Porting and optimizing the JPEG decoder on a CompSoC multi-core platform

Agarwal Nitesh 0981729, Giotas Dimitris 0924206, Rol Marijn 0776574,
Vadivel Kanishkan 0878400, Zhenyuan Liu 0899949

I. Introduction

A. The JPEG decoder

The JPEG decoding algorithm comprises three fundamental functions; Unpack, Inverse-Discrete Cosine Transform (IDCT) and Color Conversion (CC). These functions constitute the bulk of the workload of the JPEG decoder and they will, therefore, be our main focus when porting and optimizing the decoder on the CompSoC platform [1]. As seen in Figure 1, a large portion of the algorithm is inherently sequential thus limiting the benefits of porting it to a multi-core environment. Moreover, the JPEG decoder performance and resource requirements are heavily dependent on its input. Therefore, finding a one-fits-all type of solution is an almost futile task.

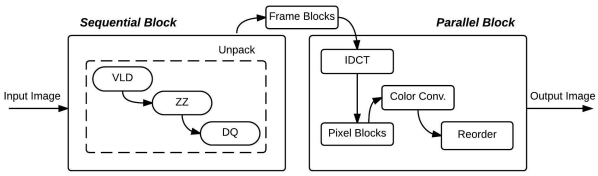

Fig. 1. JPEG decoder steps

B. The CompSoC platform

The target platform for porting the JPEG decoder is the CompSoC platform. CompSoC consists of 4 Microblaze cores, each with local on-tile distributed shared and non-shared memories, one global shared DRAM memory and a Daelite Network-On-Chip (NoC) that facilitates a means of communication between all the available resources of the platform. Data can be communicated among resources either via Memory Mapped I/O (MMIO) or via Direct Memory Access (DMA) controllers. A brief summary of the specifications of each core can be seen in Table I.

Table I. CompSoC tiles specification

	Frequency	CmemOut	CmemIn	#DMAs
Core 1, 2	60 KHz	32 KB	32 KB	1
Core 3, 4	120 KHz	8 KB (x2)	8 KB (x2)	2

C. Design choices

1) Data communication: CompSoC allows two ways of communicating data between its resources; DMA and MMIO. If the choice is DMA over MMIO then the type of DMA communication needs to be decided as well as the block sizes of the DMA transfers. Small block sizes will not take full advantage of the DMA burst transfers, whereas very large block sizes may bear superfluous overhead by transferring redundant data. Similarly, blocking DMA transfers may also cause redundant delays, whereas non-blocking may cause erroneous behaviours such as processing corrupt data. On the contrary, MMIO is much more unsophisticated meaning that it is practically transparent to the designer and its tweakable options non-existent. We will later show that the choice between DMA and MMIO can have a detrimental effect on the efficacy of the implementation.

2) Function mapping: On a multi-core platform, function mapping encapsulates the process of breaking down an algorithm into self-sufficient functions and then assigning each of these functions to one of the available cores. Before assigning functions to cores the designer needs to take into account the memory and bandwidth limitations of each core, the execution times of each function on each core and finally the communication delays between the core and the required resources. Figure 1 already provides hints on how to break down the JPEG decoder into smaller functions.

3) Bottlenecks: For data parallelism the sequential part can become a serious bottleneck that unfortunately cannot always be avoided and, as we will later show, incurs significant overhead on the data parallel versions. For function parallelism, however, the sequential part is not a real issue. The potential bottleneck on the function parallel implementation is the communication between the co-operating cores. As seen in Figure 1 there is a considerable amount of communication required between the three functions. Mapping these functions to different cores will inevitably give rise to strong data dependencies between these cores which, if left unchecked, can lead to substantial delays.

II. Profiling

For profiling we chose a single-core implementation of the JPEG decoder and used a variety of images which we believe capture, within reason, the endless variety of input images that the JPEG decoder is expected to operate on. The single core implementation is executed only on cores 1 and 3, since cores 1/2 and cores 3/4 are identical.

A. DMA vs MMIO

We incrementally evaluate the efficiency of the DMA over MMIO by starting with a sub-optimal DMA configuration. This configuration uses blocking DRAM-reads, meaning that a new block of data will be read from DRAM only after the previous block is completely processed. While the DMA read

operation is executed the core blocks until the new data are available. Figure 2 illustrates the results of the measurements.

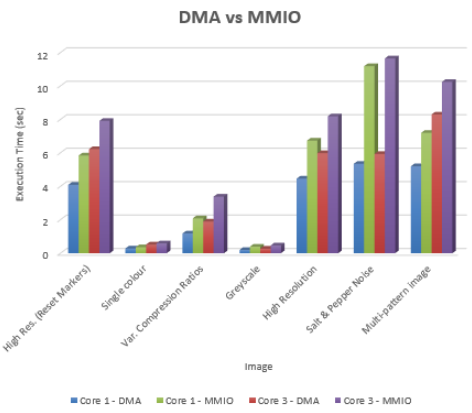

Fig. 2. DMA vs MMIO execution times

The graph of Figure 2 gives profitable insight that can be exploited when making the design choices. The DMA sub-optimal configuration greatly outperforms the MMIO by up to 52% on cores 1/2 and up to 49% on cores 3/4. The results are compellingly in favor of the DMA and show that there is no need to compare DMA with MMIO any further. We also see that cores 3/4 are consistently outperformed by their counterparts, cores 1/2, for both DMA and MMIO despite operating on twice the frequency and the same data set. This imbalance will have to be addressed when attempting any parallel version. In data parallel versions this discrepancy can cripple the core utilization, rendering the implementation inefficient. Similarly, in a function parallel, uneven execution times of the pipeline stages may cause blocking and in general make the pipeline very volatile.

To further strengthen the argument of DMA superiority over MMIO, Figures 3 and 4 illustrate a comparison of the DMA and MMIO read and write bandwidth for core 1 to the DRAM. Even for reading just 4 words from the DRAM, the DMA requires almost half the time of MMIO. Equally important is the writing bandwidth. Not only is the DMA up to 8 times faster at writing to the DMA, it can also take advantage of the posted-writes mechanism provided by CompSoC. Therefore, MMIO was not considered in any version. We finally see that core 1 to DRAM bandwidth saturates at 8KB. Similarly we have measured that core 3 to DRAM bandwidth saturates at 4KB.

B. Function execution times

The JPEG decoder comprises three self-sufficient functions, namely Unpack, IDCT and Color Conversion (CC). The first function, Unpack, is a memory-intensive function that reads the encoded data from the DRAM and decodes the header by using relatively cheap operations. IDCT is computation-intensive and performs multiple complex operations. Finally, CC is balanced in terms of both computation and memory intensity.

Figure 5 depicts the normalized average execution times of Unpack, IDCT and CC for processing one color component of

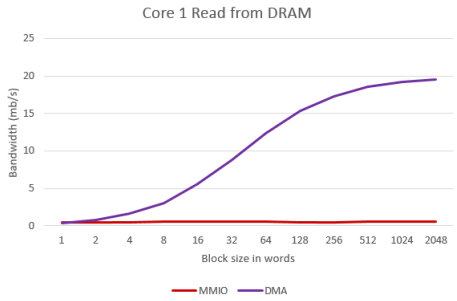

Fig. 3. DMA and MMIO bandwidth for reading

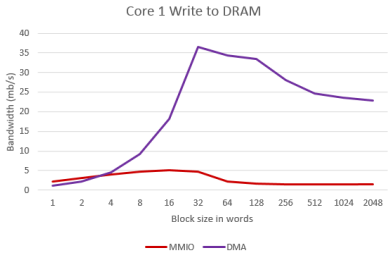

Fig. 4. DMA and MMIO bandwidth for writing

an MCU block on cores 1/2 and cores 3/4 with DMA communication. This metric is a surprisingly accurate approximation since it only considers the computation intensity of a single function rather than the whole decoding process.

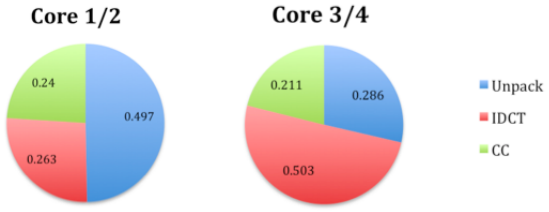

Fig. 5. Normalized average execution times for a single color component

The results illustrated in Figure 5 reveal that CC performs equally good on all cores. However, the computation-heavy IDCT requires less time to execute on cores 1/2 whereas the memory intensive Unpack is faster on cores 3/4. The reason behind this anomaly is the presence of accelerators on cores 1/2. They allow cores 1/2 to perform complex operations a lot more efficiently. On the other hand, the less complex function Unpack is unaffected by the accelerators thus its execution time is limited by the operating frequencies of the cores.

C. Blocking vs Non-Blocking DRAM-Reads

The blocking DRAM-reads scheme described earlier can bear significant overhead to the execution times primarily due to the blocking it causes during the DMA-reads. By moving from blocking to non-blocking DRAM-reads, the cores can operate on their current data while simultaneously new data are being read from the DRAM and moved to local memory. Figure 6 illustrates how the non-blocking DRAM-reads scheme works.

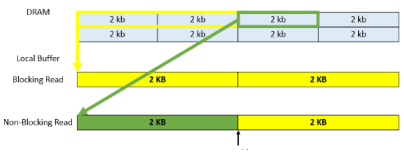

Fig. 6. Non-Blocking DRAM reads

Even though it looks very appealing, the non-blocking DRAM-reads scheme was not implemented in our final implementations. The first reason is that the code complexity for realizing it is too high. For each byte the core reads from its local buffer, a significant amount of complex conditional statements need to be evaluated, crippling the execution times as illustrated in Figure 7.

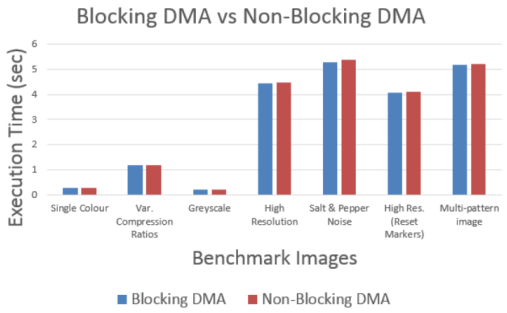

Fig. 7. Non-Blocking DRAM vs Blocking DRAM reads scheme

The second problem is that for some images, especially greyscale, cores process the data on the local buffer faster than the next chunk of data has been read from DRAM and moved to the local buffer. Nonetheless, this issue can be avoided by implementing an extra guard to synchronize DMA-writes and core-reads but this would defeat the purpose of using non-blocking DRAM-reads in the first place.

D. DRAM Read/Write size

As seen in Figure 3, the bandwidth of DMA-reads from the DRAM on all cores saturates at roughly 8KB. If we were to use 8KB reads, the communication memory of cores 3/4 would be fully occupied at all times, leaving no space for storing the processed data. One might suggest using 8KB reads on cores 1/2 and 4KB reads on cores 3/4. However, this would reduce the DRAM accesses of cores 1/2 significantly compared to cores 3/4, effectively reducing their execution times since they would now read from the DRAM at a slightly higher bandwidth with less DRAM accesses. With cores 1/2 already outperforming their counterparts this would augment the discrepancy even further. A 4KB read from the DRAM is a reasonable choice primarily because of the fact that for the function parallel and hybrid versions only cores 3/4 will be accessing the DRAM, since Unpack executes faster when mapped to them.

Writing to the DRAM is non-blocking on all cores since there are no dependencies. Each core writes back to the DRAM in bursts of 32 or 128 bytes, depending on the encoded image, every time a row of pixels of an MCU block is processed.

E. Power Estimation

The proposed power model considers only dynamic power, which is proportional to frequency. Since cores 3/4 operate on twice the frequency of cores 1/2, we assume that they consume twice the power. We assume that the accelerators encumber cores 1/2 with additional power requirements. In its simplest form the power model states that for every unit of power cores 1/2 consume, cores 3/4 consume 1.6 units of power. The following formula is used for calculating power consumption, with t the execution time:

$$P_{dyn} = t_{core1} * 1 + t_{core2} * 1 + t_{core3} * 1.6 + t_{core4} * 1.6 \quad (1)$$

III. Different Versions of the JPEG decoder

A. Data Parallel Version

For implementing the data parallel JPEG decoder we used DMA communication and 4 cores. The image is split between the 4 cores using the modulo-split scheme. Figure 8 illustrates how the image is split under the modulo-split scheme together with some alternatives. The reasons for choosing modulo-split over the alternatives is to a) be compatible with the memory alignment and b) achieve a balance in workload intensity by minimizing the abrupt changes in the image encoded frequencies distributed over the cores.

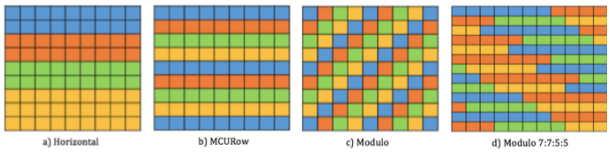

Fig. 8. Image splitting techniques

If we, naively, distribute the blocks to the available cores evenly, our implementation will most likely exhibit uneven execution times as suggested in Figure 2. In order to avoid this behavior we calculated the average execution time per image minus the initial header decoding time. We then derived an average ratio of the total execution times of cores 3/4 over the total execution times of cores 1/2 for the same image. The ratio is roughly 1.4 meaning that on average for every 1 block cores 3/4 process, cores 1/2 have processed 1.4. By implementing this approach the modulo-split modifies the splitting ratio as shown in Figure 8(d). Figure 2 shows that typically cores 3/4 take longer to execute, which means that there is some slack between the two execution times that we could use to our advantage. By distributing the workload using the workload ratio we decrease the execution time of cores 3/4 and increase that of cores 1/2. The goal is to achieve a balance between those two such that the total execution time can be reduced essentially for free.

The sequential part depicted in Figure 1 is a detrimental bottleneck and a source of huge waste of computations. It requires all cores to execute the Unpack function even in cases where the output of the Unpack function should not be used. The silver lining in the JPEG decoder is the presence of restart intervals which dissect the image in chunks with known sizes effectively removing the sequential bottleneck. Unfortunately, they are not essential for JPEG encoding thus images with restart intervals are rather scarce. However, when present,

we can exploit them and significantly mitigate the impact of the sequential bottleneck on the execution time. Our data parallel implementation takes advantage of the presence restart intervals and reduces the decoding time of images containing them by a significant margin.

B. Function Parallel Version

The JPEG decoder algorithm can be split in three fundamental functions as explained previously, effectively transformed into a streaming application. Our initial approach for realizing the JPEG function parallel version was to utilize 3 out of the 4 available cores with each of the 3 cores executing one of the fundamental JPEG functions.

However, the JPEG decoder algorithm dictates that CC executes only once per multiple Unpack and IDCT executions. Since CC is not executed as frequently as Unpack and IDCT, assigning only CC on an entire core will result in poor utilization results and inefficient power consumption. In addition to that we expect that the function parallel version will not match the performance of the data parallel version mainly because of the increased communication overhead between the cores which, depending on the image, can introduce significant blocking delays. For these reasons we implemented the 2-core function parallel version depicted in Figure 9 where core 3 executes Unpack and core 1 executes both IDCT and CC. The choices of the function mappings stem from Figure 5.

The remaining design choices in a function parallel version concern communication tuning. The C-Heap library, used in the function parallel and hybrid versions, facilitates a synchronization protocol for transferring data between cores by using FIFO buffers. It uses the notion of tokens to encapsulate the data to be communicated between the cores over the NoC. C-Heap allows the designer to freely choose the communication schedule, the size of tokens to be communicated between the cores as well as the size (in tokens) of the buffers and their placing. For an efficient and fast implementation, these choices need to be tailored to both the applications needs and the available resources of the platform.

Specifically, for the JPEG algorithm, the communication schedule must take into account that DMA burst transfers are more efficient when transferring multiples of 128 bytes. Unfortunately, even though Unpack produces a constant output of either 128 or 256 bytes an additional C-Heap and synchronization control bundle is required for communicating the tokens. The required size of the control bundle in our implementation is 48 bytes. Thus the size of the tokens communicated from core 3 to core 1 is set to be 304 bytes. The tricky part in C-Heap is deciding on the location and the size of the FIFO buffers. For our case deciding on the location is easy. The data that DMA handles need to be kept in the communication memory. In order to avoid redundant data transfers to and from the local memory to the communication memory, we place all buffers in the consumer side, core 1.

Figure 9 illustrates the mapping used in our 2-core function parallel version and depicts the best-case, worst-case and average execution times of processing a single token for each function on its mapped core. The reason IDCT contains only an average case is because the execution time of IDCT is constant and independent of the image. The delays in between

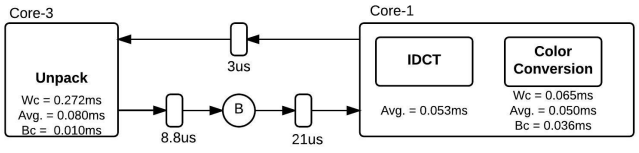

Fig. 9. Worst case and best case execution times

correspond to the fixed communication delay for moving one token.

Following the figure we can simulate a scenario that if the FIFO buffer between core 3 and core 1 is not sufficiently large to hold the data that core 3 sends while core 1 executes CC, the buffer will fill up and core 3 will block. The trivial solution to avoid this would be to increase the FIFO size to the maximum number of tokens that core 1 needs to execute CC, which for the JPEG decoder is 10. This however is a gross overestimation as Figure 10 suggests.

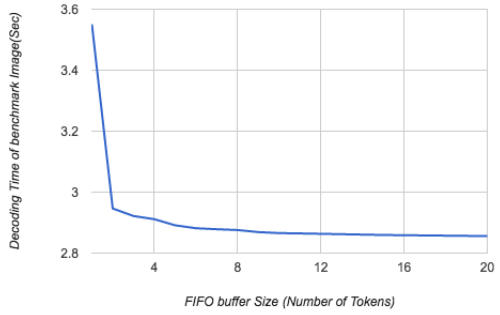

Fig. 10. FIFO Buffer size vs Total Execution time

The reduction of execution time as the buffer increases, comes at the heavy price of increased latency. In extreme cases where Unpack outperforms CC, tokens will be stored in the buffer for a longer period of time. In such cases, core 3 will finish processing its assigned data before core 1 has finished processing the tokens in his buffer, effectively remaining idle for a significant portion of the total execution time. Based on 10, a buffer size of 6 tokens is sufficiently large to prevent blocking on core 3, without adding substantial latency to the implementation.

C. Hybrid Version

Our hybrid version is an attempt to merge the two previous versions and its main objective is to filter their disadvantages. For the data parallel this disadvantage comes in the form of the excessive DRAM accesses and waste of computations by redundantly executing the sequential Unpack function on all cores. For function parallel it comes in the form of communication overhead and communication blocking. By merging the two versions we can effectively delay the appearance of these bottlenecks while simultaneously improving the resources utilization efficiency and delivering higher, or at least matching, performance. Following the profiling results and the assumptions outlined in the previous chapters we implemented two versions.

The first version tackles the issue of redundant DRAM accesses by using core 3 for executing Unpack and cores 1/2 for executing the both IDCT and CC in parallel. Cores 1/2 are assigned new data to process in a modulo fashion. With only one core executing the Unpack function, all data reads from the DRAM will be fully used and processed. Thus no DRAM accesses are wasted. On the other hand, the communication blocking problem persists. If core 3 stalls then cores 1/2 will eventually block.

The second version addresses the issue of stalling, by essentially running the same pipeline in parallel. Unpack is mapped on cores 3 and 4, and both IDCT and CC are mapped on cores 1 and 2. The image is split in a modulo fashion as depicted in Figure 8c. Cores are split into two pairs and communicate exclusively with each other. This approach decreases the chances of having multiple blocking cores due to stalling caused by one core. If one core stalls, only the core that is paired with it will block. However, in this case the problem of redundant DRAM transfers remains present, since half of the executions of Unpack on cores 3 and 4 will not be forwarded to cores 1 and 2 respectively.

In both versions the token and buffer sizes and buffer placing remain the same as in the function parallel version.

IV. Benchmarking

The purpose of benchmarking our various parallel JPEG decoder implementations is primarily to measure their efficiency in terms balancing the optimization goals, performance and resource utilization and evaluate our design choices. Initially, we put our design choices to the test by comparing them to their alternatives. If a design choice is refuted we re-evaluate our approach and present the reasons for this mismatch. A large portion of the design choices was already tested in chapter *II*. Since design choices guide the implementation steps it is better they are performed at an early stage. We then proceed by comparing each version to its alternatives and present a radar chart for a clearer illustration of the optimization trade-offs at hand. The baselines used in the benchmark suite is the single core DMA versions of JPEG and the image used in all experiments is a 1024x768 RGB image with 46 restart intervals.

A. Design choices

1) Reset Markers: Figure 11 illustrates the improvement achieved with the use of restart intervals for the 7:7:5:5 modulo version.

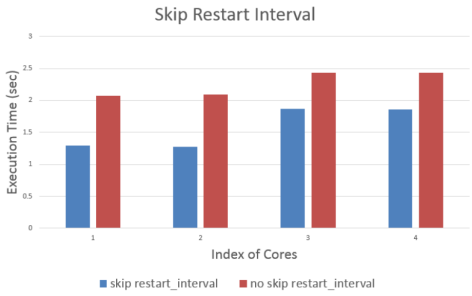

Fig. 11. Execution time with and without using restart intervals

Making use of the restart intervals reduces the execution time of cores 1/2 by 34% and cores 3/4 by 25%, effectively decreasing the total execution time by the 25%. It is no surprise that the execution time reduced significantly, since the decoding is done more efficiently by having each core decode only its designated parts of the image, rather than performing Unpack indiscriminately.

2) Power estimation for function parallel version: In chapter *III.B* we argued that the addition of a third core to our function parallel version would render it inefficient in terms of utilization and power consumption. Table II illustrates the expected power consumption of a 3-core function parallel version and the power consumption of our 2-core version, for the proposed power model.

Table II. Power consumption for the function parallel version with 2 and 3 cores

Mapping	Execution Time(us/Block)			Energy Consumption(units/block)			Total (units/block)	
	Core-3 (unpack)	Core-1	Core-2	Core-3	Core-1	Core-2	Normalised Execution time	Normalised Energy Usage
2-Core	0.08	0.103	-	0.128	0.103	-	0.103	0.2678
3-Core	0.08	0.053	0.05	0.128	0.053	0.05	0.08	0.288

In order to derive the above table we consider the average case execution times of a single token in Figure 9. The execution time of a pipeline stage is equal to the slowest stage. If we simulate our 2-core function parallel version then the maximum execution time of the pipeline for one token would be $0.103ms$ whereas for a 3-core version it would be $0.080ms$. Then from the power model (1) we can derive that $P_{2core} = 0.267$ power units and $P_{3core} = 0.288$ power units. It is apparent that from a single token of size 304 bytes the 3-core implementation requires a 7.8% more power units. This is obviously not efficient, thus our initial assumption that a 3-core function parallel implementation would be inefficient is verified.

B. JPEG parallel versions

1) Data parallel versions: Data parallel JPEG, although seemingly trivial to implement, contains multiple design choices, most importantly the image splitting techniques and the workload ratio for which is hard to reason in a sound manner because of the JPEGs unpredictability. Therefore, we test our proposed 7:7:5:5 modulo implementation against the other alternatives of Figure 8. For more accurate results the workload ratio is implemented on MCURow only, since horizontal cannot make use of it. 3:3:2:2 is the ratio derived when the header decoding execution time is included.

Using the radar chart of Figure 12 we can measure the impact of the design choices of image split technique and workload ratio on the proposed data parallel implementation and evaluate the balancing of the trade-offs each implementations achieves.

Without the workload ratio the modulo implementation delivers the worst execution time and utilization and second worst power consumption compared to the other alternatives. However, with the workload ratio implemented modulo delivers a 20% decrease in execution time, 2% reduction in power consumption and 0.8% increase in average core utilization for

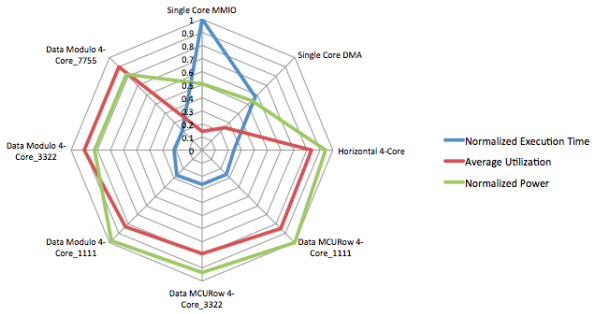

Fig. 12. Radar chart of the data parallel JPEG versions

both 3:3:2:2 and 7:7:5:5. On the other hand, MCURow does not respond equally well to the workload ratio. It delivers only 1.1% decrease in execution time, 6% reduction in power consumption and a 6% decrease in average core utilization. The results suggest that the workload ratio is not a uniformed approach that can be implemented on any given image-splitting technique. Perhaps the most interesting point we can keep from the graph is that the average utilization still remains relatively low, roughly 90%, which means that there is room for significant improvement, if another more elaborate scheme for balanced distribution is implemented.

Out of the available data parallel implementations presented above the 7:7:5:5 modulo achieves the best balance in trade-offs. It delivers the lowest execution time and highest average utilization and second lowest power consumption trailing only modulo 3:3:2:2 by a negligible 0.13%. Therefore, it is the version to be tested against the one function parallel and two hybrids.

2) Overall Comparison: Figure 13 compares the final JPEG versions in an attempt to reflect on their ability to achieve the optimization goals. Because the communication overhead, DRAM and FIFO buffers, is an important metric for comparing different implementations it will also be considered when evaluating the optimization goals achieved. If possible, an optimal point will be determined.

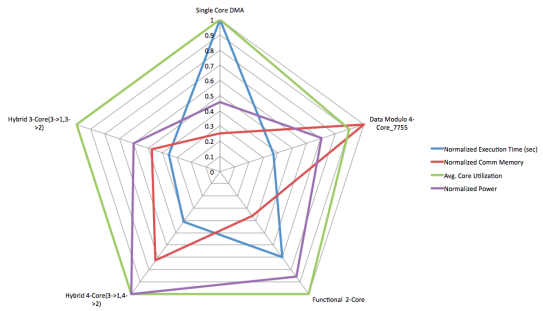

Fig. 13. Radar chart of the final parallel JPEG versions

The results validate our hypothesis that both hybrid JPEG version will show an improvement in the execution time and communication overhead compared to other versions. Our hypothesis that function parallel JPEG cannot match the performance of any of the data parallel versions of JPEG is also validated. However, the difference is surprising. An 88% increase in execution time when 63% less communication overhead is achieved hints that there is significant latency in the pipeline. This can most likely be attributed to the FIFO buffer size between core 3 and core 1. At this point we cannot be certain whether the size is small enough to cause core 3 to block or big enough to cause multiple tokens to stay in the FIFO buffers for too long. Additional checks need to be performed on the buffer sizes to pin-point the exact cause of this behavior.

Equally important is the fact that compared to the modulo 7:7:5:5 version the 3-core hybrid can achieve a 4.5% reduction in execution time, a 52% reduction in communication memory, a 15% reduction in power consumption and an 11% increase in average core utilization, effectively improving all optimization goals for free. The reason behind this dramatic improvement is the difference in the utilized cores. With only 3 cores on the hybrid it is expected that it will consume less power, all cores will be close to fully utilized and as expected more efficient in DRAM accesses which massively improves the execution time and reduces the communication overhead. The results are not as impressive when comparing the modulo with the 4-core since the addition of a fourth core increases the power consumption and the communication overhead due to the additional DRAM accesses performed by core 4. Even so, the decrease in execution time is noticeable, but offset by the increase in power consumption.

According to our implementation results the 3-core hybrid version is the optimal point, as it achieves the best balance of the optimization goals. Hence, porting the JPEG decoder on a CompSoC platform using the 3-core hybrid version will give a near optimal solution for the optimization goals considered.

Overall compared to the single core DMA baseline we have achieved a 64.4% decrease in execution time for a 36% increase in power and a rather significant 88% increase in communication overhead.

V. Conclusion

During the process of porting and optimizing the JPEG decoder on the CompSoC platform we witnessed first-hand, the interrelationships between software and hardware. In particular, we were able to experiment with the constraints that they pose to each other in the sense that the efficiency of one is dependent on the other. The platforms resources will only be utilized if the application has a use for them, such as the second DMAs in cores 3/4. Similarly, the performance of the application is limited by the platform's available resources such as the limited communication memory in cores 3/4 that causes an increase of the DRAM accesses that slows the execution time.

Balancing the dependencies between software and hardware is a cumbersome process but if executed accurately can yield good benefits. We have shown that by first analyzing the operating corners of the application and the platform and deriving design choices based on the analysis can lead to a near-optimal solution for both the utilization of the platform and the performance and efficiency of the application.

References

[1] Kees Goossens et al., *Virtual Execution Platforms for Mixed-Time-Criticality Systems: The CompSOC Architecture and Design Flow*, 2016.