Efficient Compositing Strategies for Automotive HMI Systems

Simon Gansel*, Stephan Schnitzer†, Riccardo Cecolin*,
Frank Dürr†, Kurt Rothermel† and Christian Maihöfer*
*System Architecture and Platforms Department, Mercedes-Benz Cars Division, Daimler AG, Germany
Email: firstname.lastname <at> daimler.com
†Institute of Parallel and Distributed Systems, University of Stuttgart, Germany
Email: lastname <at> ipvs.uni-stuttgart.de

***Abstract*—The relevance of graphical functions in vehicular applications has increased significantly during the last years. Modern cars are equipped with multiple displays used by different applications such as speedometer, navigation system, or media players. The rendered output of the applications is stored in so-called off-screen buffers and then bitblitted to the screen buffer at the respective window sizes and positions. To guarantee the visibility of the potentially overlapping windows, the compositing has to match the z-order of the windows. To this end, two common compositing strategies *Tile compositing* and *Full compositing* are used, each having performance issues depending on how windows overlap. Since automotive embedded platforms are restricted in power consumption, installation space, and hardware cost, their performance is limited which effectuates the need for highly efficient bitblitting. In order to increase the performance in compositing the windows, we propose *Hybrid Compositing* which predicts the required bitblitting time and chooses the most efficient strategy for each pair of overlapping windows. Using various scenarios we show that our approach is faster than the other strategies. In addition, we propose *Cache-Hybrid Compositing* which reduces the CPU execution time of our approach by up to 66 %. In case of an automotive scenario we show that our optimized approach saves up to 51% bitblitting time compared to existing approaches.**

I. Introduction

Innovations in cars are mainly driven by electronics and software today [1]. In particular, graphical functions and applications enjoy growing popularity as shown by the increasing number of displays integrated into cars. For instance, the head unit (HU)—the main electronic control unit (ECU) of the infotainment system—uses the center console screen to display the navigation system, or displays integrated into the headrests of the front seats together with the center console screen to display multimedia content. Displays connected to the instrument cluster (IC) replace analog indicators displaying speed information or warnings. Additionally, head-up displays are used for displaying navigation instructions or assistance messages on the windshield.

As demonstrated by advanced use cases already implemented in concept cars [2], there is a trend to *share* the different available displays *flexibly* by displaying content from different applications on dynamically defined display areas. For instance, while parking, applications can output information on any display, including, in particular, the IC display. For example, this allows for playing full-screen videos on the IC display while the car is not moving. Moreover, window sizes and positions can change dynamically, e.g., to reduce the size of the speedometer in favor of a larger display area of the navigation software. To support these use cases, a suitable graphical window system is required which allows for efficient compositing of windows whose parameters change during runtime.

The applications typically render their content into off-screen buffers. The graphical system (e.g., X11) then copies the content of the off-screen buffers into the screen buffer, which updates the screen. This specific copy operation is called *bitblitting* and typically performed by dedicated 2D hardware components. Automotive embedded platforms are restricted in power consumption, installation space, and hardware cost. Thus, the available performance is limited which effectuates the need for highly efficient bitblitting. The bitblitting time depends on the number of bitblit operations and the respective areas' sizes. Bitblitting the content of an off-screen buffer into the screen buffer requires GPU processing which common graphical window systems apply by using either a *Full Compositing* strategy or a *Tile Compositing* strategy. *Full Compositing* means that the whole content of each window will be bitblitted in z-order into the screen buffer which causes the system to overwrite overlapped parts of windows multiple times. In contrast, *Tile Compositing* cuts the overlapped windows according to their visibility in so-called tiles and bitblits only the visible tiles of overlapped windows. Thus, *Full Compositing* minimizes the number of bitblitting operations at the cost of a higher amount of copied data, whereas *Tile Compositing* minimizes the total amount of copied data at the cost of more bitblitting operations. In general, neither of these approaches is optimal, since both, the amount of copied data, and the number of bitblitting operations, affects the bitblitting time.

To choose an efficient strategy, we created a model that accurately predicts the execution time of bitblitting commands for a given hardware platform. Using this prediction model we propose *Hybrid Compositing*, which chooses an efficient sequence of bitblitting commands between the two extremes *Full Compositing* and *Tile Compositing*. Although our approach reduces the bitblitting time, the CPU overhead to determine the sequence of bitblitting commands increases. To compensate this, we additionally propose *Cache-Hybrid Compositing* which caches already determined optimized combinations of windows and tiles. We implemented and evaluated the different compositing strategies using random and automotive scenarios

Published in 9th IEEE International Symposium on Industrial Embedded Systems (SIES), 2015, pp.1-10, 18-20 June 2015

http://dx.doi.org/10.1109/SIES.2015.7185036

and show that our concepts outperform existing approaches.

Our contributions in this paper are (1) a prediction model for the bitblitting time of command sequences, (2) a hybrid compositing approach, (3) optimized hybrid compositing using caching, and (4) evaluation results for different scenarios.

The rest of this paper is structured as follows. In Sec. II we present our system model. In Sec. III we describe the Full and Tile Compositing approaches and explain our concept in Sec. IV. We explain our implementation and evaluation results in Sec. V. In Sec. VI we discuss related work and conclude with a summary in Sec. VII.

II. System Model

In Fig. 1 we depict the system model of a graphical system as used in modern HMI systems.

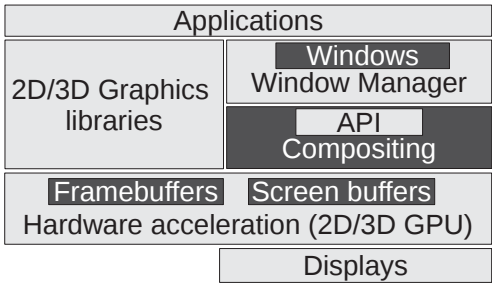

Figure 1. System Model

Applications An application is a process which uses *graphics libraries* to render 2D or 3D content. Before the graphical content of an application can be displayed on a screen, it needs to interact with a *Window Manager* to create a *window*.

2D/3D graphics libraries Each window is mapped to a dedicated *framebuffer*, i.e., an off-screen buffer used as render target. The graphics libraries provide 2D and 3D APIs to render content into framebuffers. Each time a frame is rendered into a framebuffer the *compositing* is informed.

Window Manager The *Window Manager* provides an API for applications to create, delete, or modify windows. A window is a rectangular area which is unambiguously identified by its location on screen given by x, y, and z coordinates and its size given as width and height. Additionally, each window has a unique ID to identify the window.

Compositing API By using the *compositing API* the Window Manager triggers the requests $insert$, $remove$, and $modify$ to inform the compositor about window changes. The request $mark$ is issued by the graphics libraries to notify about a framebuffer update. The request $compose$ triggers the update of the *screen buffers* with the content of all updated windows which were marked by the graphics libraries. Next, we describe the five requests in more detail.

Insert a window: The Window Manager calls the request $insert(WindowId$ wid$, Rect$ r$)$ to add a rectangle r which represents a newly created window with WindowId wid to the set of rectangles in the compositor. The rectangle r can overlap existing rectangles but can also be overlapped by them and therefore has to be inserted by the compositor in correct order, i.e., in z-order.

Remove a window: Removing a window also requires a request $remove(WindowId$ wid$)$ sent by the Window Manager using the window id wid of the deleted window.

Modify a window: Each time a window is modified in size or position the Window Manager sends the request $modify(WindowId$ wid$, Rect$ r$)$ to the compositor containing the window id wid and the new rectangle r.

Mark a window: Using the request $mark(WindowId$ wid$)$ the graphics libraries notify the compositor about an update of a window which requires an update of the associated screen buffer.

Compose: The request $compose()$ triggers the compositor to update the screen buffers with the graphical content of the marked windows.

Compositing The Compositing layer is responsible for bitblitting the framebuffer contents of the applications into the mapped window locations of the target screen buffer by using *GPU hardware acceleration*. Compositing consists of two steps, namely, the CPU execution of the compositing algorithm that determines the required bitblitting commands, and the execution of the bitblitting commands on the GPU. Thus, the compositing time t_{comp} consists of the CPU execution time t_{CPU} and the bitblitting time t_{BB}. Formally, we define $t_{BB} = c_{commit} + \sum_{fb \in FB} \left(c + t_{bitblit}(fb)\right)$, where c_{commit} is the constant overhead of flushing a command batch, c the overhead for each operation inside a batch, and $t_{bitblit}(fb)$ the execution time depending on the dimensions of fb. The target of this work is to minimize $t_{comp} = t_{CPU} + t_{BB}$.

GPU hardware acceleration Hardware accelerated GPUs are used to render the 2D/3D content into the framebuffers on behalf of the applications by using the graphics libraries. In addition, GPUs can efficiently bitblit the framebuffer contents into the screen buffers and display the screen buffers on screen triggered by the compositing layer. A bitblit operation can either be used to fully bitblit a framebuffer or only a rectangular part of it into the screen buffer. Thus, the bitblit operation directly operates on the memory of the GPU.

III. Background

The two strategies *Full Compositing* and *Tile Compositing* both are commonly used in graphical window systems. Since we compare our approaches with these two strategies we describe them in more detail in the following.

1) Full Compositing: The simplest way to update a screen buffer is to bitblit the framebuffer fully beginning with the lowermost up to the topmost window, which is why this is also called the painter's algorithm [3]. The algorithm is depicted in List. 1.

```
1  function compose_full (w_list, marks)
2    win = w_list.bottom
3    while (win)
4      if(marks[win.id])
5        bitblit(win.x, win.y, win.w, win.h, win.fb)
6        mark_dependencies(win.id, marks)
7      end if
8      win = win.next
9    end while
10 end function
```

Listing 1. Full compositing algorithm

The function $compose\_full$ gets as input a list (w_list) of the windows and an array that contains all marked windows. The algorithm starts with the lowermost window (line 2) and iterates through the list of windows (line 3). If a window is marked (line 4) then the algorithm bitblits (line 5) the according framebuffer for the given window location and size. Then, all overlapping windows are also marked (line 6) and the algorithm continues with the next window. While the number of bitblit operations is minimal, areas that are written more than once can significantly decrease performance.

2) Tile Compositing: Bitblitting only the visible parts of a window requires the tiling of the window in its visible rectangles, called tiles. Each rectangle which is overlapped by another rectangle is split into tiles which represent the visible rectangular areas of the rectangle. The *Tile Compositing* strategy divides overlapped windows into multiples tiles where the set of all its tiles represents exactly all parts of the window which are not overlapped. The *Tile Compositing* algorithm is depicted in List. 2.

```
1  function compose_tile (w_list, marks)
2    win = w_list.bottom
3    while (win)
4      if(marks[win.id])
5        bitblit_tile(win.tiles, win.fb)
6      end if
7      win = win.next
8    end while
9  end function
10
11 function bitblit_tile (tiles, fb)
12   for (each t in tiles)
13     if(t.visible)
14       bitblit(t.x, t,y, t.w, t.h, fb)
15     else
16       bitblit_tile (t.tiles, fb)
17     end if
18   end for
19 end function
```

Listing 2. Tile compositing algorithm

The function $compose\_tile$ also gets a list w_list which contains the windows with the respective tiles and the array $marks$ which contains the marked windows. Beginning from the lowermost window the algorithm checks for each window if it is marked (line 4) and calls the function $bitblit\_tile$ (line 5) with the tiles of the window and the according framebuffer. The function $bitblit\_tile$ (line 11) checks for each tile if it is visible or if the tile is also covered by other windows. If the tile is visible then it is bitblitted (line 14) else the function is recursively called with the visible tiles of the tile. In Fig. 2 we depict an example to compare *Full* and *Tile Compositing* for a simple setup where Rectangle B partially overlaps Rectangle A.

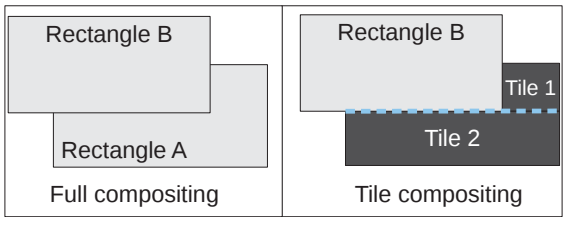

Figure 2. Overlapping windows and resulting tiles

The *Full Compositing* strategy first bitblitts Rectangle A, then Rectangle B into the screen buffer (cf., left side of Fig. 2). This implies that a part of Rectangle A is overdrawn by Rectangle B. In contrast, the *Tile Compositing* strategy splits up Rectangle A into the two tiles Tile 1 and Tile 2 by using a horizontal line (cf., right side of Fig. 2). The combined area of Tile 1 and Tile 2 represents those areas of Rectangle A which are not hidden behind Rectangle B.

If larger amounts of window areas overlap, the bitblitting time of *Full Compositing* significantly suffers from unnecessary copying. On the other hand, for *Tile Compositing* we observe that adding a window which overlaps other windows can create at most four additional tiles, where *Full Compositing* would create only one. Hence, the *Tile Compositing* strategy can result in the scale of up to 400 % of the number of bitblitting operations compared to the *Full Compositing* strategy. Since each bitblitting operation increases the bitblitting time by a constant, a higher number of rectangles can also be inefficient and wastes bitblitting time. For instance, the bitblitting of a rectangle of size 1000×1000 pixels requires less bitblitting time than the bitblitting of two rectangles of size 1000×500 pixels. From this insight, it follows that—depending on the window positions and sizes—neither *Full Compositing* nor *Tile Compositing* are optimal.

IV. Compositing Concepts

In this section we describe the concepts of our bitblitting strategy. We first briefly explain our compositing strategy, followed by our compositing architecture and a description of the relevant components. Then, we present the API used by the applications to manage their windows. Finally, we describe the algorithms for our *Hybrid Compositing* strategy and the *Cache-Hybrid Compositing* strategy.

A. Overview

The existing strategies *Full Compositing* and *Tile Compositing* provide advantages and disadvantages in efficiently bitblitting windows. The *Full Compositing* strategy requires only a minimal number of bitblit operations to update the windows but bitblits overlapping window areas multiple times which can increase the bitblitting time significantly depending on the size of the overlapping window areas. In contrast, the *Tile Compositing* strategy bitblits only the minimal visible window area but requires up to about 300 % more bitblitting operations to bitblit the visible tiles. This causes overhead and also increases the bitblitting time significantly depending on how windows overlap. For many scenarios, neither *Full* nor *Tile Compositing* is optimal, since the bitblitting time increases with both, the number of bitblit operations, and the amount of pixels that are bitblitted.

We propose the *Hybrid Compositing* strategy which searches for a combination of bitblitted rectangles that reduces the bitblitting time. To choose such a combination, the execution time of a bitblit command must be accurately estimated in advance. We created a prediction model that estimates the bitblitting time depending on the windows sizes' and the number of bitblit operations. This prediction model is calibrated for the target hardware platform.

For our *Hybrid Compositing* we use an efficient data structure which stores the relevant information necessary for our *Hybrid Compositing* strategy and optimizes the bitblitting time for screen buffer updates. Since the CPU execution time, needed to search for a good combination of bitblitted rectangles, is significantly higher than the CPU execution time of *Full* and *Tile Compositing*, we additionally propose *Cache-Hybrid Compositing*. This further optimized hybrid strategy saves already calculated knowledge about good combinations of rectangles in a cache, thus providing a speedup for future decisions.

B. Compositing architecture

For framebuffer compositing we propose the architecture depicted in Fig. 3. Next, we present the main architecture components of Fig. 3 and their interaction.

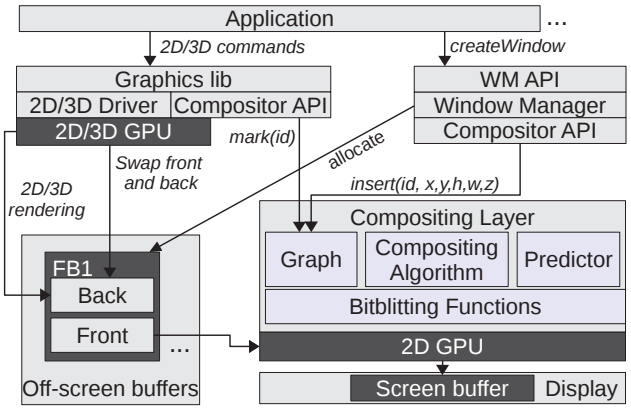

Figure 3. Compositing architecture

1) Applications: As presented in Sec. II, applications use graphics libraries like OpenGL to render into their framebuffers. Each time an application has completed the rendering of a frame it eventually calls a swap command (e.g., eglSwapBuffers) which makes the new content available in the front buffer from where it is eventually bitblitted to the screen buffer. Additionally, the swap command *marks* the window at the compositing layer to notify about the required refresh on the screen buffer and additionally blocks the application until the screen buffer has been updated.

2) Window Manager: In order to create a window, an application initially calls *createWindow* on the Window Manager. The Window Manager registers with the request *insert* the window at the compositing layer and *allocates* the respective framebuffers. Deleting a window can be performed by the request *delete*.

3) Compositing: The information about windows and tiles is stored in the *Graph* data structure that holds the set of all windows in z-order, their respective sizes, positions, and status. Each time the window manager adds, modifies, or removes a window, the tiling algorithm described in Sec. IV-C is executed which generates the new tiles. The prediction model is initially calibrated for the target hardware. The compositing algorithm executes one of the four compositing strategies described in Sec. IV-D. In case of the *Hybrid Compositing* and *Cache-Hybrid Compositing* strategies the compositor uses predicted bitblitting times which are determined using the *Predictor*.

C. Tiling concepts

Next, we propose a tiling algorithm which efficiently generates tiles of overlapped rectangles. We first classify all cases of two overlapping rectangles and unify them to four cases.

In [4] 17 different cases of two overlapping rectangles are identified. According to the number of covered edges, these 17 cases are mapped to five equivalence classes. We ignore the equivalence class with no covered edges, since both, *Full* and *Tile Compositing* would effectively be equivalent, making the optimization choice trivial.

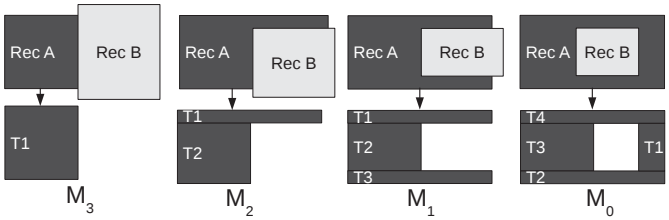

Figure 4. Four cases of overlapping rectangles and resulting tiles

As depicted in Fig. 4, in M_3 three edges of $Rec\ A$ are overlapped by $Rec\ B$, which produces one tile, only. On the other side, for M_0, where no edge is overlapped, four tiles are produced.

Using the four cases M_0 to M_3, we propose the tiling algorithm depicted in List. 3 which generates the visible tiles in case of overlapping windows.

```
1  function tiling (windows) // in z-order
2    while (not windows.isempty)
3      win = windows.pop
4      rectangles.push(win)
5      for each w in windows do
6        r_tmp.clear
7        for each r in rectangles do
8          if (not overlap(w, r))
9            r_tmp.push(r)
10         else
11           r_tmp.push(cut_new_tiles(w, r))
12       end do;
13       rectangles = r_tmp
14     end for
15     resultlist.append({win,rectangles})
16     rectangles.clear
17   end while
18   return resultlist // windows with visible tiles
19 end function
```

Listing 3. Tiling algorithm

The function $tiling(rectangles)$ iterates over a list of z-ordered windows and calculates the visible tiles for each of the windows. The algorithm begins with the lowermost window and pushes it in the rectangle list $rectangle$ (c.f., List. 3 line 3 to 4). Then, the algorithm iterates over all remaining windows (c.f., List. 3 line 5) and checks for each rectangle in list $rectangle$ if it is overlapped by a window or not (c.f., List. 3 line 8 to 11). If it is not overlapped the rectangle is stored in a temporary list r_tmp. In case the rectangle is overlapped by a window the function $cut\_new\_tiles(t, v)$ determines the visible tiles according to the four cases in Fig. 4 and stores these tiles in the temporary list r_tmp. Hence, a visible tile which is overlapped by a window is further subdivided into

visible tiles. After each rectangle in $rectangles$ is checked the list is overwritten by the list r_tmp which contains the subdivided visible tiles. Then, the algorithm checks if the next window of the list $windows$ overlaps a rectangle of the list $rectangles$. Finally, for each window the list $resultlist$ stores its visible tiles.

D. Compositing Strategies

In this section we describe our *Hybrid Compositing* and the optimized *Cache-Hybrid Compositing* algorithms in detail. Both algorithms require a data structure that contains the information about the windows and the visible tiles which are calculated by the tiling algorithm (c.f. List. 3). Since the CPU execution time t_{CPU} of our compositing heavily depends on the used data structure we next propose a data structure that supports the compositing API described in Sec. II.

1) Data structure: The data structure supports the five compositing API requests $insert$, $remove$, $modify$, $mark$, and $compose$. Since t_{CPU} depends on the number of requests which need to be executed, the design of the data structure is optimized for the requests that will be called most.

Our proposed data structure is based on a graph which stores the information for windows and their visible tiles, as depicted in Fig. 5. The requests $insert$ and $remove$ are called if windows are created or deleted, which happens seldomly. This also holds for the request $modify$, which changes window sizes and positions. In contrast, the requests $mark$ can occur up to once per screen refresh interval (e.g., 60 Hz) and window, since it is called each time an application updates a window content. The request $compose()$ is called typically once per screen refresh interval, given that at least one window is updated with each screen refresh. Thus, we optimized the data structure to store the rectangles and tiles, in order to facilitate their handling and retrieval executing the requests $mark$ and $compose$.

An example of our data structure for five windows is depicted in Fig. 5. For each window we store location, width, and height. In addition, beginning with the lowermost window, we link each window to the next window in z-order (e.g., window $W4$ to $W5$). If a window overlaps another window, the tiling algorithm (c.f. Sec. IV-C) calculates the visible tiles and links them to the corresponding window (e.g., $W3$ to $T6$). Furthermore, each tile which was created by an overlapping window, is added to the *dependency links* of the overlapping window. For instance, window $W1$ is overlapped by $W2$ which results in the tiles $T1$ and $T2$. Thus, $T1$ and $T2$ are linked from $W1$ (arrow line) and *dependency links* from $W2$ are added (dotted line). The same principles apply also to overlapped tiles. For instance, tile $T2$ is overlapped by window $W3$ which results in the visible tiles $T3$, $T4$, and $T5$. Accordingly, these tiles have dependency links to $W3$. Next, we describe the compositing strategies in more detail.

2) Hybrid Compositing: To overcome the shortcomings of *Full Compositing* and *Tile Compositing* we propose the *Hybrid Compositing* strategy which decides at runtime for each window and tile whether it is faster to bitblit it full or only its visible tiles. Each time the request *compose* is called, the compositor bitblits all marked windows. To this end, the algorithm has to walk through the data structure, and—depending on the compositing strategy—use the information about windows and/or tiles to bitblit the corresponding framebuffer contents into the screen buffers.

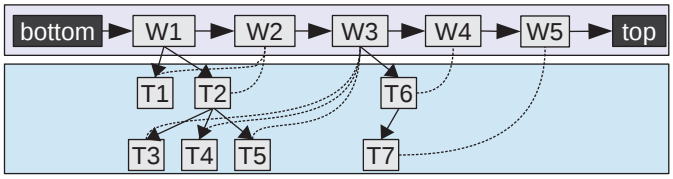

Figure 5. Data structure storing windows, visible tiles, and links

The *Hybrid Compositing* algorithm calculates the predicted bitblitting time of a window and all of its tiles, and uses the faster approach, i.e., *Full Compositing* or *Tile Compositing*. Since we do this optimization at tile granularity instead of window granularity, we often save a significant amount of bitblitting time.

```
1  function compose_hybrid (graph, marks)
2    win = graph.bottom
3    while (win)
4      if(marks[win.id])
5        bitblit_hybrid(win, marks, win.fb)
6      end if
7      win = win.next
8    end while
9  end function
10
11 function bitblit_hybrid (n, marks, fb)
12   if (n.tiles && t_full(n) > t_tiled(n))
13     for (each t in n.tiles)
14       bitblit_hybrid(t, marks, fb)
15     end for
16   else
17     bitblit(n.x, n.y, n.w, n.h, fb)
18     mark_dependencies(win.id, marks)
19   end if
20 end function
21
22 function t_full (n)
23   time_full = predicted_time (n.size)
24   for (each d in n.dependencies)
25     time_full += min(t_full(d), t_tiled(d))
26   end for
27   return time_full
28 end function
29
30 function t_tile (n)
31   time_tile = 0
32   if(n.visible)
33     time_tile = predicted_time (n.size)
34   else
35     for (each t in n.tiles)
36       time_tile += min(t_full(t), t_tiled(t))
37     end for
38   end if
39   return time_tile
40 end function
```

Listing 4. Hybrid Compositing algorithm

The *Hybrid Compositing* algorithm is depicted in List. 4. The function $compose\_hybrid$ walks through the list of windows and checks if a window needs to be bitblitted. For each marked window the function $bitblit\_hybrid$ is called (line 5) which checks if a window overlaps the marked window and if the predicted bitblitting time using *Full Compositing* is higher than using *Tile Compositing* (line 12). If this holds, the function

$bitblit\_hybrid$ is called recursively for all tiles of the marked window (line 14). Otherwise, the algorithm bitblits the marked window and marks all windows that overlap it (lines 17 & 18). The functions t_full and t_tile predict the required bitblitting time for *Full Compositing* and *Tile Compositing*. In t_full the bitblitting time required to bitblit the window (or tile) n is predicted and stored in the variable $time\_full$. Then, for each window that overlaps n—using its dependency links—the bitblitting times of *Full Compositing* and *Tile Compositing* are calculated and the smaller bitblitting time is added to $time\_full$ (line 25). Finally, the sum of all minimal bitblitting times is returned (line 27). The function t_tile is similar to t_full, but additionally checks whether a window or tile is visible or not. If a tile is visible, the predicted bitblitting time is returned (line 33 & 39). Otherwise, for each tile the bitblitting time for *Full Compositing* and *Tile Compositing* is calculated and the smaller bitblitting time added to $time\_tile$ (line 36) which is eventually returned.

3) Cache-Hybrid Compositing: To improve the performance of our *Hybrid Compositing* we introduce a caching strategy that stores the optimized combinations of windows and tiles. The *Cache-Hybrid Compositing* reduces the CPU execution time t_{CPU} and, therefore, also reduces the compositing time t_{comp}. Our evaluations (cf., Sec. V) show that using our cache reduces t_{CPU} by up to 66 %.

The approach to using a cache is driven by the assumption that the positions and sizes of windows seldomly change. Thus, already calculated windows and tile sequences can be reused multiple times. For the addressed automotive scenarios, this is a valid assumption. On the other hand, the frame rates of different windows might vary a lot. The frame rates directly influence the set of marked windows of each frame. Thus, our cache uses the set of marked windows as keys and the already calculated tiling sequences as values. In the rare case where windows change their positions or sizes, the cache is invalidated.

```
1  function compose_cache_hybrid (graph, marks)
2    key = HASH(marks)
3    if(match(cache[key], marks))
4      for (each r in cache[key])
5        bitblit(r.x, r.y, r.w, r.h, r.fb)
6      end for
7    else
8      cache_enable(key, marks)
9      compose_hybrid(graph, marks)
10     cache_disable()
11   end if
12 end function
```

Listing 5. Cache-Hybrid Compositing algorithm

More precisely, the *Cache-Hybrid* algorithm (cf. List. 5) calculates the hash value of the array containing the marked windows and checks whether the set of marked window is in the cache or not (line 3). If the set of windows is in the cache the cached bitblitting sequence is bitblitted (line 5). Otherwise, the cache will be enabled. To enable the cache, the algorithm first inserts a new entry into the hash table and then sets a flag which causes each bitblitted rectangle to be recorded and appended to the new cache entry. For bitblitting, the *Hybrid Compositing* algorithm (cf., List. 4) is used. Finally, the cache will be disabled, which means that the recording mode ends and the new cache entry is available for future requests.

V. Evaluation

Using our compositing architecture described in the previous section, in this section we evaluate the performance of our two compositing strategies. Initially, we present our implementation platform and the results of the prediction model calibration. Thereafter, we present and discuss the evaluation results of the *Hybrid Compositing* and *Cache-Hybrid Compositing* compared to *Full Compositing* and *Tile Compositing* strategies using multiple scenarios. In addition, we compare our *Cache Hybrid Compositing* strategy with optimal tiling.

A. Platform and Measurement Setup

For our evaluation, we used a Freescale i.MX6 SABRE for Automotive Infotainment quad core embedded platform running a fully preemptive 3.10.17 Linux kernel. The platform provides three different GPUs from Vivante [5]. The 3D GPU GC2000 provides OpenGL ES support, the GC320 which provides 2D compositing, and the GC355 supports OpenVG. We used the Vivante "framebuffer" driver for bitblitting.

To measure the compositing time of a compositing strategy, we used the POSIX function $clock\_gettime$. To ensure precise measurements we set the real-time priority of our application to 99 while keeping all other processes at priority 0. We pinned the compositing thread and GPU driver kernel threads to the same CPU core. The compositing time t_{comp} of a bitblitting strategy consists of the CPU execution time t_{CPU} in user-space and the bitblitting time t_{BB} needed by the GPU. To increase our insight, we measured both times separately.

B. Scenarios

To analyze the performance of our approaches compared to existing compositing strategies we defined two types of scenarios which we used to evaluate the performance. Each scenario consists of a set of windows given as rectangles with width, height, and position on screen given by the coordinates x,y, and z. Hence, windows can overlap other windows partially or completely. We first evaluated the performance if *all windows are always marked*, i.e., the set of windows (or tiles) which need to be bitblitted is always the same. Additionally, it is important to note, that in realistic scenarios not all applications update with the same frame rate, which directly influences the bitblitting time. This is due to the fact that it might not be necessary to bitblit parts that did not change. Therefore, we evaluated the influence of the update rate by using for each window a *random mark interval* in the range 1 to 4, i.e., the windows were marked at each, every second, every third, or every forth frame.

To create a **Random Scenario** like the one depicted in Fig. 6, we implemented a generator that randomly selects the width for each of the windows in the range of 10 to 1440 pixels and the height in the range of 10 to 540 pixels. The position is also randomly selected but the window has to be fully contained inside the screen buffer which has a screen resolution of 1440 by 540.

Furthermore, we created an **Automotive Scenario** that represents the application windows visible on a typical instrument cluster as depicted in Fig. 7, representing speedometer, tachometer, indicators, warning and information manager.

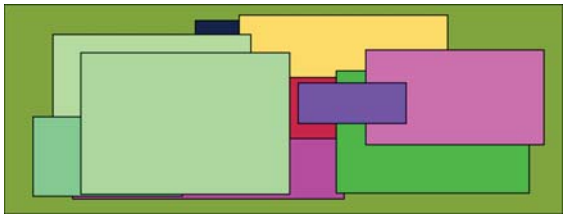

Figure 6. Window distribution, **Random Scenario**

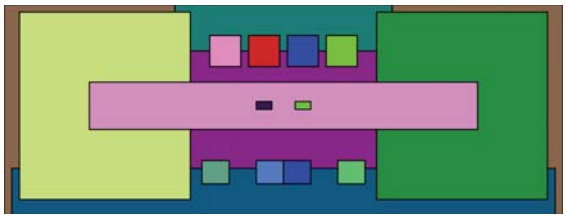

Figure 7. Window distribution, *Automotive scenario*

C. Calibration of the prediction model

In order to choose a good combination of bitblitting commands, our two concepts need a prediction model which has to be calibrated for the given hardware platform. We measured the bitblitting time for a wide range of different rectangle sizes and number of bitblit commands.

As explained in Sec. II, the bitblitting time t_{BB} depends on the number of bitblitting commands n and the respective framebuffer dimensions ($w_{fb} \times h_{fb}$). We therefore measured t_{BB} using a wide range of n=$|FB|$ and all possible combinations of w_{fb} and h_{fb}. In Fig. 8 we depict t_{BB} by n and the

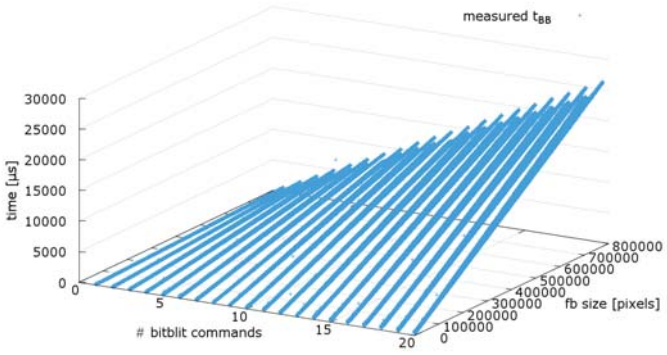

Figure 8. Bitblitting times of multiple bitblit commands

window dimensions $w_{fb} \times h_{fb}$, each point representing the mean value of 100 samples. Using Octave [6] with *bilinear least squares fitting* we derived the prediction function Eq.1.

$$t_{BB} = a + \sum_{fb \in FB} b + c * w_{fb} + d * h_{fb} + e * w_{fb} * h_{fb} \quad \text{(Eq.1)}$$

using a=67.2μs, b=9.03μs, c=1.29e-4μs, d=6.71e-4μs, and e=1.67e-3μs. Note that, although in Fig. 8 we depicted $w_{fb} \times h_{fb}$ as combined value, we actually use w_{fb} and h_{fb} as independent terms in favor of slightly better accuracy—thus achieving an average prediction error of only 0.19%.

D. Evaluation of the compositing strategies

In this section, we present and discuss the evaluation of the four compositing strategies using the setup and scenarios described in Sec. V-A and Sec. V-B.

1) Bitblitting time if all windows are always marked: To compare the different bitblitting strategies we measured the compositing time required to bitblit multiple windows on screen using the four different strategies *Full Compositing*, *Tiled Compositing*, *Hybrid Compositing*, and *Cache-Hybrid Compositing* with the **Random Scenario** and the **Automotive Scenario** (cf. Sec. V-B).

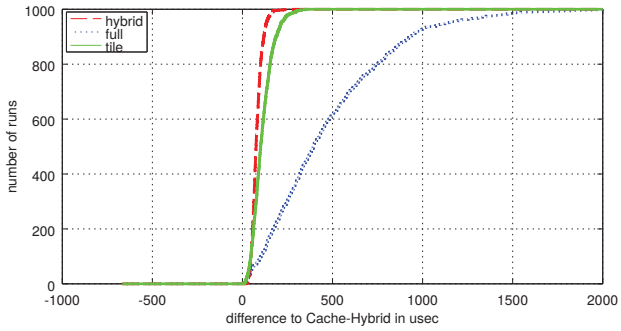

Figure 9. Difference of t_{comp}, **Random Scenario**, all windows always marked

First, we executed 1000 times the **Random Scenario** using between 10 and 17 different windows. For each of the four strategies we bitblitted 2400 frames for each **Random Scenario** and measured the average of t_{comp}. The results are depicted as a cumulative distribution function in Fig. 9. The y-axis represents the number of runs and the x-axis represents the difference of the compositing times of the strategies in μs. The three lines depict the differences of the compositing time for the three strategies *Full Compositing* (blue dotted line), *Tile Compositing* (green line) and *Hybrid Compositing* (red dashed line) compared to the *Cache-Hybrid Compositing* strategy. It can be observed that in all cases the *Cache-Hybrid Compositing* is more efficient in bitblitting a scenario than the other strategies. The improvement we achieved in these 1000 scenarios with our *Cache-Hybrid Compositing* was in average 27 % less compositing time compared to *Full Compositing* which is with 474 μs fairly good. Compared to the *Tile Compositing* we saved in average 8 % of the compositing time. Since the *Cache-Hybrid Compositing* primarily optimizes the compositing time required to calculate the optimal bitblitting we also achieved a noticeable improvement of our *Hybrid compositing* strategy with 6 % less compositing time in average.

We executed the **Automotive Scenario** 10000 times and marked all 17 windows in every frame. We depicted the average compositing time for each of the strategies in Fig. 10. The y-axis represents the number of runs and the x-axis represents compositing times of the strategies in μs.

It can be observed that the *Hybrid Compositing* is about 44% (or 1570 μs) faster than the *Full Compositing* and about 11% (or 243 μs) faster than the *Tile Compositing*. Our *Cache-Hybrid Compositing* improves the performance about 5 % in average compared to the *Hybrid Compositing*.

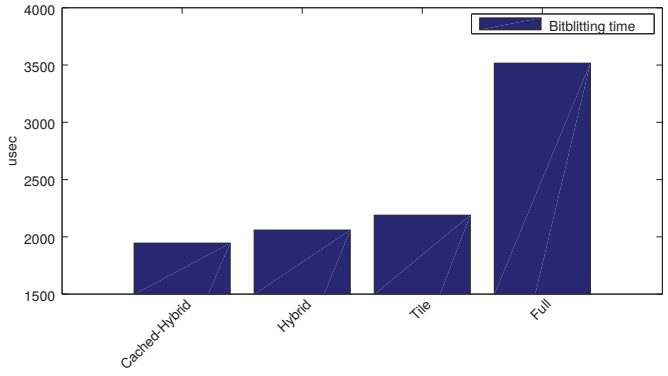

Figure 10. Difference of t_{comp}, **Automotive Scenario**, all windows always marked

2) Bitblitting time using random mark intervals: In a second simulation, we randomly assigned for each window the interval in which it is marked (cf., Sec. V-A). We executed the **Random Scenario** and the **Automotive Scenario** each 1000 times for 2400 frames. We kept the set of marked windows fixed for 20 frames and calculated the average of t_{comp} of these 20 frames. The results are depicted as a cumulative distribution function in Fig. 11 and Fig. 12. Similar to Fig. 9 the y-axis represents the number of runs and the x-axis represents the difference of the composting times of the strategies in μs.

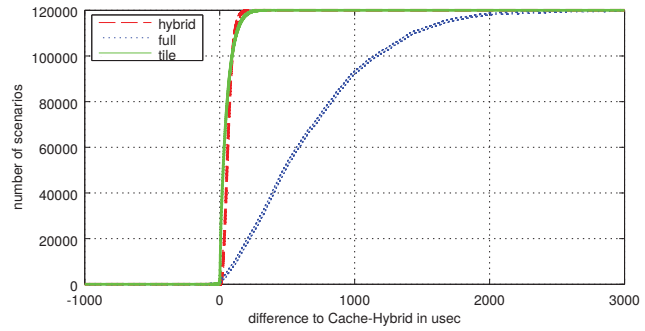

Figure 11. Difference of t_{comp}, **Random Scenario**, random mark interval

As depicted in Fig. 11 the compositing using *Cache-Hybrid Compositing* is in average 46 % (683 μs) faster than the *Full Compositing*. The bitblitting time of our *Hybrid Compositing* is in all cases better or as good as the bitblitting time of the *Tile Compositing*. Thus, in average we achieved an improvement at the compositing time of only 1 %. This is due to the higher CPU execution time of our approach which—in some cases—leads to a higher compositing time compared to *Tile Compositing*. This drawback is overcome by our *Cache-Hybrid Compositing* which outperforms the other approaches in all cases and is in average about 7 % faster than *Tile Compositing*.

As depicted in Fig. 12, in the **Automotive Scenario** the *Full Compositing* strategy (blue dotted line) is less efficient than the other strategies. The compositing time of the *Full Compositing* strategy is in average 1183 μs—about 51 %—higher than the compositing time of the *Cache-Hybrid Compositing* strategy. Similar to the **Random Scenario** the bitblitting time of our *Hybrid Compositing* is in all cases better or as good as the bitblitting time of the *Tile Compositing*. However,

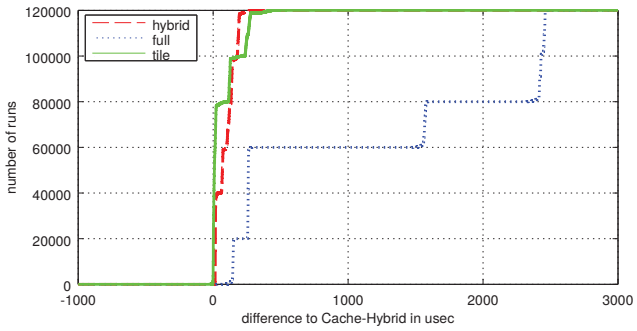

Figure 12. Difference of t_{comp}, **Automotive Scenario**, random mark interval

the compositing of the *Hybrid Compositing* is in average only 1 % faster than the *Tile Compositing*. Since our approach requires more CPU execution time than the *Tile Compositing*, in some cases the compositing time of our *Hybrid Compositing* is higher. Again, our *Cache-Hybrid Compositing* overcomes this drawback since it outperforms the other approaches in all cases and achieves an improvement of about 6 % in average compared to the *Tile Compositing*.

3) Optimal tiling: The performance of our *Hybrid Compositing* depends on the tiling algorithm (List. 3) which generates the visible tiles. While our algorithm is very fast, it is not always optimal, since not all possible combinations of tilings are considered. For instance, it starts traversing tiles always with the uppermost rectangle, whereas other starting points are ignored. In order to evaluate our algorithm, we measured both, the bitblitting time and the CPU execution time of the algorithm, in comparison to mathematically optimal tiling. To this end, we brute force all combinations of tiles to determine the optimal tiling. The number of combinations exponentially depends on the number of overlapping tiles. For instance, the maximum number of combinations for two overlapping tiles is the permutation of the four possible cutting edges (cf. Fig. 2 case M_0) which makes 24 different cases of cutting and leads to 16 combinations of different sets of tiles. We evaluated for different scenarios the optimal set of tiles and compared the predicted bitblitting time to that of our tiling algorithm.

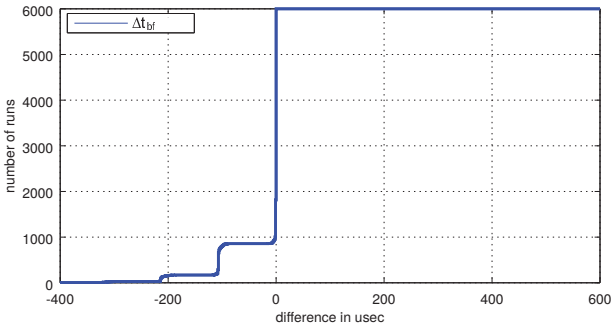

Figure 13. Difference of t_{BB} between optimal tiling and our tiling algorithm

We evaluated for 2 to 7 windows in 1000 randomly generated scenarios the optimal bitblitting time (t_{bf}) and the bitblitting time (t_{ta}) using our tiling algorithm. The results are depicted in Fig. 13. The y-axis represents the number of runs and the x-axis the difference of the minimal bitblitting times

t_{bf} to the bitblitting time t_{ta} in μs as cumulative distribution function. Thus, the blue line represents $\Delta t_{bf} = t_{ta} - t_{bf}$.

It can be observed that in about 17% of all runs our tiling algorithm is not optimal. However, in average the bitblitting time is only about 1% less than using our tiling algorithm. On the other hand, the computational effort for brute force was many orders of magnitude higher than with our approach. For instance, to calculate the optimal tiling for one set of five windows takes about 10^6 times longer.

E. CPU execution time

We evaluated the CPU execution time t_{CPU} for all four strategies using the **Automotive Scenario** (as $Auto\_X$ and $Auto\_C$) and the **Random Scenario** (as $Rand\_X$ and $Rand\_C$) described in Sec. V-B. For each of the scenarios we evaluated with all windows always marked (as $Auto\_C$ and $Rand\_C$) as well as randomly assigned mark intervals (as $Auto\_X$ and $Rand\_X$). In Fig. 14 the average of 1000000 runs is depicted, respectively.

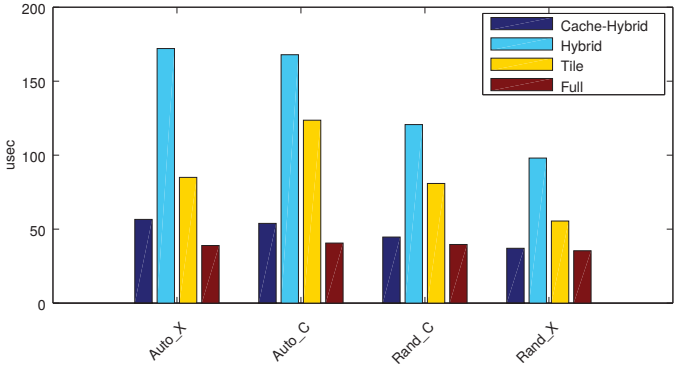

Figure 14. CPU execution time of the four strategies

As expected, the *Full Compositing* strategy requires less CPU execution time than the other strategies since it simply walks through the list of windows. The *Tile Compositing* strategy walks through the list of tiles, which is typically longer than the list of windows and therefore more CPU intensive. Since the *Hybrid Compositing* strategy additionally predicts the bitblitting time for windows and tiles it has the highest CPU execution time. However, we observe that using our cache significantly reduces the CPU execution time and is efficient even compared to *Tile* and *Full Compositing*.

We summarize and conclude that the bitblitting time of our *Hybrid Compositing* strategy outperforms existing approaches in realistic scenarios. Although our *Hybrid Compositing* had in some cases a higher CPU execution time compared to other approaches, this drawback is overcome by our *Cache-Hybrid Compositing* strategy which always outperformed existing approaches in our evaluations. Compared to optimal tiling, we showed that our approach consumes only 1% less bitblitting time, but outperforms optimal tiling by saving orders of magnitude of CPU execution time t_{CPU}.

VI. Related Work

Since the invention of raster graphics a broad variety of graphics window systems emerged that manage the graphical output of applications and compose them on screens.

Meyrowitz and Moser [7] developed a window manager called BRUWIN which focuses on the adaptability of the graphical system to a variety of devices and operating systems. They propose a display manager that is responsible for the compositing of z-ordered windows. They propose to use a *Full Compositing* approach which skips all windows that are fully overlapped. From this it follows that for partly overlapped windows, BRUWIN uses expensive *Full Compositing*.

Myers [8, 9] developed a window manager called Sapphire (the Screen Allocation Package Providing Helpful Icons and Rectangular Environments) that supports rectangular windows which can overlap each other. Sapphire also imposes a total z-ordering of all windows. In [10] they propose to hold the visible part of windows on the screen buffer and store in the off-screen buffers only the covered parts to minimize the required memory. Sapphires maintains a list of rectangles for each window that contains the visible parts which is similar to the *Graph* data structure used for our concepts. However, since Sapphire does not store the complete content of windows in off-screen buffers, shifting windows often requires applications to costly redraw their screen content, potentially making short-time artifacts occur on the screen.

Pike et al. [11] developed a data structure for storing the covered parts of a window (called layer by the Authors) to reduce the window update latency. The fully visible windows or visible parts will not be stored there, since their content is stored in the screen buffer. Thus, they extended the domain of the bitmap operator *bitblt*–defined in [12]–to be able to include wholly or partly covered windows. Basically, they propose *Tile Compositing* where visible tiles are only kept in the screen buffer. Although this saves a small amount of memory, it is very costly to change the tiling, which happens, e.g., if windows change their positions, since data has to be moved between the screen buffer and their off-screen data structure.

One of the most commonly used graphical systems is the X windowing system (X11) [13] which also supports 3D rendering, e.g., using OpenGL ES 2.0 and EGL. The X Server handles windows and supports compositors by the Xdamage or the Xcomposite extension. *xcompmgr* [14] is a simple compositor for X that uses the Xdamage [15] extension to get X event notifications about window regions that were modified and require an update. Since no backbuffers are used, affected applications need to refresh their content on request which is sent in z-order. By using the *Xcomposite* of the X server the content of all windows is held in off-screen buffers from where the compositor bitblits it to the screen buffer. When a window is updated the compositor is notified and can update the screen. While this might be a good concept for systems where window contents are stored on high-performance memory (e.g., dedicated GDDR5), this is not the case for many embedded systems. For X11 compositing the *Full Compositing* strategy is prevailing.

A successor of X11 is *Wayland* [16] which is less complex than X11 and limits the functionalities to the main tasks which is managing and compositing of windows. Wayland does not separate display server, window manager, and the compositor. An efficient Wayland implementation is *Weston* which uses the *Pixman* library [17] for tiling. Pixman considers how windows overlap and calculates lists of the resulting visible tiles of windows. Hence, the compositor can minimize the

region that has to be bitblitted, avoiding unnecessary redraws of windows which are covered by other windows. By using Pixman, the compositor in Weston is affected by advantages and disadvantages of using the *Tile Compositing* strategy.

In the Android operating system a surface manager is used to manage the surfaces (widgets and view) of applications rather than traditional windows, since windows typically cover the whole screen on Android. Windows are represented by surfaces which are composited by SurfaceFlinger [18]. SurfaceFlinger performs the compositing in a way similar to X11, comparing rectangles at runtime to detect what regions have been updated to redraw the surfaces which have to appear on top of them [18]. Thus, Android uses slightly optimized *Full Compositing*, since at typical use cases windows don't overlap, this seems to be a reasonable, yet not very generic choice.

VII. Summary

The trend towards more graphical functions and applications and the increasing number of displays in cars lead to advanced use cases that allow for dynamic and flexible display usage. Applications are not bound to a display and a static location which enables dynamic configuration of application windows like resizing or moving. In conjunction with z-ordered windows, which allows for overlapping, the graphical system can be as flexible as common desktop systems. Unfortunately, the embedded platforms used in cars are highly restricted in power consumption which limits the performance. Hence, efficiency is a key requirement in automotive graphical window systems.

Typically, automotive applications like speedometer, tachometer, navigation system, and video player draw their content in off-screen buffers which are bitblitted by the graphical window system like X11 in the screen buffers of the displays. In case of overlapping windows the compositing strategy can either bitblit all windows in z-order—*Full Compositing*—which leads to overdrawing of overlapped window areas or bitblit only the visible tiles of the windows—*Tile Compositing*—which requires additional bitblit operations for these visible tiles. To save bitblitting time we proposed the *Hybrid Compositing* algorithm which searches for a combination of bitblitted rectangles to reduce the bitblitting time. To this end, the execution time of a bitblit command must be accurately estimated to choose such a combination. We created a prediction model that estimates the bitblitting time depending on the window sizes and the number of bitblit operations.

In addition, we presented our *Cache-Hybrid Compositing* approach which uses a cache to store the determined sequence of bitblit operations for future use and has a significantly lower CPU execution time. We showed that our *Cache-Hybrid Compositing* strategy is faster than or as fast as the *Full Compositing* and *Tile Compositing* strategies. The compositing time is, in random generated scenarios, with all windows always marked, in average up to $5\ \%$ faster compared to the *Tile Compositing* strategy. When using a randomly selected interval to mark the windows the compositing using the *Cache-Hybrid Compositing* strategy was even $51\ \%$ faster compared to the *Full Compositing* strategy.

To this end, we showed that in our evaluations, our approach always achieved a significant increase in performance compared to existing approaches. Especially for novel automotive HMI systems, the GPU is a vital component. Since the GPU is shared among all applications, using our *Cache-Hybrid Compositing* compellingly reduces GPU utilization, thus providing performance gains for other applications.

Acknowledgment

This paper was supported by the ARAMiS project of the German Federal Ministry for Education and Research with funding ID 01IS11035.

References

[1] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, April 2009.

[2] Mercedes-Benz F125. http://www.motorauthority.com/news/1070046_mercedes-benz-ponders-the-future-ofin-car-tech,2014.

[3] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice (2Nd Ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[4] B. A. Myers, "A taxonomy of window manager user interfaces," *j-IEEE-CGA*, vol. 8, no. 5, pp. 65–84, 1988.

[5] (2015, Mar.) Freescale i.mx 6quad processors. [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX6Q

[6] (2015, Mar.) GNU Octave. [Online]. Available: https://www.gnu.org/software/octave/

[7] N. Meyrowitz and M. Moser, "Bruwin: An adaptable design strategy for window manager/virtual terminal systems," *SIGOPS Oper. Syst. Rev.*, vol. 15, no. 5, pp. 180–189, 1981.

[8] B. A. Myers, "A complete implementation of covered windows for a heterogeneous environment," University of Toronto, Computer Science Dept., Tech. Rep., 1984.

[9] ——, "The user interface for sapphire," vol. 12, pp. p. 12–23, Dec. 1984.

[10] ——, "A complete and efficient implementation of covered windows," *Computer*, vol. 19, no. 9, pp. 57–67, sep 1986.

[11] R. Pike, "Graphics in overlapping bitmap layers," *ACM Trans. Graph.*, vol. 2, no. 2, pp. 135–160, 1983.

[12] L. J. Guibas and J. Stolfi, "A language for bitmap manipulation," *ACM Trans. Graph.*, vol. 1, Jul. 1982.

[13] R. W. Scheifler and J. Gettys, "The x window system," *ACM Trans. Graph.*, vol. 5, no. 2, pp. 79–109, Apr. 1986.

[14] (2015, Mar.) xcompmgr. [Online]. Available: http://freedesktop.org/xapps/release/xcompmgr-1.1.tar.gz

[15] J. Gettys and K. Packard, "The (re) architecture of the x window system," *In Proceedings of the 2004 Linux Symposium*, vol. volume 1, pp. pages 227–237, 2004.

[16] (2015, Mar.) Wayland (display server protocol). [Online]. Available: http://wayland.freedesktop.org/docs/html/

[17] (2015, Mar.) Pixman: The pixel-manipulation library for x and cairo. [Online]. Available: http://cgit.freedesktop.org/pixman/

[18] (2015, Mar.) Surfaceflinger. [Online]. Available: https://android.googlesource.com/platform/frameworks/native/+/android-cts-4.1_r1/services/surfaceflinger/SurfaceFlinger.cpp