Race-To-Sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds

Haibo Zhang[1], Prasanna Venkatesh Rengasamy[1], Shulin Zhao[1],
Nachiappan Chidambaram Nachiappan[1]*, Anand Sivasubramaniam[1], Mahmut T. Kandemir[1],
Ravi Iyer[2], Chita R. Das[1]
The Pennsylvania State University[1], Intel[2]
{haibo,pur128,suz53,anand,kandemir,das}@cse.psu.edu,cnnachiappan@gmail.com,ravishankar.iyer@intel.com

ABSTRACT

Video streaming has become the most common application in handhelds and this trend is expected to grow in future to account for about 75% of all mobile data traffic by 2021. Thus, optimizing the performance and energy consumption of video processing in mobile devices is critical for sustaining the handheld market growth. In this paper, we propose three complementary techniques, *race-to-sleep, content caching* and *display caching*, to minimize the energy consumption of the video processing flows. Unlike the state-of-the-art frame-by-frame processing of a video decoder, the first scheme, *race-to-sleep*, uses two approaches, called *batching of frames* and *frequency boosting* to prolong its sleep state for saving energy, while avoiding any frame drops. The second scheme, *content caching*, exploits the content similarity of smaller video blocks, called macroblocks, to design a novel cache organization for reducing the memory pressure. The third scheme, in turn, takes advantage of content similarity at the display controller to facilitate *display caching* further improving energy efficiency. We integrate these three schemes for developing an end-to-end video processing framework and evaluate our design on a comprehensive mobile system design platform with a variety of video processing workloads. Our evaluations show that the proposed three techniques complement each other in improving performance by avoiding frame drops and reducing the energy consumption of video streaming applications by 21%, on average, compared to the current baseline design.

CCS CONCEPTS

• **Human-centered computing** –> **Mobile computing**; • **Computer systems organization** –> **Architectures**; **Embedded hardware**;

KEYWORDS

Mobile SoC, SoC, Memory, Video streaming, Display, Caching

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123948

ACM Reference format:
Haibo Zhang[1], Prasanna Venkatesh Rengasamy[1], Shulin Zhao[1], Nachiappan Chidambaram Nachiappan[1]*, Anand Sivasubramaniam[1], Mahmut T. Kandemir[1], Ravi Iyer[2], Chita R. Das[1]. 2017. Race-To-Sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 15 pages.
https://doi.org/10.1145/3123939.3123948

1 INTRODUCTION

Video streaming has become the single most important user experience in mobile devices, with this trend only likely to be even more prevalent in the future. It has been reported that more than half of YouTube views, constituting over a billion each day, come from mobile devices currently [33]. Cisco predicts that by 2021, more than 75% of mobile data traffic will be video [22]. Apart from fetching video streams online from the internet over wireless, offline viewing of pre-downloaded video is also very popular to deal with network connectivity issues. Further, video display is a critical component of many other widely used applications, including high-end gaming (apart from Snapchat, Skype, etc.) where applications have started employing high-end 4K or even 8K video formats and VR displays in order to provide users with an immersive user experience. It is thus no surprise that video performance, and its consequent energy consumption, is essential to the continued successful growth and acceptance of mobile devices, especially with their limited battery capacities that do still run out within a day of normal usage.

Race-To-Sleep: To better understand the processing and energy consumption of this important paradigm (video streaming), we first conduct an in-depth hardware and software analysis of different applications on today's system (see Fig.1a). The hardware video decoder, display and memory system contribute to nearly 75% of the total energy during streaming and 85% of the execution time, making these important targets for optimization. Further, as we will show, there is a gross inefficiency in how video processing is currently done, that leads to high energy inefficiencies. Specifically, even though current hardware decoders, which process encoded video frames and render them for subsequent display, expose different low-power modes, the spacing between processing of successive frames is not large enough to meaningfully transition to a low enough power mode. In fact, the transition times and transition energy outweigh any possible savings offered by these low power modes. To date, no prior work has identified this deficiency, or offered ways of providing energy savings, while accommodating the high transition costs.

*Work was done as a student at The Pennsylvania State University.

Our solution to this problem, employs two complementary ideas: *batching* and *race-to-sleep*. It is based on the observation that, while the time separation between any two successive frames is small for the intended transitions, summing up this time across multiple frames would create a large enough time window that enables meaningful power mode transitions. To achieve this, we need to accumulate several successive frames, and batch them together for decoding. Current video processing, even if it buffers several frames from the network or storage device (to accommodate bandwidth vagaries), still sends these frames one-by-one to the decoder at periodic intervals. Instead, we propose to batch-send several frames to the decoder, leaving no time gap (batching). Further, we also propose to speed up the decoder to not only allow us to better utilize the memory Activation/Precharges more efficiently, but also to create more slack/idle time for transitioning to deep sleep states, despite incurring higher transition costs and requiring larger memory buffers.

Content-based Caching: A consequence of the above solution is the need to store multiple frames readily accessible to the decoder, which shoots up the memory storage and bandwidth requirements. Since memory is a precious resource on handhelds, we need to alleviate these increased requirements. Current decoders do use a cache, to reduce memory demands. However, this cache is woefully inadequate when dealing with several frames concurrently, as mandated by our batching solution. We will show that just boosting this cache's capacity is insufficient. Instead, we propose a caching solution that exploits content similarity typically found within and across frames. Such similarity, though widely exploited in encoding algorithms (e.g. VP9/H.264/H.265 [55, 80, 89]), has not been previously exploited at the post decoding stages, especially towards building a different caching structure, as in our work.

We empirically show that more than half of the current frame's content have been recently accessed by the decoder, despite the addresses being different. Consequently we propose a MAcroblock caCHe (MACH) of only 8KB, that stores the content of recurrent macroblocks (mab) – a rectangular segment of a frame's pixels (3840x2160 pixels) that is also the commonality exploitation technique of many encoding schemes (e.g., VP9/H.264/H.265 [55, 80, 89]). Additionally, we introduce the notion of a "Gradient", that helps us discount slight variances that may occur across mabs, exploiting the fact that the differences between the successive pixels within a mab would match across mabs, even if those mab contents are themselves different. With these enhancements, the decoder only needs to write the unique content and the pointers to this content, thus saving valuable memory bandwidth as well as energy.

Display Caching: The display expects the decoded data to be in memory, and the resulting accesses are expected to sequentially span through the arrays of pixels, exhibiting high spatial locality. However, with the above content-based mechanism, the memory can no longer be sequentially accessed by the display, which has to now look up pointers and fetch the appropriate content for each mab. To address this issue, we again leverage the locality of the displayed content, by maintaining a very simple direct-mapped cache, called Display Cache. Since content is similar within and across several frames, by caching recent content at the display, we can easily find the rendered data, and also avoid going to memory to find much of the rendered content.

Contributions: This is the first study to explicitly focus on the video processing hardware and software pipeline of mobile devices to identify the energy and memory bandwidth inefficiencies, and propose a novel three-step strategy to address these concerns. Using a diverse set of videos, and a detailed hardware simulation and complete system software emulation stack, this paper shows that:

- The hardware video pipeline and memory system constitute around 49.9% and 37.5% of the time in video processing, and 29.7% and 45.8% of the energy. More than half of the frames in the videos do not have sufficient time gap for effective power mode transitions. In fact, 4% of them are not even rendered before their deadline, leading to "frame drops".
- With our proposed `Race-to-Sleep` mechanism, the decoder is in deep sleep state for around 60% of the time, compared to only 5% of the time in baseline. Further, we eliminate all frame drops.
- With content-caching, we reduce the overheads of memory capacity and memory accesses by 34%, compared to the baseline.
- The display cache contributes to an additional 33.5% reduction in display's memory bandwidth, compared to the baseline.

All these enhancements put together provide a video streaming platform that is 21% lower in energy compared to today's baseline system while eliminating frame drops altogether.

2 BACKGROUND AND MOTIVATION

In this section, we illustrate how a video is processed in a handheld using the Android framework, describe how the frames get rendered on screen, and finally present the inefficiencies in the system through a detailed analysis of timing and energy breakdowns.

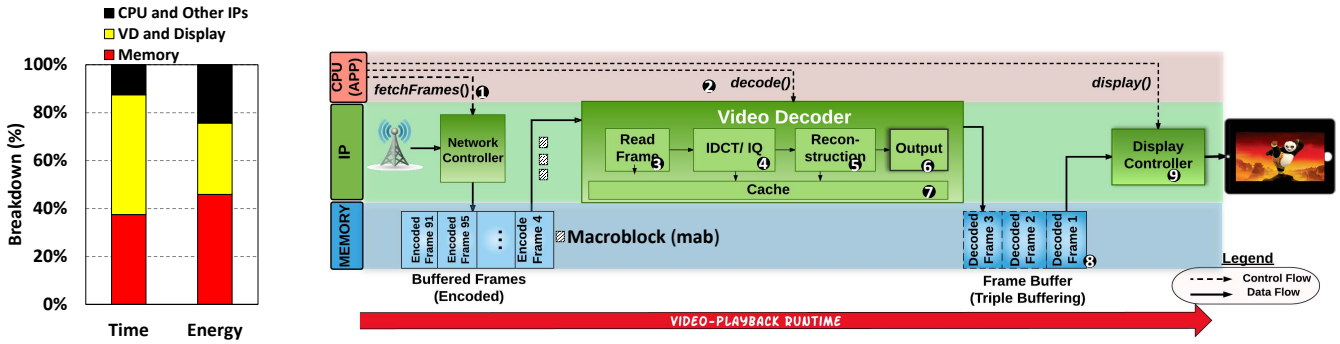
2.1 Video Processing in Mobile Systems

To explain the video processing steps in a mobile system, let us consider an example of a 4K video playback in YouTube on a Nexus 7. During playback, we use the *ftrace* tracing tool built in Android's kernel to capture all the IP calls. With this, we map out the data and control flows and show them pictorially in Fig. 1b. Here, each frame undergoes three stages of processing, namely, *Buffering*, *Decoding*, and *Displaying*.

Buffering: Once a user clicks to play a video, YouTube will buffer several seconds worth of frames from the media streaming server to the device memory. Buffering frames [6] is commonly employed in network streaming applications as it helps to deal with network bandwidth vagaries. It also enables the decoder to process the frames ahead of time, thereby increasing the time available before which a frame is needed for rendering on a screen. Usually, the frames delivered from the media-server are compressed using one of the encoding mechanisms such as H.264 [89], H.265 [80], or VP9 [55]. These encoded frames[1] are buffered in memory, with each taking hundreds of KBytes of memory space. In order to meet the frame delivery QoS, buffering occurs periodically at an interval of around 400-500ms [6], shown as ❶ in Fig. 1b.

Decoding: Once a bunch of frames are available in the memory, YouTube calls the hardware video decoder (VD) to decode the

[1]The encoded video normally contains various types of frames, namely, *I, P and B frames*. To reuse data, *B/P frames* consist of all types (I/P/B) mabs and have references to the previous/next *I/P frames*, whereas *I frames* consist of *I mabs* and are self-contained.



(a) Time/energy breakdown

(b) Control flow and data flow of YouTube.

Figure 1: (a) Performance and energy breakdown of YouTube; (b) Control flow and data flow of YouTube.

frames one by one. For a video playing at 60fps, the application calls VD every 16ms (1/60s), shown as ❷. Upon invocation, the hardware decoder reads a buffered frame from the memory, decodes it, and stores the decoded frame in another memory location, commonly called as *frame buffer* [88]. A frame buffer will store a decoded frame, and waits for the display to read the frame. Note that, the decoded frames are usually large, e.g., a decoded frame from a 4K video can consume up to $8M pixels * 3 BytesPerPixel = 24M Bytes$ memory space. In modern mobile systems, the number of frame buffers can be 2 (*Double Buffering* [30]) or even 3 (*Triple Buffering* [29]), allowing one to be written into, while another is being displayed concurrently.

Displaying: Modern mobile platforms are usually equipped with a 60Hz display [31]. This provides the display with the potential to present a new frame every 16ms (as explained earlier). To do so, the display checks for a new frame in the frame buffer, and if present, reads it in the next 16ms; if not present, the display detects a *frame-drop* and renders the previous frame again.

2.2 Insomnia: Inefficiencies of Current Video Processing

We next discuss the inefficiencies in managing this flow. The time a hardware decoder takes to decode a frame is not always constant as explained below. The decoder reads an encoded frame and starts decoding it. An encoded frame consists of a number of smaller regions of the frame, each called a *macroblock*[2] (mab) [89]. The encoded mab is the basic processing element (unit) in video decoding, containing the information for a region of the frame (typically includes 16x16 [89], 32x32 or 64x64 pixels [55, 80]). The VD reads the encoded frame at a mab granularity, shown as ❸ in Fig. 1b. The encoded mabs exist in a quantized, DCT-transformed, and compressed format. As a result, they need to pass through a series of stages including entropy-decoding, inverse-DCT and inverse-quantization in step ❹. Next, the mabs are reconstructed in various ways, depending on their types. An I-Type mab is reconstructed from its neighboring mabs of the same frame, whereas a P-Type or B-Type mab is reconstructed from the mabs in the previous/latter frames that are indicated by the extra information stored as *motion vectors*. Note that, the number of I/P/B mabs varies from frame to frame, and this is the main reason why the decoding time varies depending on the frame. In step ❺, the VD reconstructs the frame from mabs. Therefore, if the required work is low, the decoder can finish processing before the 16ms deadline, and can save energy by transitioning into the *sleep* or even *deep-sleep* state.

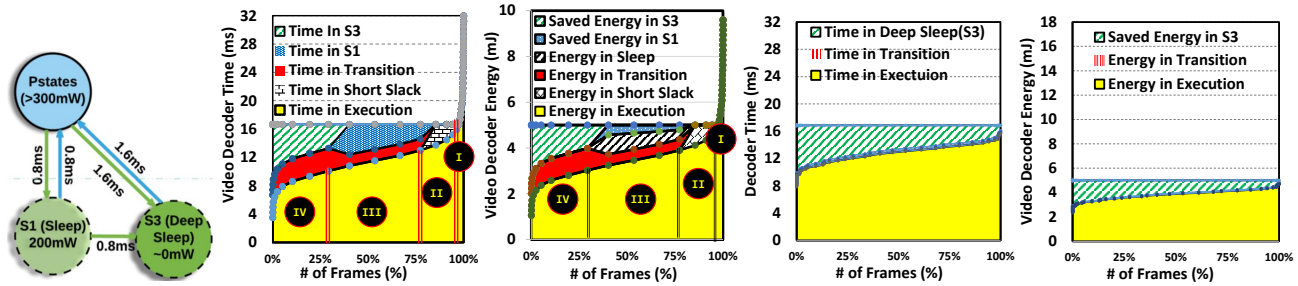
Fig. 2a shows the power states in a current mobile SoC [95], including running states (P-state), and two sleep states (S1, S3). The active P-states consume more power than the two sleep states because different computation units are busy during this time. The device cannot execute tasks when in sleep states. The transition from one power state to another is called *Power Transition*. Transitioning from the S1 state to P-state requires 0.8ms, whereas transitioning from the deep-sleep state to P-state takes around 1.6ms [95]. Hence, *it is not always beneficial for the VD to go to lower power states in between frames, even if it may have the time to do so.*

Time and Energy Breakdown of Video Processing Flows: We evaluate the video processing flow and plot the breakdown of frame time and energy in Fig. 2b and Fig. 2c. Fig. 2b shows the distribution of decoding time for ≈5000 frames, while the corresponding energy breakdown is shown in Fig. 2c. Based on the frame's completion time, we divide the frames into four regions:

- ***Region I:*** *Frames that are dropped.* We see that around 4% of the frames are dropped since their decoding times are longer than 16ms. The decoder has no chance to go to a sleep state while processing these frames; instead, it just continues to decode the next frame to prevent subsequent frame-drops.
- ***Region II:*** *Frames taking close to 16ms.* This region contains around 12% of the total frames. These frames get processed within the deadline, but VD does not get adequate time to transition into any low power state due to short slacks.
- ***Region III:*** *Frames that can go into the sleep state (S1).* 37% of frames finish earlier than the deadline, and thus, have enough slack to allow VD to comfortably transition into the S1 state. However, by doing so, the decoder incurs extra 3.6% and 4.7% of transition energy and latency, respectively. Note that, the slack available in this case is not sufficient to go to the deep-sleep (S3) state.
- ***Region IV:*** *Frames that can go into the deep-sleep state (S3).* The remaining 40% of the frames finish even earlier than region III, and have enough slack to allow the decoder to transition into the S3 state. Again, the transition incurs an extra cost of 10.2% and 7.9% in transition energy and latency costs, respectively.

Note that, before moving to the S1 or S3 state, if the decoder finds that it does not have enough sleep time to offset for the transition

[2] Similar concept is also defined as *coding tree block* (CTB) in [80] or *super-blocks* (SB) in [55]. In this paper, we use the term *macroblock (mab)* to describe a block of pixels.



(a) Power states and transitions (b) CDF of frame execution time in increasing order (c) CDF of frame energy consumption in increasing order (d) CDF of batching timing (e) CDF of batching energy

Figure 2: (a) State diagram of power state transitions. (b) and (c), CDF plots of frame execution time/energy consumption for 5000 frames sorted in increasing order. Per frame execution time is fixed at 16.6ms and per frame energy is fixed at 5mJ (decoder power 300mW * frame time). The plots show the time/energy spent in five different states (S3, S1, transition, short slack, and execution). (d) and (e) depict time/energy CDFs with *batching*.

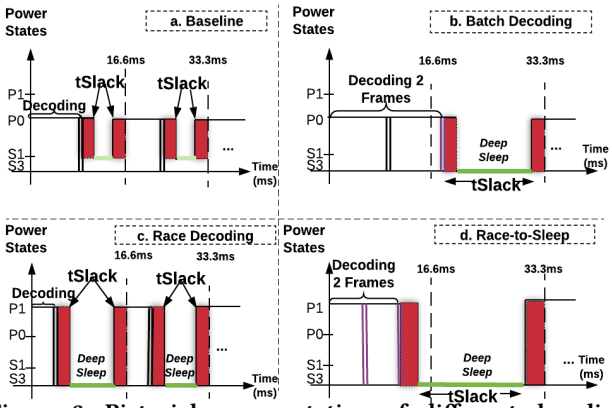


Figure 3: Pictorial representation of different decoding schemes. tSlack is the time taken to process the next frame. P1 and P0 are Pstates. The transition energy between P and S states is shown in red.

energy, then it would not transition to that sleep state. One can observe two inefficiencies in the existing video decoding flow:

- *More than half (60%) of the frames cannot transition into deep sleep.* Specifically, frames in Regions I, II, and III do not allow the decoder to transition into *deep sleep* because of two reasons: a) not enough slack to allow the transition time, and b) the energy savings in *deep sleep* cannot offset the *deep sleep* transition energy.
- *Power state transitions reduce sleep time.* Frames in Regions III and IV spend 13.8% more decoding time in *power state transitions*, as well as 12.6% energy even with active power management.

3 RACE-TO-SLEEP

In this section, we tackle both the inefficiencies mentioned above by (i) employing a batched decoding of frames for reducing the number of transitions and thus, increasing the slack time, and (ii) speeding up the decoding with a higher frequency to gain more slack per frame. We discuss both these techniques and combine them to propose *Race-to-Sleep*.

3.1 Batch Decoding

In the current video processing setup, shown in Fig. 3(a), the video decoder finishes the decoding of a frame and goes to a sleep mode if there is enough slack. From each frame's perspective, it may be a locally optimal strategy, but given that there are a number of frames buffered, this may not be the global optima.

Motivated by this, we propose a *Batch Decoding (Batching)* setup shown in Fig. 3(b). As explained earlier in Sec. 2, the input frames (in encoded form) are already streamed and buffered in the memory. Batch decoding leverages these buffered frames by sequentially decoding one frame after another, and transitioning to sleep only after finishing all the frames in a batch. To show the effect of this scheme, we batch 16 frames and plot the time and energy CDF curves in Fig. 2d and Fig. 2e. When compared to the four regions discussed earlier (Fig. 2b and Fig. 2c), this scheme effectively brings three benefits:

It eliminates frame drops. Due to batching, the frames that are dropped in the baseline scheme (Region I) can start earlier, and finish within their deadline by borrowing slacks from the frames in Regions II, III, and IV.

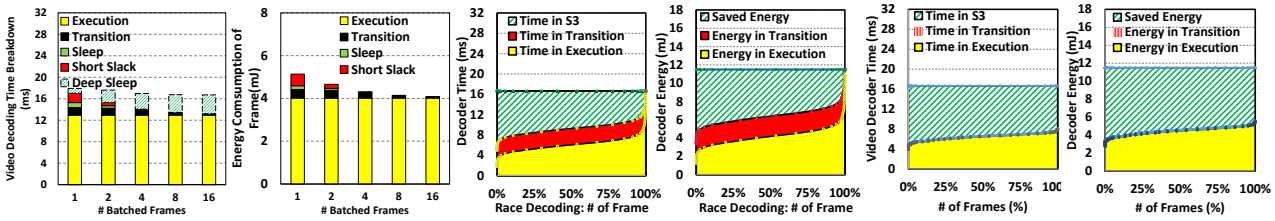
It creates better opportunities for the video decoder to go into the deep-sleep mode. Batching helps in accumulating the slack time of individual frames to put the VD in a longer deep sleep state at the end of each batch, as shown in Fig. 2d.

It reduces the number of transitions. The transition frequency is reduced by a factor of *n*, where n is the number of frames in a batch. In Fig. 2d timing breakdown, we can see that the transition overheads per frame are negligible. In fact, they are reduced by 16x to 0.2ms, which translates to around only 1.2% of the frame time.

Figs. 4a and 4b show the time and energy breakdown of the execution of a batch of *n* frames. It can be observed that, by batching and decoding of 16 frames together, we are able to reduce the transition energy by 86%, and the total video decoder energy by 20%.

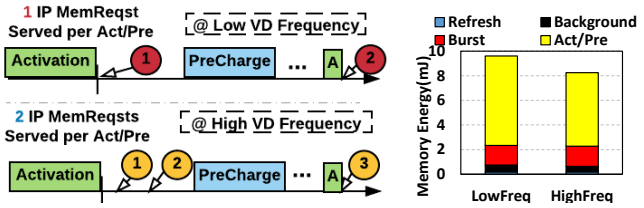
3.2 Race Decoding

Why is batching still not the global optima? While batching is one approach to solve the inefficiencies of current video processing, another alternative is to increase the video decoder's frequency. Modern mobile systems provide options to scale to higher frequency modes to speed up processing [10, 99]. Although higher frequency will increase the decoder's power, we can still achieve energy savings by reducing the frame processing time because a decoder operating at a high frequency (300MHz) can utilize the



(a) Batching Time (b) Batching Energy (c) Race Decoding (d) Race-to-Sleep

Figure 4: Effects of various schemes. (a) and (b) show the effect of *Batch Decoding*. (c) and (d) plot the time/energy CDFs with *Race Decoding* and *Race-to-Sleep* for processing 5000 frames (as similar to Fig. 2b and Fig. 2c). *Race Decoding* increases transition a lot while *Race-to-Sleep* reduces transition mostly.



(a) Memory Timing (b) Energy Breakdown

Figure 5: (a) Memory access pattern with low and high frequency settings. DRAM needs more `Act/Pre` for a low frequency setting. (b) showing energy saving of using a high-frequency VD.

DRAM (1.6GT/s) more efficiently. Intuitively, bridging the gap between the video decoder (VD) and DRAM operating frequencies seems to be a better idea than just *batching*. Note that, we do not want to reduce the memory frequency to impact CPU performance (or the behavior of other latency critical accesses to memory) by increasing the memory response time [40, 87]. Hence, we propose *Race Decoding (Racing)*, which transitions VD to a high-frequency mode. We later discuss its timing and energy benefits, as shown in Fig. 3(c).

Where will we gain from race decoding? Let us consider the memory access patterns of VDs with two different frequencies shown in Fig. 5a, where a memory access will *activate* (`Act`) a memory row (*row-buffer*) and read/write data into the row in *burst mode*. The row-buffer can hold a row up to a limited time-duration to avoid starving requests to other rows [54, 74]. After that duration, the memory controller needs to precharge the row and swap the current row out to serve other requests. Note that each extra *precharge* (`Pre`) consumes additional energy. Therefore, if two consecutive memory accesses are destined to the same row (typical "streaming behavior" for VD), it will be better if they are served in one `Act/Pre`-duration, since doing so will also reduce the number of `Act/Pre` for the same row. In our video decoding scenario, operating at a higher frequency consumes an additional 0.5 mJ per frame at VD, but saves around 1 mJ on the memory side. We show the memory energy breakdown of two VDs' frequency settings in Fig. 5b. One can see that the VD using high frequency can save more `Act/Pre` energy than the one using low frequency. Fig.4c shows the impact of *racing*, leading to the following observations:

The frame drops are eliminated. Compared to *batching*, *racing* reduces frame-drop by increasing the processing rate.

The time spent in S3 is increased. This is mainly because we now have more slack for sleep due to faster decoding.

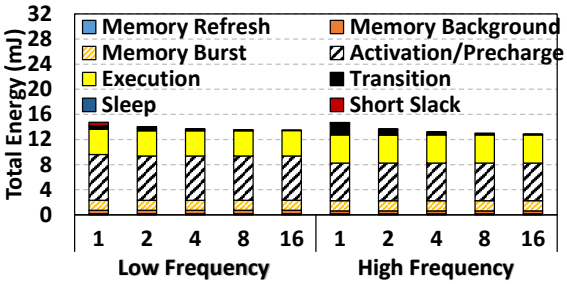


Figure 6: Impact of Race-to-Sleep. Left: VD at low frequency; Right: VD at high frequency (Batching up to 16 frames.)

The decoding energy decreases. As we reduce the `Act/Pre` energy on the memory side, it more than compensates for the energy increase in the VD.

The energy in transitions increases. This is because the operating frequency is increased, as shown in the *red* area in the timing breakdown given in Fig. 4c. Note however that this can be reduced by batch decoding.

3.3 Race-to-Sleep

Since *batching* and *racing* are orthogonal techniques, we can combine them as *Race-to-Sleep*, illustrated in Fig. 3(d). As can be seen in Fig. 4d, the VD gets to go to deep-sleep for the longest duration in this method. In Fig. 6, we evaluate two frequency settings with batching 1 to 16 frames. We save most energy by tuning the VD to high frequency and batching 16 frames over the no *batching* and no *racing* setting. Based on the results, we summarize the following key take aways:

The energy consumption in video decoding is reduced. *Race-to-Sleep* reduces 12.9% of the frame energy; 6.7% of these savings come from *batching* and 6.2% come from *racing*.

The memory energy is still high. Although we use *racing* to reduce memory energy by 14%, we still see around 64.2% of the energy consumption coming from memory, where `Act/Pre` energy contributes to 46% and the burst energy contributes to another 12.7%.

The memory capacity requirements increase. By *batching*, we increase the runtime memory requirements by 5.3x (from triple buffering up to 16 buffers for 16 frames), which costs around 384MB for a 4K video.

The memory bandwidth requirements increase. *Race-to-Sleep* also increases the memory bandwidth (accessing the same volume of data in a shorter period of time).

Note that, *Race-to-Sleep* does *not* need to wait for 8 frames to start. In fact, it is adaptive to network performance and can leverage any number of frames that are already buffered. Fig. 6 shows that, even if there are only 2 frames in the buffer, we can still save 7% energy. With higher number of buffered frames, which can often happen because of burstiness of network transfers, we can save up to 12.9% energy.

In the next two sections, we propose two caching techniques (content caching for the VD and display caching for the DC) to reduce memory capacity and accesses both from the VD and DC.

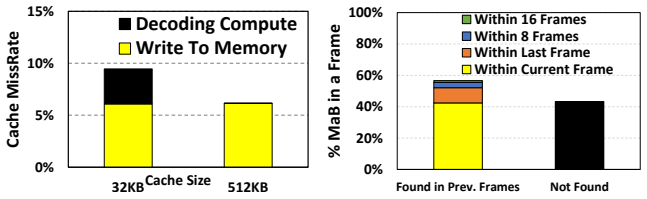
4 CONTENT CACHING

Let us refer to Fig.1 for understanding the steps executed in displaying a frame after it has been decoded. After decoding a mab ❺, the decoded mab is written back ❻ to the memory (frame buffers ❽) for the display ❾ to read it. Current VDs [38, 43, 99, 100] are usually equipped with a "cache", shown as ❼, to take advantage of locality. As we discussed in Sec. 3, the memory energy is high and *Race-to-Sleep* increases both the memory space requirement (for storing frame buffers) and the memory bandwidth requirements. In this section, we try to leverage the available cache in the VD to minimize the number of memory accesses and thus, save memory energy. More specifically, we identify the inefficiencies of the current cache designs and then propose our alternative structure to reduce the memory accesses. In the rest of the paper, a "macroblock" (mab) refers to a *decoded* block of pixels (and not the preliminary compressed version). We assume the pixels in the frames to be in RGB color space (as used in Android Frame-buffer [32]). Note however that our technique is generic and can be applied to all the other color spaces as well (e.g., YUV, YCbCr, etc,). Also, our design does not rely on or change the hardware codecs. These hardware codecs work with compressed, encoded video frames for transmission/storage. However, they are still in need of subsequent decoding, and processing for display them. Our proposal addresses the phase of the video processing (i.e. after the frames have been decoded and are being processed).

4.1 Address Locality vs. Value Locality

Could a larger cache help? Although video processing is mainly stream-oriented, the computations involved in decoding a frame (such as entropy-decoding, iDCT, etc.) can leverage address locality/reuse [39]. However, the output of the decoder is a stream of mabs, which has no address locality/reuse. We investigate this further by increasing the cache size in a VD from 32KB to 512KB, and plot cache miss rate in Fig. 7a. We observe that a larger cache helps in decoding by reducing the miss rate by caching almost all of the compute-intensive memory accesses (e.g., motion-compensation). However, for writing back the streams of frame mabs, employing a large cache is not beneficial, implying that we are not going to reduce the DRAM write bandwidth requirements by simply adjusting the cache parameters.

Similarity between macroblocks: To understand what else can be done in terms of memory bandwidth reduction, we exploit the compressed format frame representation and study the similarity of data between the macro blocks (within or across frames). In particular, for each mab, we want to see whether its content has an exact match with those of the previous 16 frames. We plot the number of mabs that can be found in any of these previous frames in



(a) Conventional Cache. (b) Content matching of mabs.

Figure 7: Access locality vs. Value locality. (a) Conventional cache can hardly exploit address locality in writing a frame back to memory. (b) More than half of the current frame are already stored in memory.

Fig. 7b. From these results, we make the following observations:

42% Intra-Match. Around 42% of the mabs can be found within the same frame, called as Intra-Match.

15% Inter-Match. We found that another 15% of the mabs can be found in the previous 16 frames. Also, the possibility of finding a match of a mab beyond the past 16 frames is very low (less than 1%). If a mab can be found in one of the previous frames, we name it as Inter-Match.

43% No Match. Finally, we also see that around 43% of mabs could not be found in the same or previous 16 frames.

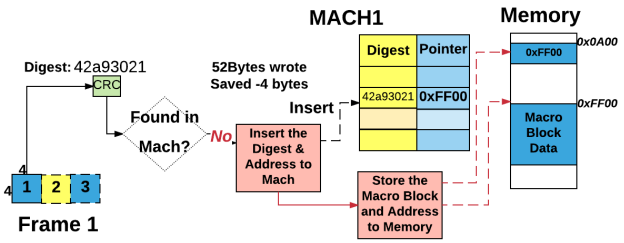
Overall, 57% of the total mabs can be found in either the current or the previous 16 frames. This raises the question "Can we reduce the number of mabs written into the memory by reusing those which are already in the memory?" In other words, can we take advantage of "content caching" across mabs? To this end, we propose MAcroblock caCHe (MACH).

4.2 MACH: Macroblock Cache

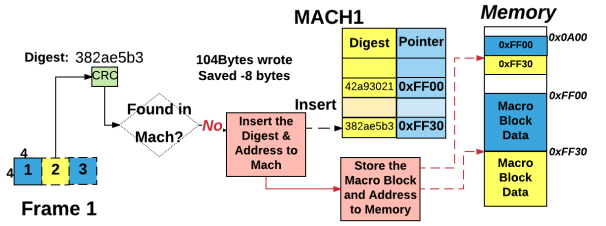
MACH is a set of entries that records where a mab is stored in memory. Each entry contains a *tag* and a *value*. The *tag* represents a mab, and the *value* is the address of the mab stored in the memory (frame buffer). Fig. 8 illustrates the working of MACH with examples.

Assume that we start decoding the first frame and the MACH is initially empty, as shown in Fig. 8a. After the first mab is decoded, we compute the *digest* of the mab. The digest is the representation of the mab after being hashed by CRC32 [65]. We use digest instead of the real data as the tag since, compared to the $4{*}4pixels{*}3Bytes/pixel = 48B$ mab size, a digest is just 32 bits (4B), which leads to much lower overheads (we found collision probability to be extremely low even when a very simple CRC mechanism is used to compute this digest). Once we have the digest, we check if it is already present in MACH. Since this is the first mab in our example, it is not found in MACH. Consequently, we store the mab and the address of the mab in the memory (pointer), and insert the digest and its address into MACH. Note that, at this point, since we store the mab and its pointer, we do not save any memory access; in fact, we store 4 more bytes (52-48 = 4 bytes) compared to the conventional strategy.

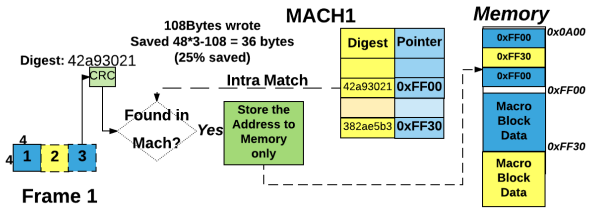
Assume now that the second mab is different from the first mab, shown in Fig. 8b. Since its digest is different from the first one, we cannot find it in MACH. As a result, we need to store the whole mab and its pointer in the memory, and insert a record for it into MACH, requiring again 4 extra bytes. We now assume that the third mab has the same 16 pixels as the first mab, as shown in Fig.



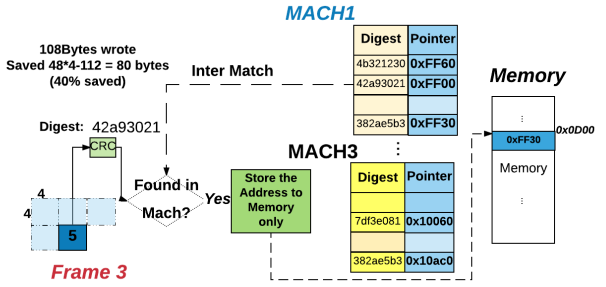
(a) MACH is initially empty. Originally, 48 bytes are written into memory. Now, we write 52 bytes (pointer and the macroblock), resulting in 4 bytes overhead.



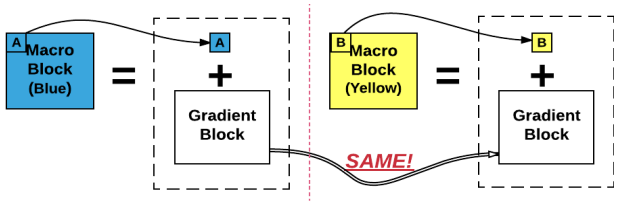
(b) Reference missing in MACH.



(c) Reference hit in MACH reduces 25% of the memory access/space requirements (Intra Match).



(d) Reference hit in MACH reduces 40% of the memory access/space requirements (Inter Match).



(e) Two mabs have the same Gradient Blocks but different bases.

Figure 8: Examples illustrating how MACH works.

8c. In this case, we can find a match in MACH. Note that, at this point, we know that there is an *identical* mab already stored in the memory, and we also know where it is being stored from the pointer. Consequently, instead of storing the whole mab (48 bytes), we can just store the pointer, which saves 44 bytes, resulting in a 36 bytes saving compared to the original 144 bytes (25%) of the decoded data.

In our implementation, we have one MACH per frame. Therefore, once all the mabs in a frame are written back to memory, MACH essentially contains a set of unique mab digests and their pointers for a particular frame. Note that, this MACH is now fixed and will not be changed any more. We can keep this MACH (marked as MACH1) in the video decoder for future references. In Fig. 8d, a mab from frame 3 finishes its decoding and its contents are same as the first mab in frame 1. Hence, we can find a match in MACH1, and also obtain the pointer of the mab which contains the same content as the current mab. Now, instead of storing the whole mab, we can store only the pointer, which points to where the data is, and this again saves another 44 bytes, resulting in a total saving of 80 bytes over 192 bytes when considering all 3 frames in the example (i.e., 42% savings). To further optimize the storage of mabs in MACH, we explore the possibility of representing mabs as gradient blocks.

4.3 Gradient Blocks

Consider two mabs A and B in Fig. 8e where A has 16 pixels of blue and B has 16 pixels of yellow. If we compare these two mabs, we find that their data are different. However, for each mab, if we substract the first pixel from each of the pixels in the mab, we can also see that *the gradient of these two blocks are the same*. We define *gradient block* as the color difference of the macroblock which uses the first pixel as the base. Therefore, if we store in MACH, the "gradient block" (gab) instead of mab representation, we can find more matches and save even more memory accesses, and better utilize the MACH entries. Now, if we go back to the example (Fig. 8) and use gab instead of mab, we do not need to store the second gab in Fig.8b. Instead, we find a match in MACH, since the gabs corresponding to mab1 and mab2 are the same. The overhead we pay for each matched mab is 4 bytes of *pointer*, to point to the gab, and the *base* of the mab, which is the first pixel (3 bytes). This gab-based strategy thus saves $48 * 4 - (48 + (4 + 3) * 4) = 116$ bytes over 192 bytes (60% savings), which corresponds to 18% additional savings over using just a mab-based caching.

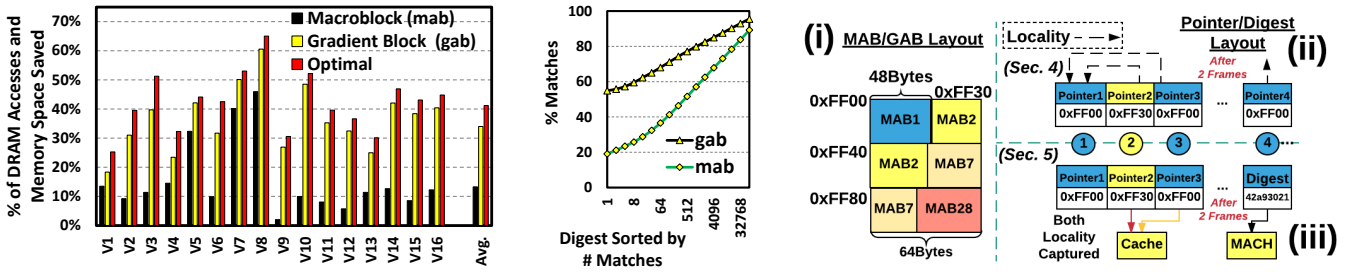
4.4 Implementation of MACH

There are various design issues that need to be addressed when implementing our proposed MACH in the VD:

Generating the base and gradient block: The required subtractor for generating gab can be implemented as a vector unit [11, 28]. To compute the gab, we use the first pixel (top-leftmost) in mab as the "base". Each subtractor unit takes one color channel of a pixel in 4x4 mab, and subtracts each of the base correspondingly to generate the gab.

Generating the digest: We use CRC32 to generate the digest of mab/gab in this work. As will be later shown in Sec. 6, compared to other hashing schemes, CRC32 is quite efficient and has a very low number of collisions.

Coalescing: The base, mab/gab and pointers are not aligned with



(a) MACH reduces memory accesses and storage. (b) Top Digests. (c) Framebuffer layout after using MACH.

Figure 9: Effect of content caching. (a) mab saves 13%, and gab saves 34% memory bandwidth and memory space. (b) gab always captures more matches than mab. (c) i. Memory layout of mab/gab; ii. layout of the pointers in Sec.4; and iii. layout of the pointers/digests in Sec.5.

a cache line size (64 bytes in our case). A base is only 3 bytes, a pointer is 4 bytes, and a mab/gab is 48 bytes. Clearly, we do not want to issue a 64 byte memory request, for only sending one base or one pointer back into memory. Instead, we *coalesce* these outputs into three different buffers, each of which is 64 bytes. Only when a buffer is full, it is written back to memory to reduce the number of memory requests.

Design parameters of MACH: There are several design options in implementing MACH. To capture more matches in the current frame as well as in the previous frames, we choose to implement 8 MACH caches in the VD. Since the number of mabs in a 4K frame is around 500k, if we store the digests (4 bytes each) of all the mabs in MACH, it will be around 2MBytes for just one MACH. Also, it is very costly to build a fully-associative MACH. Instead, we opt to employ a 256-entry, 4-way associative LRU MACH. As a result, we need to pick 6 bits from the 32 bit digest for *indexing*. Our study shows that all 32 bits of the digest are equally distributed (we do not present the detailed results due to lack of space). So, we pick the lower 6 bits to index the set. We present a sensitivity study of our design choices in Sec. 6.

Reconstruction of the frame on the display side: Since we save each mab/gab's pointer, the display controller (DC) first needs to read the pointer, and then fetch the actual block using the pointer. In the mab-based implementation of MACH, the DC needs to read the pointer, which increases a mab access by 4 bytes. In our gab-based MACH implementation, the DC also needs to read the base of each gab, and add the *base* back to each pixel to restore the mab. This increases the DC's memory accesses. The next section will discuss mechanisms for addressing this inefficiency.

4.5 Discussion

We pick 16 popular videos, shown in Table 1, to evaluate MACH's performance. We quantify the impact of both mab-based and gab-based implementations in Fig. 9a. The MACH settings we use is 8 MACH with 256 entries and 4-way each. By using 8 MACHs, a frame can refer up to eight previous frames if needed. We can see that, by using mab, we save around 13% of memory accesses as well as memory space. Using gab on the other hand, we save more than 34% of memory accesses as well as memory space. Note that, in the mab-based implementation, the metadata occupies 4/48 = 8.33%, whereas, in the gab-based scheme, the metadata (including pointers and bases) occupies 7/48 = 14.58%. We also show the optimal case in Fig. 9a. The optimal case is when we can pick the "most useful" mab to reside in MACH. Compared to the optimal, we are 7% worse, because we use LRU due to its simplicity. We leave how to intelligently pick what digest resides in MACH to a future study.

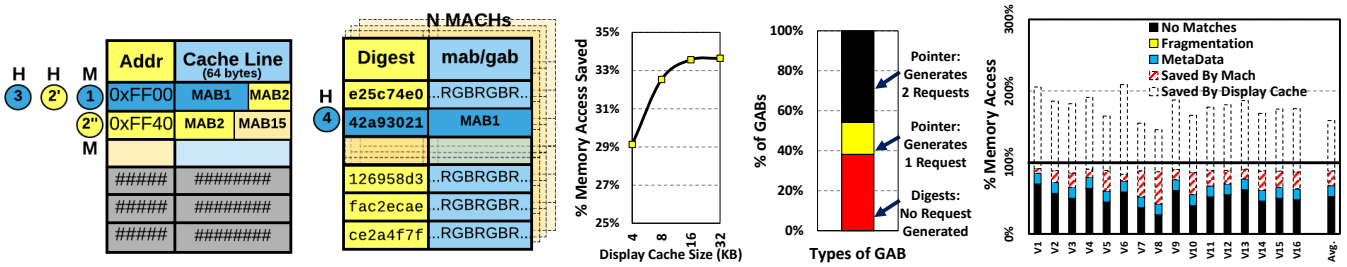
To understand why gab is better than mab, we show in Fig. 9b the matches of each digest with respect to the overall matches. We can see that, when using mabs, the most matching digest contributes to around 20% of the total matches; however, with gabs, the most matching digest contributes to 58% of the total matches. In case of mab, only a ***specific*** pure color mab (a mab with all same color pixels) can match the topmost matching mab, but, in gab, ***any*** pure color mabs can match the topmost matching mab. We can also see that, given a fixed number of entries in MACH, the maximum number of matches with gab is always better than that with mab.

Although MACH reduces the memory accesses considerably, it requires the memory layout of a frame in the frame buffer to be changed, as depicted in Figs. 9c(i) and 9c(ii). The mabs are originally stored one by one sequentially; however, since now we compact the frame, some of the mabs are not stored as shown in Fig. 9c(i). Instead, to find all the mabs, we first need to read the pointer, then figure out from the pointer where to find the mab as shown in Fig. 9c(ii). However, since we need to fetch the pointers first and then the mab, the overhead of restoring the frame using this layout can be quite high. In the next section, we address this issue by reducing the memory accesses from the DC side.

5 DISPLAY CACHING

We first summarize the problems of making the display controller (DC) read from the MAB/GAB layout (as seen in Figs. 9c(i) and (ii)) resulting from applying MACH at the VD (for the example scenario in Fig. 8). Note that, we introduced an additional level of indirection in this layout (the pointer layouts) which is absent in the baseline setup. This indirection causes two additional problems, namely, (a) request fragmentation because of the unaligned pointer accesses (to 48 byte size mabs), which causes (b) locality loss. We use the example in Figs. 9c(i) and (ii) to illustrate these problems.

When the DC is initiated, it starts reading the pointer data sequentially in Fig. 9c(ii) to look up for the corresponding mab. This is shown in the numbers (1 to 4). By doing so, it starts reading the block at 0xFF00 (MAB1), then 0xFF30 (MAB2), and so on. The first pointer points at MAB1. MAB1 occupies 48 bytes, and the rest of the cache line (16 bytes) is taken by MAB2. So, when the DC issues a memory request to fetch the first mab, it essentially fetches a cache line which contains the whole MAB1 and a part of MAB2.



(a) (b) (c) (d) (e)

Figure 10: Architecture and effects of display cache and MACH buffers. (a) Illustration of display cache; (b) Illustration of MACH buffers; (c) Sensitivity to size of display cache; and (d) Distribution of types of gab; (e) Fraction (%) of the memory accesses.

Then, the next pointer points to an address which is not aligned with the cache line size. In order to fetch MAB2, the memory request generator in the DC needs to generate two different memory requests causing *fragmentation*. The third pointer points at the first MAB1, meaning that MAB3 has the same content as MAB1 (intra-match, explained in Sec. 4). Note that, instead of reusing MAB1 fetched for the first access, the DC just fetches MAB1 again, causing *locality loss*. As a result, another request is generated to fetch the same cache line, which has just been requested two requests before. After *two frames*, the fourth pointer points at an address in the first frame buffer, which has the same content as MAB1 (inter-match). A request is sent to DRAM to fetch MAB1, which may cause a row activation due to accessing a different memory region. In total, we need 5 memory requests and a potential row activation to fetch just 4 mabs. Although we see a lot of content locality across these four mabs, they will not be exploited by the current layout and conventional DC.

5.1 Display Cache and MACH Buffer

To fix this under-utilization of locality, we propose two new hardware enhancements oriented towards exploiting the content locality: the *display cache* and the *MACH buffer*. The display cache is a direct-mapped cache, which is indexed by **any** pointers, and the value is a cache line (64 bytes). As shown in Fig. 10a, the display cache stores the recent memory accesses generated by the DC.

The MACH buffer is built by using the MACHs in the video decoder. Note that, a MACH is not changed once its corresponding frame finishes decoding and writes back into the frame buffer. Therefore, once all the mabs/gabs are written into the memory, we dump its MACH into the memory. When the DC sends out the frame, it first picks up the frame's MACH. Now, from each entry of MACH, the DC can find the pointer of mab/gab, and then prefetch the whole mab/gab into a MACH buffer, as illustrated in Fig. 10b. The buffer is indexed by the digest, and the value now is the mab.

Note that, in the current memory layout, the display has only the pointers, not the digests of each mab/gab, whereas the buffer is tagged by the digests rather than the pointers. We cannot simply use the pointers as indexes since, in the VD, each set of MACH is originally indexed by digests. Instead, we change the memory layout again, this time as shown in Fig. 9c(iii). In this new layout, the pointers are mixed with the digests. When decoding, instead of storing the pointers, we store all the inter-matches using digests. Note that these inter-matches can be found in the display's MACH-buffer (it will not be changed), and as a result, we save memory accesses. For those mabs/gabs with no-matches or intra-matches, we still need to store the pointers. We also use a bitmap for macroblocks provided by the VD to differentiate which is stored as a pointer, and which is stored as digest.

To demonstrate the effect of these two hardware enhancements, let us consider the memory layout shown in Fig. 9c(i), this time using the layout in Fig. 9c(iii). The first request is a no match; so, it will still access the memory, and bring back the whole cache line. Now that we have the display cache, it will insert the cache line brought, including the whole MAB1 and a part of MAB2, into the display cache for potential future reuses. The second pointer will generate two requests; however, since we have already cached the first requested cache line, only one request will be sent to the memory. Furthermore, since the third pointer also generates a request for the cached data, the corresponding memory request is eliminated. The fourth pointer's request is already cached in the MACH, and hence, this request is also eliminated (inter-match). In total, we now issue only 2 memory requests to fetch 4 mabs.

Implementation of the Display Cache and MACH Buffer. The MACH buffers are implemented in the same way as MACHs in the VD are implemented, except that here we store the whole mab/gab, instead of the pointers. As for the display cache, a 16KB direct-mapped cache is sufficient, as shown in Fig. 10c. Also, a mab/gab can at most refer to the previous n number of MACH; as a result, the DC needs to hold a set of frame buffers (8 previous frames in our case), until all the mabs referring to that frame are displayed. We will discuss the extra frame buffers needed in Sec. 6.

5.2 Memory Access Savings in Display

We now show the effect of the display cache and MACH buffer in Fig. 10e. Here, we only give the results with gab. Overall, we save 33.5% of the memory accesses by utilizing the display cache and the MACH buffer. This is similar to what we saved on the VD side, indicating that we exploit most of the content locality. Note that, in terms of the memory access, the metadata contributes to around 15%. It can be observed that, 20% of the memory access savings comes from MACH buffer, while 15.5% comes from the display cache. Note also that, we would need to issue more than 60% additional memory requests if we had simply used the original layout and not employed the display cache or the MACH buffer. Specifically, if we break down the number of gabs into those indexed as digest and those indexed as pointer, we can see that around 38% of the gabs are indexed by digest, demonstrating the need for employing a MACH buffer in the DC. The remaining 62% of gabs are

Key	Video Name	Description	#Frames
V1	SES Astra [77]	TV Test Video	6507
V2	Honey Bees [2]	Timelapse @ 120 fps	5461
V3	Puppies Bath [1]	Home Video; Macro Lens.	3593
V4	NASA [59]	NASA WebCam	1758
V5	Elysium [60]	2013 Movie Trailer	3176
V6	Gone Girl [25]	2014 Movie Trailer	3591
V7	Interstellar [20]	2014 Movie Trailer	2429
V8	007 Skyfall [76]	2012 Movie Trailer	3676
V9	Batman Origins [3]	Adventure Game Video	4702
V10	Battlefield [3]	Shooter Game Video	2899
V11	Call of Duty [3]	Action Game Video	5799
V12	Crysis 3 [3]	Survival Game Video	10147
V13	Dear Esther [3]	Exploration Game Video	1699
V14	Metro LastNight [3]	Atmospheric Game Video	4981
V15	Tomb Raider [3]	Protagonist Game Video	5981
V16	Watch Dogs [3]	Hacking Game Video	3806

Table 1: Workload videos.

indexed by pointers, and more than 45% of them will generate two memory requests due to fragmentation. We can see that those fragmentation requests are saved significantly by the display cache.

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup and Workloads

Workloads: To evaluate our schemes effectively, we selected 16 popular 4K videos, shown in Table 1. Specifically, V1 [77] is a satellite TV test video with stringiest requirements from the video processing system; V2 [2] is a time-lapse video captured at 120fps; V3 [1] is a video captured by macro lens; V4 is a video captured by NASA astronauts; and V5-V8 are trailers of some popular movies. We also tested 8 gaming videos listed as V9-V16 in the table.

Experimental Platform: We use FFmpeg [27] and pintool [49] to gather the video frame and macroblock traces. We model the whole video processing flow on top of Android Emulator to capture system level calls across IPs and the OS. We use GemDroid [19], which is an open-source platform that builds on top of Android Emulator and utilizes Gem5 + DRAMSim2 [14, 52, 75], for micro-architecture performance simulations. We created a detailed timing model of the video decoder IP, as described in [99]. Also, we modeled the display and frame-buffers as described in [37]. Our memory is configured after an LPDDR3 DRAM [52]. For each of the schemes tested, we simulated more than 70K frames. Further details about our platform including the parameters of its various components are given in Table 2. More details can be found at `https://huz123.github.io/mach.html` .

6.2 Results

For each of the 16 videos, we evaluate six schemes `Baseline`, `Batching`, `Racing`, `Race-to-Sleep`, `Race-to-Sleep + MAB (MAB)`, and `Race-to-Sleep + GAB(GAB)` and report the *normalized* energy consumption results in Fig. 11. The `Baseline` is the no-batch and no-race scheme. All the results presented have been normalized with respect to the baseline scheme. In these graphs, `MAB` means that we apply MACH on both the video decoder (VD) and the display controller (DC) using *mab*, whereas `GAB` means that we apply MACH on the VD and the DC using *gab*. Note that our results account for the energy consumed by the DC and the energy overheads due to MACH for both `MAB` and `GAB`. To better explain the effect of different schemes, we split the energy consumption bars into nine parts: DC, memory background activities, VD processing, sleep, short slack, memory burst, memory `Act/Pre`, power state transitions and overheads of `MAB/GAB`. Out of these, DC and memory background overheads are pretty much constant across all schemes and orthogonal to our discussion. We also plot the MACH+Display Optimization result to demonstrate the benefits of this proposed optimization. Overall, `Race-to-Sleep` saves 11.3% energy; `MAB` saves 12.5% energy, and `GAB` saves 21% energy, compared to the baseline setup.

Batching, Racing and Race-to-Sleep (Sec. 3):
In Fig. 11, across the spectrum, `Batching` reduces the transition energy. Also, as batching increases the total time spent in sleep/deep-sleep states, the total energy savings are significant. An exception is V4, where there is only marginal energy reduction due to 6% additional energy being consumed in PStates because of the short slacks (this will be handled by `Racing`). On an average, `Batching` saves around 7% energy.

With `Racing`, we observe around 20% reduction in the energy consumed by the `Act/Pre` commands in memory. However, since we increased the VD's frequency, the VD's energy increased by around 11% due to the cubic impact of frequency and voltage. Also, since the transitions are to/from higher P states, and as there is no batching to reduce per frame transition time, the average transition energy increases. This leads to a net increase of 12% in overall energy. We see that applying `Racing` in a stand-alone fashion does not save energy, and in cases such as V3 and V6 more energy is spent.

The effect of `Race-to-Sleep` is plotted in Fig. 11 as the fourth bar. As this inherits the benefits of both `Batching` and `Racing`, both transition energy and memory `Act/Pre` energy are reduced. Overall, we save about 11% energy compared to the baseline. However, we still see around 53% energy being consumed on the memory side. We next discuss the results of applying `MAB` and `GAB` on top of `Race-to-Sleep`.

Race-to-Sleep + MAB and GAB (Sec. 4 and Sec. 5):
Benefits of `MAB` are shown as the fifth bar in Fig. 11. One can observe that the memory energy reduced by 10%, compared to `Race-to-Sleep`. Compared to `Race-to-Sleep`, with `MAB`, *both memory* `Act/Pre` *and memory burst energy are reduced* since there are fewer memory accesses. However, the `MAB` scheme introduces MACH and its auxiliary hardware units (shown in Table 2), which cause extra energy overheads (3% of the total energy). For example, in V9, we observe an increase in net energy consumed since `MAB` could not save much energy to compensate for the energy overheads it brings. Therefore, the overall energy reduction is around 12.5% compared to the baseline.

`GAB` reduced a significant fraction of memory accesses, leading to an average energy reduction of 21%(up to 33% in V8) compared to the baseline. `GAB` outperforms all other schemes in every scenario. Note that the net energy saving follows the memory access savings trend as shown in Fig. 9a and Fig. 10e, respectively. Comparing memory energy in `GAB` to `Race-to-Sleep` scheme, it can be seen that `GAB` saves around 30% energy, which corresponds to the number of memory accesses reduced by `GAB`.

MACH Can Benefit Existing Compression Schemes: To complete our study, we also study the benefits of using MACH with a state-of-the-art color compression scheme. Color compression methods such as Delta Color Compression (DCC) [5, 61] are available in commercial platforms. They compute the deltas between pixels in a block, and store the pixels in compressed formats to reduce memory bandwidth. Note that this technique is an "intra-block" compression method, whereas our scheme explores "inter-block" reuse within/across frames. Therefore, DCC and our scheme are orthogonal to each other, and consequently, they can

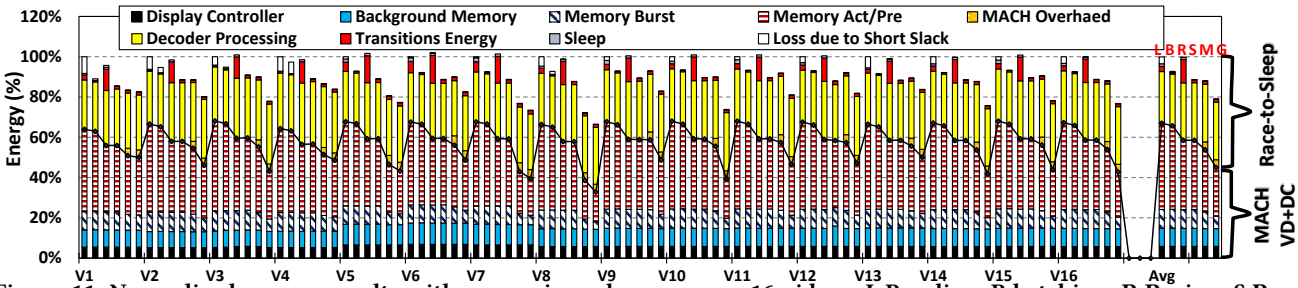


Figure 11: Normalized energy results with our recipes shown across 16 videos. L:Baseline; B:batching; R:Racing; S:Race-to-Sleep; M:MAB; G:GAB. Normalized to baseline. (the lower the better).

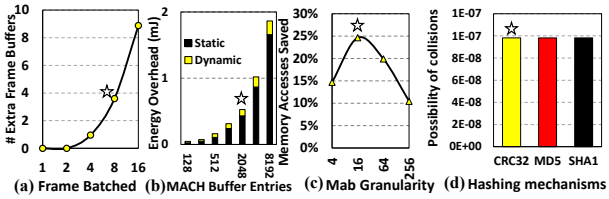


Figure 12: Sensitivities to (a) the number of extra frame buffers when using gab; (b) the number of MACH buffer entries; (c) size of mab; and (d) different hashing mechanism.

be safely combined. In fact, we experimented with such combined scheme (GAB + DCC[61]), and the results show that the combined scheme provides about 18% more memory bandwidth savings compared to plain DCC. These savings are from exploiting the inter-block locality that exists across mabs.

6.3 Sensitivity Study and Overhead

Sensitivity Results: Fig. 12(a) shows the extra number of frame buffers (beyond Triple Buffering) needed as the number of MACHs increases after applying GAB. We choose 8 MACHs for two reasons: 1) when using 4 MACHs, the transition energy still contributes to 2% of the total energy, and 2) using 16 MACHs, the extra frame buffers needed is around 300MB in memory. From Fig. 12(b), we see that the energy consumed exponentially increases when the total number of MACH buffers increase. We propose using 2K MACH buffers as a trade-off between energy costs and probability of finding mab/gab matches (Fig. 9b). Mab/gab can potentially be of any size, hence we performed a study on V14 which showed that a 4x4 region is the optimal size (see Fig. 12(c)).

Across different hashes (CRC32 [65], MD5 [73] and SHA1 [15]) that we studied, we did not observe any major differences. It is to be noted that the probability of a collision on one gab is extremely low, resulting in only one 4x4 block colliding in around 200 frames, as shown in Fig. 12(d).

To further reduce collisions, we now propose a deeper hashing mechanism and an additional COllision-MACH (referred to as CO-MACH) to store all collided entries for the currently decoding frame. To detect collisions in our hashing scheme, we combine the original CRC32 with an additional CRC16 [90], making the hash 48-bits deep. However, if we simply use a 48-bit hash and dump it as the digest for each matching mab (to use at the display side), the memory bandwidth would increase substantially and the cost of MACH buffer at display would also increase. To address this issue, we do not send the extra CRC16 hash as digest to the memory. We instead keep the original 8 MACHs as the same and propose *an additional collision detecting 16-bit auxiliary field (using CRC16) in the data fields* of each MACH.

Note that, in the original scenario, if two different mabs have the same CRC32, the collision can be neither detected nor fixed. Now, when two mabs have the same CRC32, we further check if their corresponding CRC16 hashes are also the same or not. If not, we have detected a collision of CRC32. As a result, we insert this mab as a new entry in the CO-MACH cache and use a 48-bit tag concatenating both CRC32 and CRC16. Note also that, the data part of the CO-MACH cache is still the pointer to the mab, as described earlier. Now, to index into MACHs for each new mab produced at the VD, we search all the nine caches (8MACHs + 1 CO-MACH cache), but the rest of the process remains the same. At the memory and display side, we do not need to dump or use the CRC16 data fields anymore. So, we do not increase the memory bandwidth requirements.

We emphasize that, with this strategy, while the memory bandwidth does not increase, the collision is reduced to practically *zero* in all the 16 video workloads tested. The additional hardware cost includes a 5.5KB (4KB for 16 bit CRC fields in 8 MACHs, and 1.5KB for the CO-MACH) SRAM in MACH and a hardware CRC16 generator, increasing MACH power consumption by only 1.4mW [56].

Hardware Overheads: To batch 8 frames using GAB, we need four more frame buffers, increasing the memory requirement from 72MB to 168MB. To utilize MACH in the VD and DC, we need an 8KB MACH, a 16KB display cache, and a 96KB MACH buffer. The total resulting power consumption is around 35mW (5.7mW from 8KB MACH, 4.1mW from 16KB display cache, and 25.4mW from MACH buffer), which is very low compared to the power envelopes of state-of-the-art SoC architectures [94]. The area cost of our additional SRAM is $0.68mm^2$, which is less than 0.5% of a modern mobile SoC [94]. Further, we do not expect the vector units and the CRC32 to incur any significant cost.

6.4 Other Potential Applications of MACH

In this paper, we mainly discussed how MACH can be employed on the video processing pipeline. Note that, the high-level MACH concept can be applied to other IP pipelines in the SoC as well. For example, the video recording/encoding pipeline also has a process sequence which is similar (but reversed order) to that of the video processing pipeline, and also exhibits rich value locality. The video recording pipeline continuously captures frames from camera, and uses memory to pass each frame to the video encoder. In this case, we can employ MACH at both the camera and video encoder to exploit the content similarity of the frames, and thus reduce the memory accesses at the camera and the video encoder.

Another potential use-case is the graphics pipeline. This pipeline renders frames using the graphics processing units (GPUs)

Parameter	Value
Android	Google Android Emulator; Google Nexus 7.
DRAM Timing	2GB, 2 channels; 1 ranks per channel; 8 Banks per rank; [52] Vdd = 1.2V; tCL,tRP,tRCD = 12, 18, 18 ns; 800Mhz; RoRaBaCoCh;
VD Display	0.30W@150Mhz; 0.69W@300Mhz [99] Display Controller, 3840x2160@60hz, 0.12W[16].
MACH Auxiliaries [56]	8 MACH Cache with 256 entries, 4-way each; Total: 2K entries; 8KB MACH@VD : 1.9mW(Static); 3.8mW(Dynamic), 0.2ns. 96KB MACH Buffer@DC: 24mW(Static); 1.4mW(Dynamic), 0.4ns. 16KB Display Cache@DC: 3.6mW(Static); 0.5mW(Dynamic), 0.2ns.

Table 2: Simulation Configuration

on the SoC, and sends the rendered frames to the display via memory. Note that many applications (e.g., mobile games, VR, AR etc.) employ a graphics pipeline. Modern mobile games (e.g., our V9-16 workloads) also capture content similarity at the display. Note that, these games typically render at 60fps or more, and require significant memory bandwidth as well. In the future, we expect to accommodate MACH at both graphics GPU and display sides for any frame-based workload scenarios, to save memory bandwidth by exploiting content similarity.

7 RELATED WORK

Prior works related to this paper are summarized in the following three areas:

Energy Optimization: Multiple energy optimization techniques have been studied in CPU/GPU/Memory domains [4, 7, 13, 24, 42, 45, 48, 50, 58, 62, 79, 81, 83–85, 92, 93, 97, 101, 102]. Particularly *Race-to-Sleep* (or race-to-idle) has been studied in [4, 50, 67, 68] in the context of traditional CPU/GPU domains from both the energy and thermal perspectives. [67, 68] discussed how to burst a CPU for a short period of time to achieve thermal benefits. [63] proposed cooperative boosting (CB) to balance the thermal budget and performance demand between CPU and GPU on a chip. [17, 41, 86] showed examples of batching and pipelining at a macroblock level, whereas our scheme batches (buffers) and races at a frame level. Contrary to computational sprinting, previous schemes such as [57, 66] explored the benefits of scaling down the frequency to leverage the slack available in decoding every frame. While they obtain energy improvements, these benefits come at the cost of frame-drops. Furthermore, none of these works have considered batching frames for performance gain.

On top of gaining potential thermal efficiency in mobile devices [67, 68], employing race-to-sleep in video decoding provides two additional advantages. First, as a frame's decoding time is unpredictable, especially when decoding larger frames (4K or 8K), utilizing a history-based slack prediction scheme [57] can lead to frame drops. By employing *racing* and *batching*, we create more slack for each frame, thereby reducing the possibility of frame drops. Second, we consider system-level energy efficiency by accounting for both video decoder and memory as discussed in Sec.3.

Exploiting Locality in Streaming Applications: Streaming applications such as video processing and camera recording expose rich computation parallelism [23, 44]. Modern video decoders usually employ a cache [38, 43, 72, 99, 100] to capture the address locality during the computation (decoding) phase [12, 18, 21, 46, 82].

In the CPU/GPU domain, value locality [47, 53, 64, 70, 71, 91, 98] has been well studied. [98] took advantage of value locality in CPU benchmarks to reduce cache misses. [96] leveraged delta-compression to reduce network traffic in the NoC. [64, 70] further studied value locality to compress cache lines. [53] uses *approximate content similarity* in CPU cache by exploiting programmer hints on data-approximations. However, to the best of our knowledge, we are the first to take advantage of value/content locality in the context of *video streaming applications* in an energy-constrained handheld platforms. As shown in Sec. 4, video processing exhibits rich value locality. Our proposed MACH converts this value locality into address locality in video decoding; and, this is exploited by the MACH buffers employed on the DC side.

Optimizations on Display: There exist many prior works, from both academia and industry, on compressing a frame [5, 8, 9, 26, 61, 69, 78], reducing power in display hardware [35, 36], and developing software optimizations [34, 51]. In general, compressing a frame leads to heavy computational overheads [8, 78]. Delta Color Compression (DCC) methods such as [5, 61] are available in commercial platforms. In general, DCC is an "intra-block" compression method, whereas our scheme explores "inter-block" reuse within/across frames. Hence, our scheme can work on top of DCC, as explained in Sec. 6. Industrial efforts such as [9, 35] use checksum to identify whether a part of frame is changed compared to *the previous frame*, in an attempt to reduce the amount of memory bandwidth. Compared to the previous works, which are all mab-based, our lossless schemes have the following advantages: 1) MACH does not compress any mab; instead, it reuses the mabs that are already stored in the memory, and doing so leads to minimal computation overhead; 2) MACH makes full use of both the intra-frame matches and inter-frame matches; and 3) compared to the mab-based solution, the gab-based solution generates consistently better results. Also, compression optimizations can still be applied in conjunction with MACH for additional benefits.

8 CONCLUSIONS

In this paper, we have presented a coordinated three-pronged approach for improving the energy efficiency of video streaming applications in handheld platforms. In contrast to the conventional per frame processing adopted in video decoders, our first scheme, called *race-to-sleep*, employs a batch processing of video frames along with frequency boosting to avoid any frame drop and reduce energy consumption by transitioning to a prolonged deep-sleep mode. Our second scheme, *content caching*, intelligently exploits the content similarity of the video frames at a smaller granularity, and proposes an effective cache architecture for minimizing the decoding memory bandwidth and space demands imposed by race-to-sleep. Finally, our third scheme, *display caching*, employs a similar content caching technique for the display hardware to further reduce the memory pressure. These three schemes, which need minimal hardware overhead and software modifications, are integrated to develop an end-to-end video processing framework and have been extensively evaluated with off-the-shelf videos. Our experimental analysis shows that each of these techniques individually contributes to energy savings, and together they can provide 21% energy saving in processing videos. Given the dominance of video streaming applications as a core workload for energy-constrained handhelds, these savings are significant.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants #1213052, #1302557, #1317560, #1320478, #1409095, #1439021, #1439057, #1626251, #1629129, #1629915, #1714389, #1526750, and Intel. We would also like to thank Jack Sampson for his feedback on this paper.

REFERENCES

[1] 4K SAMPLES. 2012. Puppies Bath in 4K. https://goo.gl/sbq8AD. (2012). Accessed: 2017-03-15.

[2] 4K SAMPLES. 2015. Honey Bees 96fps In 4K (ULTRA HD). https://goo.gl/u0pTz9. (2015). Accessed: 2017-03-15.

[3] 4K SAMPLES. 2017. 4K Gaming Montage. https://goo.gl/qJExOF. (2017). Accessed: 2017-03-15.

[4] Susanne Albers and Antonios Antoniadis. 2014. Race to Idle: New Algorithms for Speed Scaling with a Sleep State. *ACM Trans. Algorithms* (2014), 9:1–9:31.

[5] AMD. 2016. RADEON: Dissecting the Polaris Architecture. *AMD Whitepaper* (2016).

[6] Pablo Ameigeiras, Juan J. Ramos-Munoz, Jorge Navarro-Ortiz, and J.M. Lopez-Soler. 2012. Analysis and modelling of YouTube traffic. *Transactions on Emerging Telecommunications Technologies* (2012).

[7] Antonios Antoniadis, Chien-Chung Huang, and Sebastian Ott. 2015. A Fully Polynomial-time Approximation Scheme for Speed Scaling with Sleep State. In *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1102–1113.

[8] ARM. 2016. ARM Frame Buffer Compression. https://goo.gl/ETnZTF. (2016).

[9] ARM. 2016. Transaction Elimination. https://goo.gl/91cDSG. (2016).

[10] ARM. 2017. big LITTLE Technology. https://www.arm.com/products/processors/technologies/biglittleprocessing.php. (2017).

[11] ARM. 2017. NEON. https://www.arm.com/products/processors/technologies/neon.php. (2017).

[12] Arnaldo Azevedo and Ben Juurlink. 2009. An Efficient Software Cache for H.264 Motion Compensation. In *Proceedings of the 11th International Conference on System-on-chip (SOC)*. 147–150.

[13] Peter Bailis, Vijay Janapa Reddi, Sanjay Gandhi, David Brooks, and Margo Seltzer. 2011. Dimetrodon: Processor-level Preventive Thermal Management via Idle Cycle Injection. In *Proceedings of the 48th Design Automation Conference (DAC)*. 89–94.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* (2011), 1–7.

[15] James H Burrows. 1995. *Secure Hash Standard*. Technical Report. DTIC Document.

[16] Aaron Carroll and Gernot Heiser. 2010. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*.

[17] Tung-Chien Chen, Yu-Wen Huang, and Liang-Gee Chen. 2004. Analysis and Design of Macroblock Pipelining for H.264/AVC VLSI Architecture. In *Proceedings of 2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No.04CH37512)*. II–273–6 Vol.2.

[18] X. Chen, Peilin Liu, Jiayi Zhu, Dajiang Zhou, and S. Goto. 2009. Block-pipelining Cache for Motion Compensation in High Definition H.264/AVC Video Decoder. In *Proceedings of 2009 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1069–1072.

[19] Nachiappan Chidambaram Nachiappan, Praveen Yedlapalli, Niranjan Soundararajan, Mahmut Taylan Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2014. GemDroid: A Framework to Evaluate Mobile Platforms. In *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 355–366.

[20] Christopher Nolan. 2014. Interstellar Movie - Official Trailer. https://goo.gl/I4bpGF. (2014). Accessed: 2017-03-15.

[21] Tzu-Der Chuang, Lo-Mei Chang, Tsai-Wei Chiu, Yi-Hau Chen, and L. G. Chen. 2009. Bandwidth-efficient Cache-based Motion Compensation Architecture with DRAM-friendly Data Access Control. In *Proceedings of 2009 IEEE International Conference on Acoustics, Speech and Signal Processing (CASSP)*. 2009–2012.

[22] CISCO. 2017. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016 - 2021 White Paper. "http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html". (2017).

[23] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. 2014. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *IEEE Micro* (2014), 34–43.

[24] Anup Das, Geoff V. Merrett, and Bashir M. Al-Hashimi. 2016. The Slowdown or Race-to-idle Question: Workload-aware Energy Optimization of SMT Multicore Platforms Under Process Variation. In *Proceedings of 2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 535–538.

[25] David Fincher. 2014. Gone Girl Official 2 (2014) Ben Affleck, Rosamund Pike HD. https://goo.gl/EBHqmQ. (2014). Accessed: 2017-03-15.

[26] K.C. Dyke. 2013. Method for Reducing Framebuffer Memory Accesses. (2013). http://www.google.tl/patents/US8358314 US Patent 8,358,314.

[27] FFmpeg. 2016. A Complete, Cross-platform Solution to Record, Convert and Stream Audio and Video. "https://ffmpeg.org/". (2016).

[28] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. 2008. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel white paper* (2008).

[29] Google. 2016. BoardConfig.mk. https://android.googlesource.com/device/google/marlin/+/android-7.1.1_r6/marlin/BoardConfig.mk. (2016).

[30] Google. 2016. FramebufferSurface.cpp. https://android.googlesource.com/platform/frameworks/native/+/android-7.1.1_r28/services/surfaceflinger/DisplayHardware/FramebufferSurface.cpp. (2016).

[31] Google. 2016. SurfaceFlinger and Hardware Composer. https://source.android.com/devices/graphics/arch-sf-hwc.html. (2016).

[32] Google. 2017. Android Gralloc Framebuffer. "https://goo.gl/TMuNFU". (2017).

[33] Google. 2017. YouTube for Press. "https://www.youtube.com/yt/about/press/". (2017).

[34] MyungJoo Ham, Inki Dae, and Chanwoo Choi. 2015. LPD: Low Power Display Mechanism for Mobile and Wearable Devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. 587–598.

[35] Kyungtae Han, Zhen Fang, Paul Diefenbaugh, Richard Forand, Ravi R. Iyer, and Donald Newell. 2009. Using Checksum to Reduce Power Consumption of Display Systems for Low-motion Content. In *Proceedings of the 2009 IEEE International Conference on Computer Design (ICCD)*. 47–53.

[36] Kyungtae Han, Alexander W. Min, Nithyananda S. Jeganathan, and Paul S. Diefenbaugh. 2013. A Hybrid Display Frame Buffer Architecture for Energy Efficient Display Subsystems. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design (ISLPED)*.

[37] P.F. Holland, H.G.R. Thirunageswaram, and J.J. Irwin. 2016. Display Pipe Line Buffer Sharing. (2016). https://www.google.com/patents/US20160086298 US Patent App. 14/493,755.

[38] C. T. Huang, M. Tikekar, C. Juvekar, V. Sze, and A. Chandrakasan. 2013. A 249Mpixel/s HEVC Video-decoder Chip for Quad Full HD Applications. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*. 162–163.

[39] Aamer Jaleel. 2010. Memory Characterization of Workloads Using Instrumentation-driven Simulation. *Web Copy: http://www.jaleels.org/ajaleel/workload* (2010).

[40] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012. A QoS-aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*. 850–855.

[41] G. Jin, J. S. Jung, and H. J. Lee. 2007. An Efficient Pipelined Architecture for H.264/AVC Intra Frame Processing. In *Proceedings of 2007 IEEE International Symposium on Circuits and Systems*. 1605–1608.

[42] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Nilardish Chatterjee, Steve Keckler, Mahmut T. Kandemir, and Chita R. Das. 2015. Anatomy of GPU Memory System for Multi-Application Execution. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS)*. 223–234.

[43] C. C. Ju, T. M. Liu, K. B. Lee, Y. C. Chang, H. L. Chou, C. M. Wang, T. H. Wu, H. M. Lin, Y. H. Huang, C. Y. Cheng, T. A. Lin, C. C. Chen, Y. K. Lin, M. H. Chiu, W. C. Li, S. J. Wang, Y. C. Lai, P. Chao, C. D. Chien, M. J. Hu, P. H. Wang, Y. C. Huang, S. H. Chuang, L. F. Chen, H. Y. Lin, M. L. Wu, and C. H. Chen. 2016. A 0.5 nJ/Pixel 4 K H.265/HEVC Codec LSI for Multi-Format Smartphone Applications. *IEEE Journal of Solid-State Circuits* (2016).

[44] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. 2002. The Imagine Stream Processor. In *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*. 282–.

[45] Onur Kayiran, Adwait Jog, Ashutosh Pattnaik, Rachata Ausavarungnirun, Xulong Tang, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. 2016. uC-States: Fine-grained GPU Datapath Power Management. In *Proceedings of 2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 17–30.

[46] J. H. Kim, G. H. Hyun, and H. J. Lee. 2007. Cache Organizations for H.264/AVC Motion Compensation. In *Proceedings of 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*. 534–541.

[47] Marios Kleanthous and Yiannakis Sazeides. 2011. CATCH: A Mechanism for Dynamically Detecting Cache-content-duplication in Instruction Caches. *ACM Trans. Archit. Code Optim.* (2011).

[48] Jagadish B. Kotra, Narges Shahidi, Zeshan A. Chishti, and Mahmut T. Kandemir. 2017. Hardware-Software Co-design to Mitigate DRAM Refresh Overheads: A Case for Refresh-Aware Process Scheduling. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 723–736.

[49] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 190–200.

[50] Paul McKenney. 2012. A big.LITTLE scheduler update. https://lwn.net/Articles/501501/. (2012).

[51] Hongyu Miao and Felix Xiaozhu Lin. 2016. Tell Your Graphics Stack That the Display Is Circular. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*.

[52] MICRON. 2014. Production Data Sheet: 8Gb, 16Gb: 253-Ball, Dual-Channel 2C0F Mobile LPDDR3 SDRAM (pdf). "https://www.micron.com/resource-details/75340edb-6b8e-4c43-968a-2323d5127aa6". (2014).

[53] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. DoppelgÄNger: A Cache for Approximate Computing. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. 50–61.

[54] Thomas Moscibroda and Onur Mutlu. 2007. Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*.

[55] D. Mukherjee, J. Bankoski, A. Grange, J. Han, J. Koleszar, P. Wilkins, Y. Xu, and R. Bultje. 2013. The Latest Open-source Video Codec VP9 - An Overview and Preliminary Results. In *2013 Picture Coding Symposium (PCS)*.

[56] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A Tool to Model Large Caches. *HP Laboratories* (2009).

[57] N. C. Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das. 2015. Domain Knowledge Based Energy Management in Handhelds. In *Proceedings of 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 150–160.

[58] Nachiappan Chidambaram Nachiappan, Haibo Zhang, Jihyun Ryoo, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T. Kandemir, Ravi Iyer, and Chita R. Das. 2015. VIP: Virtualizing IP Chains on Handheld Platforms. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. 655–667.

[59] NASA. 2015. Ultra High Definition Video from the International Space Station. https://archive.org/details/NASA-Ultra-High-Definition. (2015). Accessed: 2017-03-15.

[60] Neill Blomkamp. 2013. Elysium 2013 2160p 1 Minute Sample Footage. https://goo.gl/E6QJ0V. (2013). Accessed: 2017-03-15.

[61] Nvidia. 2016. NVIDIA GeDorce GTX 1080. *Nvidia Whitepaper* (2016).

[62] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)*. 31–44.

[63] Indrani Paul, Srilatha Manne, Manish Arora, W. Lloyd Bircher, and Sudhakar Yalamanchili. 2013. Cooperative Boosting: Needy Versus Greedy Power Management. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. 285–296.

[64] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 377–388.

[65] W. W. Peterson and D. T. Brown. 1961. Cyclic Codes for Error Detection. *Proceedings of the IRE* (1961).

[66] Erwan Raffin, Erwan Nogues, Wassim Hamidouche, Seppo Tomperi, Maxime Pelcat, and Daniel Menard. 2016. Low Power HEVC Software Decoder for Mobile Devices. *Journal of Real-Time Image Processing* (2016).

[67] Arun Raghavan, Laurel Emurian, Lei Shao, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M.K. Martin. 2013. Computational Sprinting on a Hardware/Software Testbed. *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013), 155–166.

[68] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. 2012. Computational Sprinting. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA)*. 1–12.

[69] J. Rasmusson, T. Akenine-Möller, J. Hasselgren, and J. Munkberg. 2011. Frame Buffer Compression and Decompression Method for Graphics Rendering. (2011). https://www.google.com/patents/US8031937 US Patent 8,031,937.

[70] Prasanna Venkatesh Rengasamy and Madhu Mutyam. 2014. Using Packet Information for Efficient Communication in NoCs. In *Networks-on-Chip (NoCS), 2014 Eighth IEEE/ACM International Symposium on*. 143–150.

[71] Prasanna Venkatesh Rengasamy, Anand Sivasubramaniam, Mahmut T. Kandemir, and Chita R. Das. 2015. Exploiting Staleness for Approximating Loads on CMPs. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*. 343–354.

[72] Prasanna Venkatesh Rengasamy, Haibo Zhang, Nachiappan Chidhambaram Nachiappan, Shulin Zhao, Anand Sivasubramaniam, Mahmut Kandemir, and Chita R. Das. 2017. Characterizing Diverse Handheld apps for Customized Hardware Acceleration. In *Proceedings of 2017 IEEE International Symposium on Workload Characterization (IISWC)*.

[73] R. Rivest. 1992. The MD5 Message-Digest Algorithm. https://tools.ietf.org/html/rfc1321. (1992).

[74] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory Access Scheduling. *SIGARCH Comput. Archit. News* (2000), 128–138.

[75] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* (2011), 16–19.

[76] Sam Mendes. 2012. Skyfall 4K (UHD) . https://goo.gl/LgnZ15. (2012). Accessed: 2017-03-15.

[77] SES Astra. 2016. SES Astra UHD Test 2160p UHDTV Free 4K Sample Footage. https://goo.gl/S7i4nN. (2016). Accessed: 2017-03-15.

[78] Hojun Shim, Naehyuck Chang, and Massoud Pedram. 2004. A Compressed Frame Buffer to Reduce Display Power Consumption in Mobile Systems. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference (ASP-DAC)*. 818–823.

[79] Akbar Shrifi, Wei Ding, Diana Guttman, Hui Zhao, Xulong Tang, Mahmut Kandemir, and Chita Das. 2017. DEMM: A Dynamic Energy-saving Mechanism for Multicore Memories. In *Proceedings of 2017 IEEE 25th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[80] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand. 2012. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology* (2012).

[81] K. Swaminathan, J. Kotra, H. Liu, J. Sampson, M. Kandemir, and V. Narayanan. 2015. Thermal-Aware Application Scheduling on Device-Heterogeneous Embedded Architectures. In *2015 28th International Conference on VLSI Design*. 221–226.

[82] Xulong Tang, Hong An, Gongjin Sun, and Dongrui Fan. 2013. A Video Coding Benchmark Suite for Evaluation of Processor Capability. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 101–116.

[83] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *Proceedings of 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[84] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. 2017. Phoenix: A Constraint-Aware Scheduler for Heterogeneous Datacenters. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 977–987.

[85] Andrea Tilli, Andrea Bartolini, Matteo Cacciari, and Luca Benini. 2012. Don'T Burn Your Mobile!: Safe Computational Re-sprinting via Model Predictive Control. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*.

[86] Tang-Hsun Tu and Chih-Wen Hsueh. 2010. Batch-Pipelining for H.264 Decoding on Multicore Systems. In *Proceedings of the 2010 Data Compression Conference*. 553–.

[87] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. 2016. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *ACM Trans. Archit. Code Optim.* (2016).

[88] Geert Uytterhoeve. 2001. The Frame Buffer Device. https://www.kernel.org/doc/Documentation/fb/framebuffer.txt. (2001).

[89] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. 2003. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology* (2003).

[90] J. K. Wolf and D. Chun. 1994. The Single Burst Error Detection Performance of Binary Cyclic Codes. *IEEE Transactions on Communications* (1994), 11–13.

[91] Jun Yang and Rajiv Gupta. 2002. Energy Efficient Frequent Value Data Cache Design. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 197–207.

[92] Praveen Yedlapalli, Nachiappan Chidambaram Nachiappan, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T. Kandemir, and Chita R. Das. 2014. Short-Circuiting Memory Traffic in Handheld Platforms. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 166–177.

[93] Seyed Majid Zahedi, Songchun Fan, Matthew Faw, Elijah Cole, and Benjamin C. Lee. 2017. Computational Sprinting: Architecture, Dynamics, and Strategies. *ACM Trans. Comput. Syst.* (2017).

[94] R. Zahir. 2012. Medfield smartphone SOC Intel Atom Z2460 processor. In *2012 IEEE Hot Chips 24 Symposium (HCS)*. 1–20.

[95] Rumi Zahir, Mark Ewert, and Hari Seshadri. 2013. The Medfield Smartphone: Intel Architecture in a Handheld Form Factor. *IEEE Micro* (2013).

[96] J. Zhan, M. Poremba, Y. Xu, and Y. Xie. 2014. NoD: Leveraging delta compression for end-to-end memory access in NoC based multicores. In *Proceedings of 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 586–591.

[97] Haibo Zhang, Wenting Han, Feng Li, Songtao He, Yichao Cheng, Hong An, and Zhitao Chen. 2014. A Criticality-Aware DVFS Runtime Utility for Optimizing Power Efficiency of Multithreaded Applications. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*.

841–848.

[98] Youtao Zhang, Jun Yang, and Rajiv Gupta. 2000. Frequent Value Locality and Value-centric Data Cache Design. In *SIGPLAN Not.* 150–159.

[99] D. Zhou, S. Wang, H. Sun, J. Zhou, J. Zhu, Y. Zhao, J. Zhou, S. Zhang, S. Kimura, T. Yoshimura, and S. Goto. 2016. 14.7 A 4Gpixel/s 8/10b H.265/HEVC Video Decoder Chip for 8K Ultra HD Applications. In *Proceedings of 2016 IEEE International Solid-State Circuits Conference (ISSCC)*.

[100] D. Zhou, J. Zhou, X. He, J. Zhu, J. Kong, P. Liu, and S. Goto. 2011. A 530 Mpixels/s 4096x2160 60fps H.264/AVC High Profile Video Decoder Chip. *IEEE Journal of Solid-State Circuits* (2011), 777–788.

[101] Yuhao Zhu and Vijay Janapa Reddi. 2013. High-performance and Energy-efficient Mobile Web Browsing on Big/little Systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 13–24.

[102] Yuhao Zhu and Vijay Janapa Reddi. 2017. Optimizing General-Purpose CPUs for Energy-Efficient Mobile Web Computing. *ACM Trans. Comput. Syst.* (2017), 1:1–1:31.