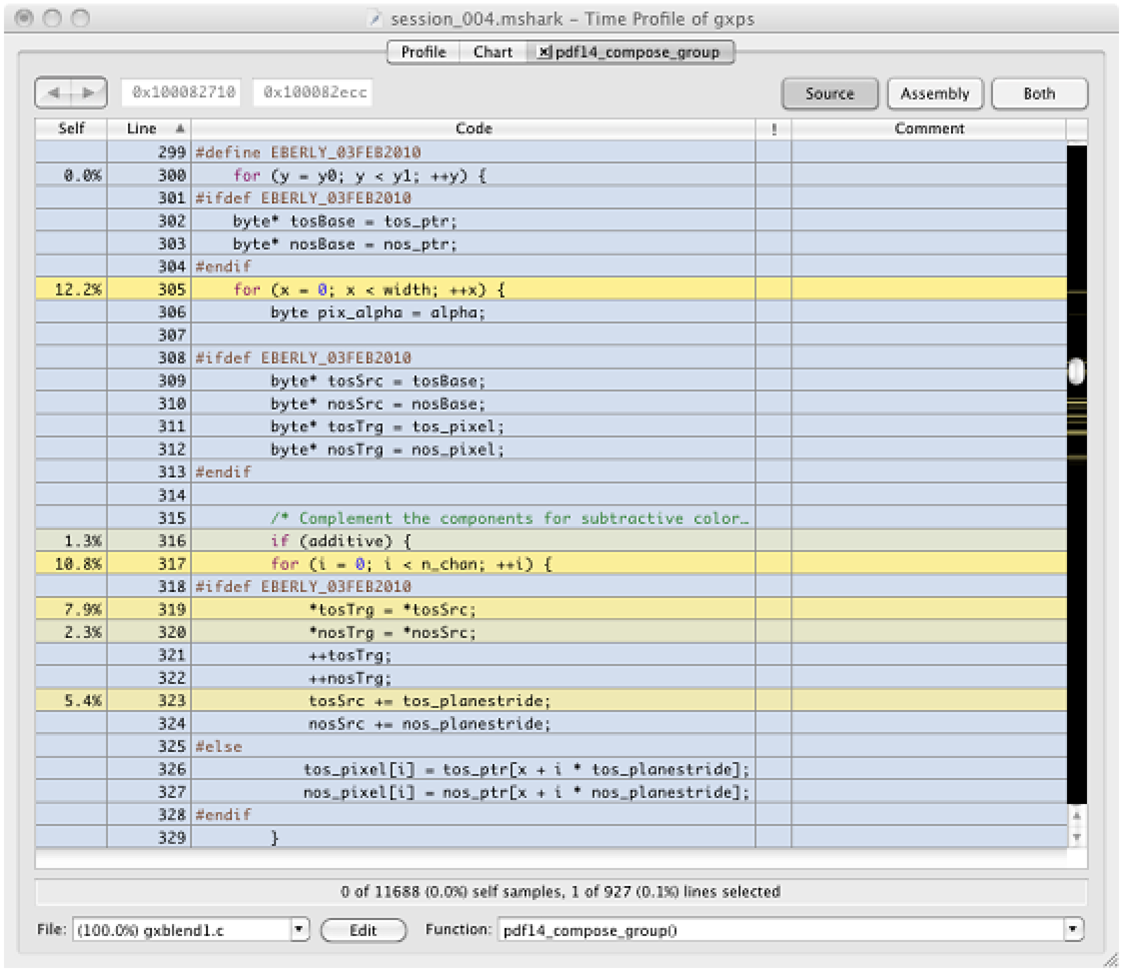
0 of 13583 (0.0%) self samples, 1 of 927 (0.1%) lines selected

File: (100.0%) gxblend1.c

Function: pdf14_compose_group()

Figure 1

Figure 2 shows a screen capture with the modified pointer address calculations.


session_004.mshark – Time Profile of gxps

```
Self    Code
        #define EBERLY_03FEB2010
0.0%        for (y = y0; y < y1; ++y) {
        #ifdef EBERLY_03FEB2010
            byte* tosBase = tos_ptr;
            byte* nosBase = nos_ptr;
        #endif
12.2%       for (x = 0; x < width; ++x) {
                byte pix_alpha = alpha;

        #ifdef EBERLY_03FEB2010
                byte* tosSrc = tosBase;
                byte* nosSrc = nosBase;
                byte* tosTrg = tos_pixel;
                byte* nosTrg = nos_pixel;
        #endif

                /* Complement the components for subtractive color…
1.3%            if (additive) {
10.8%           for (i = 0; i < n_chan; ++i) {
        #ifdef EBERLY_03FEB2010
7.9%                *tosTrg = *tosSrc;
2.3%                *nosTrg = *nosSrc;
                    ++tosTrg;
                    ++nosTrg;
5.4%                tosSrc += tos_planestride;
                    nosSrc += nos_planestride;
        #else
                    tos_pixel[i] = tos_ptr[x + i * tos_planestride];
                    nos_pixel[i] = nos_ptr[x + i * nos_planestride];
        #endif
                }
```

0 of 11688 (0.0%) self samples, 1 of 927 (0.1%) lines selected

File: (100.0%) gxblend1.c   Edit   Function: pdf14_compose_group()

Figure 2

The reduction in time is apparent for the address calculations.

One possible solution to eliminate the bottleneck involves adding assembly code that sets cache hints for the `nos_ptr` data with the hope that the data is copied to cache in time for when the `i`-loop needs it. Naturally, the assembly code is conditionally compiled for each platform.

The more deep-rooted problem, though, is the copy to/from *chunks* (tiled images) and *planar images* (row-major order in linear memory). A solution that avoids conversions between the two storage types is preferable, but will require more code changes than adding the cache hints. That said, the cache hints are not guaranteed to work–it might not be possible to provide them soon enough, in which case the code will stall as it has waiting for cache to be updated.