Center for Embedded and Cyber-Physical Systems
University of California, Irvine

Mapping of an APNG Encoder to the Grid of Processing Cells Architecture

Vivek Govindasamy and Rainer Dömer

Technical Report CECS TR 22-02
September 29, 2022

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

vbgovind,doemer@.uci.edu
http://www.cecs.uci.edu

Mapping of an APNG Encoder to the Grid of Processing Cells Architecture

Vivek Govindasamy and Rainer Dömer

Technical Report CECS TR 22-02
September 29, 2022

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

vbgovind,doemer@.uci.edu
http://www.cecs.uci.edu

Abstract

Modern processors experience memory contention when the speed of their computational units exceeds the rate at which new data is available to be processed. This phenomenon is well known as the memory bottleneck and is a great challenge in computer engineering. In this report, a proposed computer architecture using local memory called "checkerboard architecture" is compared against existing shared memory architectures. Specifically, a well known multimedia application, Animated Portable Network Graphics (APNG) is used to benchmark the performance. APNG is a lossless image compression algorithm which we have parallelized as well as recoded to use small memory buffers. These optimizations enable implementation on the checkerboard architecture. The specification and modeling of the APNG encoder application and the checkerboard architecture are discussed. Next, the application is mapped to the new architecture and is compared against existing types of architectures such as existing shared memory processors to confirm the existence of the memory bottleneck and indicate possible solutions. Our experimental results show significant decrease in both execution time and memory contention in the checkerboard architecture compared to single core as well as shared memory processors.

Contents

List of Figures

List of Tables

Mapping of an APNG Encoder to the Grid of Processing Cells Architecture

Vivek Govindasamy and Rainer Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
vbgovind,doemer@.uci.edu
http://www.cecs.uci.edu

Abstract

Modern processors experience memory contention when the speed of their computational units exceeds the rate at which new data is available to be processed. This phenomenon is well known as the memory bottleneck and is a great challenge in computer engineering. In this report, a proposed computer architecture using local memory called "checkerboard architecture" is compared against existing shared memory architectures. Specifically, a well known multimedia application, Animated Portable Network Graphics (APNG) is used to benchmark the performance. APNG is a lossless image compression algorithm which we have parallelized as well as recoded to use small memory buffers. These optimizations enable implementation on the checkerboard architecture. The specification and modeling of the APNG encoder application and the checkerboard architecture are discussed. Next, the application is mapped to the new architecture and is compared against existing types of architectures such as existing shared memory processors to confirm the existence of the memory bottleneck and indicate possible solutions. Our experimental results show significant decrease in both execution time and memory contention in the checkerboard architecture compared to single core as well as shared memory processors.

1 Introduction

The increase in processor speeds over the past years has led to increased time spent in accessing the main memory to retrieve data to perform more computations for a specified time. As many cores try to access the same main memory it leads to contention and delays as each core waits longer to access the shared memory. Most modern CPUs used are usually shared memory processors. In this work, we explore if shared memory CPUs could benefit from using distributed local memory.

1.1 System Level Modeling

Given the growing complexity of embedded systems, it is necessary to model a design at higher abstraction levels to check whether it is suitable for implementation (Figure 1). To solve this issue, Electronic System Level (ESL) design and verification was introduced in the early 21st century [1]. An entire system can be modeled by using System-Level Design Languages (SLDLs), such as SpecC [2] and SystemC [3]. ESL design achieves the concurrent design of both the hardware and software components of an embedded system.

To perform ESL design, a simulator is required. An effective SLDL for this purpose is SystemC, which simulates concurrent processes using C++ syntax [5]. SystemC uses concepts inherited from the SpecC language. Using SystemC, a variety of models can be developed at varying levels of abstraction (Figure 2). The initial model is untimed and has a simulated time of zero, with only delta cycles passing. Delta cycles are used by the SystemC simulator to order events which are meant to run in parallel. The next step introduces timing delays in the modules. These delays can be used to measure the execution time of the model, and determine whether the design is an improvement

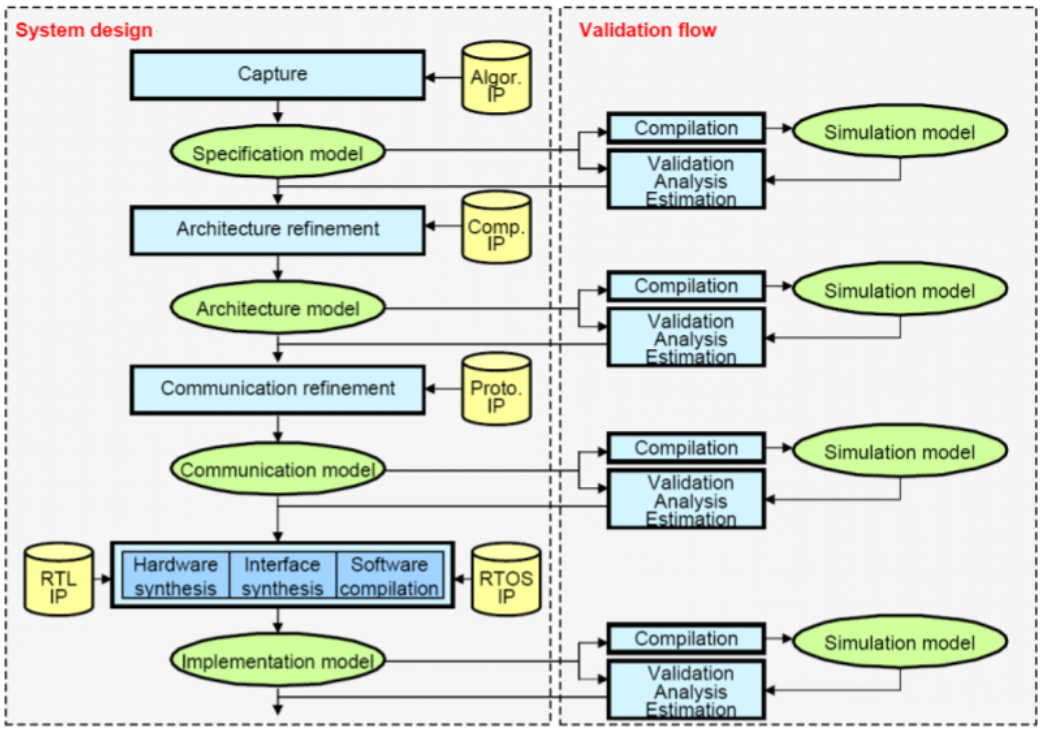

Figure 1: System on Chip design methodology [4].

over its predecessors.

Modules are the basic building blocks of a SystemC design. They are used to represent a component in a real system. SystemC uses Transaction-Level Modeling (TLM) which allows communication between modules using a method call [6]. The original TLM-1.0 used channels to perform communication between modules. The channels were modeled as FIFOs or buses. However, to obtain a more accurate model of a real world design, it is essential to also include memories. For this purpose SystemC TLM-2.0 was introduced where communication between modules takes place through explicit memories. In this work, we are using TLM-2.0 features to create an abstract model of the proposed checkerboard architecture [7] and map an application to it.

1.2 Problem Definition

A new type of computer architecture has been proposed, called the ”checkerboard architecture” by Professor Dömer at the University of California, Irvine [7]. To test the effectiveness of this new architecture, a multimedia application is run on the checkerboard architecture and compared with currently used architectures such as the shared memory architecture and single core architecture [8]. The multimedia application and the different architectures are modeled entirely using SystemC TLM-2.0. Modeling of the architecture was performed by a team of the CECS group at UCI [9] including the author of this report. This work contains three parts, (1) the application, (2) the architecture and (3) the mapping. First a suitable application is chosen which can be modified to run effectively on both a shared memory processor and a distributed memory processor.

The application used here is an Animated Portable Network Graphics (APNG) encoder [10]. APNG encoders are basically PNG encoders which concatenate the generated PNG images with additional information chunks which contain parameters needed by video players such as the frame rate. The increase in the number of compressed images lets the simulation run for longer and provide improved results. PNG images [11] are ubiquitously used to store screenshots and other pixel-based images [12]. However, the in-

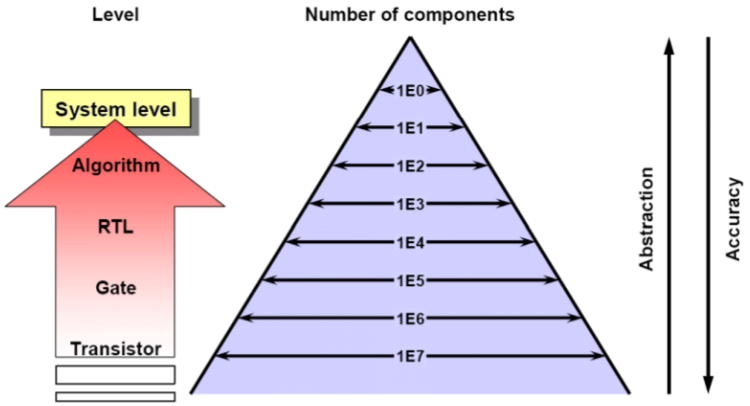

Figure 2: The increase in abstraction causes a reduction in accuracy at the system level [4].

terior working is not as well known as other image compression formats like JPEG [13]. PNG encoders have two main working components which perform the actual compression, the filters and the DEFLATE algorithm [14]. Filters are of five types, and they are mainly used to reduce pixel values so that they occupy less space. The reduced pixel values require less number of bits to transmit, providing some compression. Filtering also creates runs of data, and they are compressed in a similar way as run length encoding. The filtered values are sent in as inputs to the DEFLATE algorithm, which uses a combination of Lempel–Ziv–Storer–Szymanski (LZSS) and Huffman encoding to perform lossless compression. DEFLATE works better on values which are highly correlated to each other, which will be discussed later in this work.

For the architecture a "checkerboard" style of many cores with local memory is used (Figure 3). The name comes from the checkerboard pattern where the cores and memory are placed one after another, and each core has access to its own and three other neighbour memories. We model the checkerboard architecture using SystemC TLM-2.0. It contains several variations of a main logical component which is termed as a cell. Each cell is designed with the ideology that it represents a core and components that are private to that particular core. Therefore, the cell consists of a processing core which performs computation, a memory for the core that also facilitates communication between the adjacent cores, a core multiplexer which translates addresses seen by individual cores to global addresses, a memory multiplexer which forwards data transfer requests coming from various cores to the intended memory and a signal which the core uses to notify surrounding cores that it has modified data at a specific memory location. On the edges of the checkerboard, four off-chip memories are connected to the cores which do not have a memory on the edge of the checkerboard. These off-chip memories are much larger than the local memories attached to each core. The architecture can also be referred to as a grid of processing cells (GPC). The GPC contains the checkerboard as well as four surrounding off-chip memory units with multiplexers attached to them. The intention of this design is to accommodate the situation where multiple cores try to access the off-chip memory at the same time. In a test bench, the off-chip memories are connected to the stimulus and monitor modules which send and receive the input and output data to the checkerboard.

The main objective in this report is the mapping of the application to the various cells on the checkerboard. This step involves modifying the APNG encoder and rewriting the source code so that it is suitable for the architecture. The main step of this modification is to split computationally intensive parts of the encoder into smaller units and utilize communication between the cells to accommodate the splitting of data. SystemC delays are introduced at the multiplexers to time the performance and compare it to other possible mappings. Upon mapping the application to the architecture, experimental results show highly de-

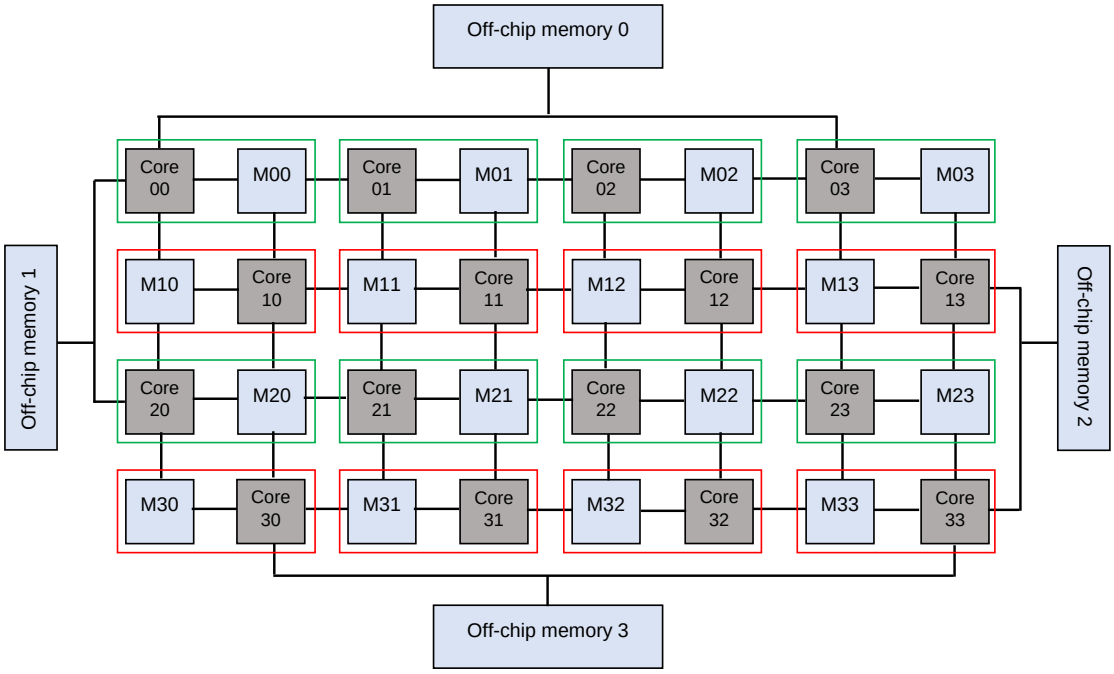

Figure 3: The simplified checkerboard model within the Grid of Processing Cells (GPC) [7]. The red and green outlines denote cells.

creased execution time compared to the shared memory model with negligible amounts of memory contention.

1.3 Related Work

Computer architecture is a vast field which started with two types of possible architecture, the von Neumann architecture [15] and Harvard architecture [16]. Von Neumann architectures share both data and instruction memory, whereas for Harvard architecture the storage and pathways are separate for data and instructions. The von Neumann architectures are simpler but suffer from the issue of contention between data and instructions. This is known as the von Neumann bottleneck [17]. Since then the most commonly used architecture has been the modified Harvard architecture for most general purpose computers [18], in which there exist seperate caches for both instruction and data but the address space is shared. Over the past many years the general trend is that processors become faster as designers increase the clock rate but the increase in access speed of memory is almost always slower [19]. This is known as the memory wall. To deal with this issue there has been more focus on implementing and improving caches [20]. However even the most advanced caches suffer from high miss rates if the cache is too small due to capacity misses or from higher memory access times if associativity is increased too much [21]. Caches also must implement cache coherence protocols so that the same data is shared between cores in the case of multicore processors. Caches have given rise to the Non-Uniform Memory Access (NUMA) where each core can access its own memory faster than the shared memory. An issue with NUMA is that the time to maintain cache coherency is quite high [22] and leads to contention as the interconnect used is common to every core. In this work we evaluate the proposed checkerboard architecture, which aims to create multiple pathways between cores thereby reducing contention significantly by changing the architecture of the processor itself and is novel in the area of methods to mitigate the effect of memory contention.

2 The APNG Encoder Application

This section [23] discusses the process of creating an APNG file. The process starts with compressing the input images into the PNG format and then using the output from DEFLATE to create an additional APNG file. A PNG image is comprised of multiple information and compressed data packets called chunks [24]. Chunks carrying information are not compressed, whereas every chunk carrying data has the data part compressed using the DEFLATE algorithm. To effectively compress data, the data must first be filtered using one of five different filter methods, which is discussed in Section 2.2.

Our SystemC model is structured as a parallel filtering architecture, followed by a comparator to choose the optimal filtered result. Filters operate row-wise and each row is sent to DEFLATE serially. SystemC modules then create chunks to be written to the final PNG file. The pixels are filtered and then restructured as a 1 dimensional array. The change in color is due to filtering and the subsequent reduction in pixel intensities. The first element of the array represents the filtering method used. Every row is joined as one large array and compressed to obtain the PNG data. Chunks are then created to provide information to the decoder.

2.1 PNG Chunks

PNG chunks carry information about the image or the pixel data. Chunks are also the only information that the PNG decoders are capable of reading. PNG decoders have been developed with the intention of ignoring any unknown chunk types so that they can maintain compatibility with newer encoders. APNG uses many chunks which are not recognizable to PNG-only decoders, but they still use the PNG critical chunks such that when a APNG file is opened on a PNG-only decoder it recognizes only the first frame and ignores the remaining frames.

The basic format of chunks is shown in Figure 4. This format is universal across every chunk and it consists of chunk type, chunk data and Cyclic Redundancy Check (CRC) [25]. The length parameter denotes the length of the data component of the chunk (Figure 4), but is not actually a part of the chunk. This can be confusing as the length parameter ignores the extra bytes used by the chunk type, and CRC. If a chunk of size 8 is encountered by the decoder, it means that it carries 8 data bytes. However the true length of the chunk is 16 data bytes. Extra zeros are added to the front of the length, chunk type and CRC in case the value is too small, so that the size of 4 bytes is always maintained.

The chunk type parameter denotes the type of chunk and its purpose. Some chunks are mainly used to transfer pixel values, whereas others are used for image attributes such as the resolution. Chunk type is always denoted by four alphanumeric characters. The last part of the PNG chunk is the CRC. This value is generated once the chunk type and chunk data have been set. CRC is computed using the Adler-32 checksum [26]. The data used for the computation of the checksum starts at the first letter of the chunk type and extends until the last byte of the chunk data. A potential error in CRC computation is the inclusion of the length parameter in front of the chunk. However, the PNG standard does not take the length into CRC computation. Hence, this error should be avoided as the extra four bytes representing the length could mark the chunk corrupt at the decoder side.

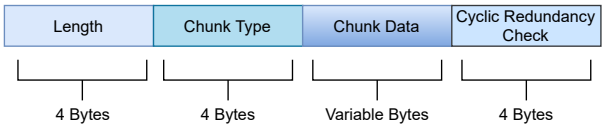

Figure 4: The universal format for every PNG and APNG chunk

At the very start of a PNG or APNG file, the 8 byte PNG signature is written. This universal value is always **89 50 4E 47 0D 0A 1A 0A**. When viewed in ASCII 50 4E 47 translates to PNG for easy identification.

The following subsections discuss the various PNG and APNG chunks required by the encoder.

2.1.1 Chunks used by PNG

PNG image data is stored as chunks, such as Image Header (IHDR), Image Data (IDAT), Frame Data(fDAT), Animation Controller(acTLl), Frame

Controller(fcTL), Image Palette (PLTE), Image Time (tIME), Image End (IEND), etc. Here we focus on IHDR, IDAT and IEND for the PNG encoder as these are critical chunks. The other chunks are known as ancillary chunks and are only required in specific situations. fDAT, acTL and fcTL are used for the APNG encoder discussed in the next section. [27] provides more information on encoders, and multiple encoder implementations in a variety of programming languages. The SystemC model developed uses a different source code from LibPNG as it must be converted to use small memory buffers and compatible with the new architecture which is being tested.

The Image Header Chunk (IHDR)

The IHDR is the chunk which contains important specifications about the image. It is exactly 13 bytes long. To convey its length, the '0D' hex value is always present before IHDR. Table 1 shows the specifications of IHDR chunk.

Table 1: Table for the PNG Header Chunk

Specification	Number of Bytes
Chunk name (IHDR)	4 bytes
Width	4 bytes
Height	4 bytes
Bit depth	1 byte
Colour type	1 byte
Compression method	1 byte
Filter method	1 byte
Interlace method	1 byte
IHDR CRC	4 bytes

The bit depth (Table 1) is the number of bits per pixel color. Our SystemC encoder uses 8 bits per color, and hence 24 bits per pixel are used for RGB. Color type indicates the color scheme. Truecolor is used in the SystemC model. Compression method is always 0, which means that the data is compressed using DEFLATE. Filter method is also always 0, which implies that the standard five filters are applied. Interlace is useful when it is required to render some rows or columns before the others, but it isn't required in the SystemC model and hence interlacing is not used and it is set to 0. A CRC check is present at the end with a length of 4 bytes.

The Image Data Chunk (IDAT)

The IDAT chunk contains the image data (Table 2). Similar to other chunks, before the chunk type (IDAT) is specified, its length is specified as a hexadecimal value. The only difference is that this value varies with parameters such as compression level, while it is fixed for other chunks. It is possible to identify the size of the IDAT chunk only after the pixel data is compressed using DEFLATE. The IDAT chunk contains the compressed data, held within a zlib wrapper. PNG supports only zlib for now. The first two bytes of IDAT's chunk data are always 78, which specifies the zlib wrapper.

Table 2: Table for the PNG IDAT chunk

Specification	Number of Bytes
Chunk name (IDAT)	4 bytes
Pixel data compressed using zlib	Variable bytes
IDAT CRC	4 bytes

A property of the IDAT chunk is that it can be broken down into many smaller chunks. These smaller IDAT chunks continue to use the same IDAT name, but carry different pixel information. They also need to have their CRCs recomputed for every individual IDAT chunk as a result. It is important to note that the multiple smaller IDAT chunks are still part of a single zlib stream. To ensure correct decoding, they must be written in correct order to the PNG file. This comes with the additional complexity of computing the zlib stream and breaking the compressed data into smaller IDAT chunks. However, there is no known advantage of splitting the IDAT chunk. Hence, the proposed APNG encoder uses only one large IDAT chunk, and this leads to a simpler design. Further, small memory buffers are used in the checkerboard model. Using small buffers adds a small increase in compression size as DEFLATE now works as a sliding window, rather than compressing the entire input data at once.

The Image End Chunk (IEND)
IEND is the final chunk in any PNG image. It always has the values- **49 45 4E 44 AE 42 60 82**. The first four values translate to the numerical characters IEND when converted to ASCII. The data field is empty. The last four values are the CRC values.

2.1.2 PNG Chunk Order

PNG chunks must follow the specific ordering of IHDR, IDAT followed by IEND. It is essential that this ordering is followed as normal PNG does not support out of order chunks. Figure 5 illustrates the chunk ordering of the SystemC PNG encoder.

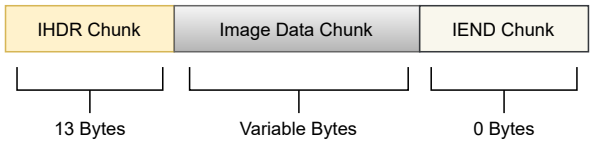

Figure 5: The PNG chunk order. The actual byte size of a chunk is the displayed value and an additional 8 bytes for the chunk name and CRC.

2.1.3 Chunks used by APNG

APNG chunks are ancillary chunks only readable by APNG decoders [10]. PNG decoders are able to recognize only the first frame as it is stored as an IDAT chunk, and they ignore the other chunks.

The Animation Controller Chunk (acTL)
The animation controller chunk is written immediately after the IHDR chunk to prepare the decoder to allocate enough memory for the specified number of frames. Table 3 shows the chunk specifications. APNG files can store a lot of frames (255 here), but

Table 3: Table for the APNG acTL chunk

Specification	Number of Bytes
Chunk name (acTL)	4 bytes
Number of frames	4 bytes
Number of times to loop the frames	4 bytes
acTL CRC	4 bytes

it is ideal to keep the frame count low as the size of the file can grow fast if the individual PNG frames are large. To infinitely loop frames a value of 0 is written to the four bytes before the CRC.

The Frame Control Chunk (fcTL)
The fcTL chunk provides information on the next frame to be rendered to the output buffer (Table 4). One fcTL chunk is present for every frame in the APNG file. The fcTL chunk starts with its sequence number in the APNG file. The meaning of this will be explained in the next section. Width and height can be changed for every individual frame. The X and Y offset denote the location of the first (top left) pixel to be rendered and subsequently the image itself. These two parameters are useful when the currently rendered frame is of a lower resolution than its predecessors. The delay numerator and denominator denote how long the frame is rendered on the screen. If the target is 30 frames per second, the delay numerator can be set to 1 and the denominator to 30. Frame disposal denotes what must be done to a frame once it has finished rendering. A value of 0 means that the frame is not disposed off and is left as is until it is overwritten, 1 means that the frame buffer is set to black before rendering the next frame, and a value of 2 means that the frame should be disposed of completely. The final frame blending byte can take the value of either 0 or 1, with 0 for no overwriting of pixel data of the next frame in the output buffer, and 1 for blending of the R,G and B values between the current and next APNG frame.

The Frame Data Chunk (fdAT)
The fdAT chunk is very similar to the IDAT chunk, except they differ by 4 bytes, which is the sequence number (Table 5). Every frame other than the very first frame must use the fdAT chunk to store its pixel data. At the start of the fdAT chunk the sequence number is specified. The sequence number is a very important parameter as it specifies the order in which the fcTL and fdAT chunks must be read.

APNG decoders support out of order presence of fcTL and fdAT chunks so that if a fcTL chunk comes with sequence number 8 at the end of the file, it will still be used before a fcTL chunk with sequence number 12 at the start of the file. Due to this feature, fdAT chunks can be broken down similarly to IDAT chunks

Table 4: Table for the APNG fcTL chunk

Specification	Number of Bytes
Chunk name (fcTL)	4 bytes
Sequence number	4 bytes
Width	4 bytes
Height	4 byte
X offset	4 byte
Y offset	4 byte
Delay numerator	2 byte
Delay denominator	2 byte
Frame disposal	1 byte
Frame blending	1 byte
fcTL CRC	4 bytes

Table 5: Table for the APNG fdAT chunk

Specification	Number of Bytes
Chunk name (fdAT)	4 bytes
Sequence Number	4 bytes
Pixel data compressed using zlib	Variable bytes
fdAT CRC	4 bytes

and may appear out of order, as long as they have a sequence number before their pixel data starts. To correctly order APNG chunks, the very first fcTL chunk would have the sequence number of 0. The following chunk would be the IDAT chunk which does not have a sequence number, but is automatically considered as the very first frame. The next fcTL chunk would have a sequence number of 1, and its fdAT chunk has a sequence number of 2 if it is not broken down into multiple chunks. The third fcTL chunk would have a sequence number of 3 and its fdAT is 4. The nthth fcTL chunk has a sequence number of n, and its fdAT chunks following it would have sequence numbers of n+1, n+2, .. and so on. It is essential that every fcTL and fdAT chunk has a unique sequence number associated with it.

2.1.4 APNG Chunk Order

In the case of APNG there is flexibility in the ordering of chunks. However, the default PNG chunks, IHDR, IDAT, and IEND must still remain in order. Two additional requirements are that the acTL chunk must follow the IHDR chunk and the IDAT chunk and its fcTL chunk must come before every other fdAT chunk (Figure 6). The remaining fcTL and fdAT chunks may come out of order. The SystemC model encodes the APNG file with ordered chunks, as small buffer sizes would not be suitable for holding the large amount of image data processed by the encoder.

2.2 PNG Filtering Algorithms

The technical details of filtering are described now. Filtering is essential to improve the compression level when the pixel data is compressed using the DEFLATE algorithm. The filters operate on every row individually [28]. Filters are of five different types, namely None, Sub, Up, Average and Paeth (Table 6). The filters find similarities between adjacent pixels. This helps predict the next pixel and it is a form of delta encoding, which provides a degree of compression. The most basic filter is None, which performs no operation. The Sub filter correlates with the left pixel value, the Up filter correlates with the previous row, the Avg filter correlates with both left and up pixel values and the Paeth filter correlates with the left, up and diagonal pixel values. The detailed algorithm for current pixel x is shown in Figure 7. The pixel on the left is denoted by a, the top pixel by b and the diagonal pixel by c. The Sub filter subtracts the current pixel value from the pixel on its left side (a), the Up filter subtracts the current pixel value from the previous row's pixel at the same index (b), and the Avg filter subtracts the current pixel value from the floor value of $(a+b)/2$.

The Paeth filter uses a function to calculate Paeth Predictor denoted by Pr to compute the Paeth filtered value. Figure 7 shows the method to compute the Paeth Predictor (Pr). Pr is subtracted from the current pixel value to compute the Paeth filter output. The Paeth filter is not as straightforward as the other filters. However experimental results show that it is often better than other filters when there is a high de-

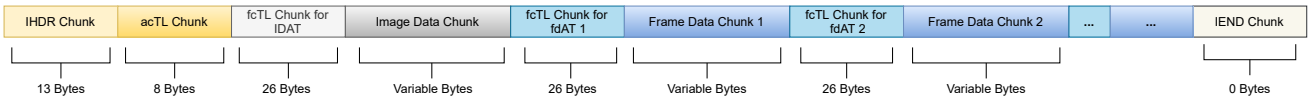

Figure 6: The APNG chunk order. The actual byte size of a chunk is the displayed value and an additional 8 bytes for the chunk name and CRC. The '...' represents the fcTL and fdAT chunks present for the remaining frames to be compressed in the APNG file.

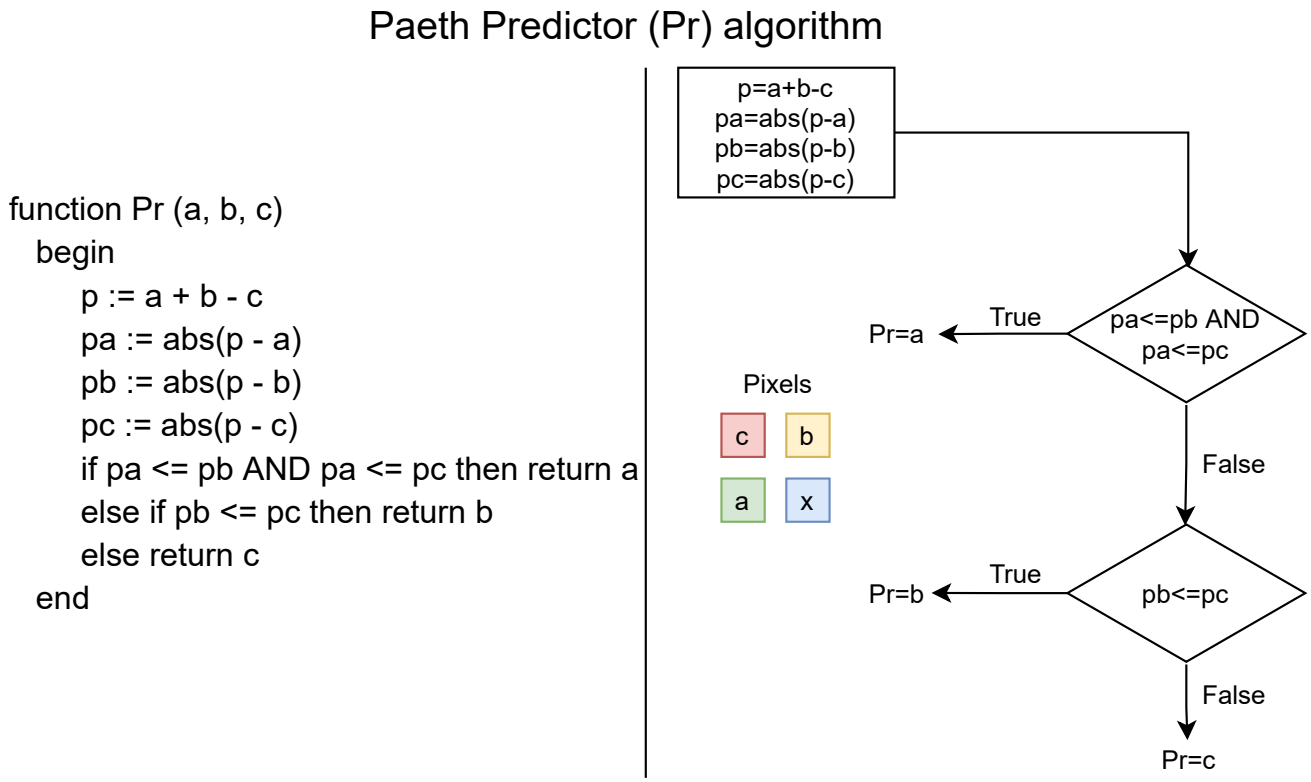

Figure 7: The steps to compute the Paeth Predictor (Pr), needed for the Paeth filter. The x pixel on the bottom right is to be predicted by the Paeth Predictor. The Paeth Predictor is then subtracted from the pixel value x.

Table 6: Table for PNG filter algorithms

Type	Filter Name	Filter Function	Reconstruction Filter
0	None	Filtered(x)=x	Reconstructed(x)=Filtered(x)
1	Sub	Filtered(x)=x-a	Reconstructed(x)=Filtered(x)+Reconstructed(a)
2	Up	Filtered(x)=x-b	Reconstructed(x)=Filtered(x)+Reconstructed(b)
3	Avg	Filtered(x)=x-floor((a+b)/2)	Reconstructed(x)= Filtered(x)+floor(Reconstructed(a)+Reconstructed(b))/2
4	Paeth	Filtered(x)=x-PaethPredictor(a,b,c)	Reconstructed(x)= Filtered(x)+PaethPredictor(a,b,c)

gree of variance among pixels in the current row. Up, Avg and Paeth filters require buffers to store the previous row in order to compute their output. If the current pixel which needs to be filtered is in the first row or column, then its neighbouring pixels a, b and c are assumed to be 0.

The five modeled filters operate in parallel. The filtered outputs are almost always stored as 8 bit characters (a byte), unless specified otherwise in ancillary chunks. If the filtered value is negative, 256 is added to it. Once the filtered values are computed, only one filtered output can be compressed by DEFLATE, as all of them represent the same row. The decision to choose the optimal filtered output is dependent on the encoder and the choice is not clear from the references. [28] recommends computing the absolute sum of the filtered row, and selecting the row with the least sum. [29] suggests using the sub filter for the first row and the Paeth filter for every other row.

While viewing the hexadecimal file of compressed PNG images, it was observed that images which were generated by computer tools used the first method of filter selection. These images used every filter available. However, images which were captured with a camera used the second method. A possible explanation for this is that even though camera captured images may have regions which seem to be of the same color (like an image of the sky), they are actually pixels of different intensities and the Paeth filter works best for them.

Our SystemC version of the encoder uses the first method of filter selection, and this process is done by a module called the Comparator.

2.3 The DEFLATE algorithm

DEFLATE is an algorithm originally developed in 1996 by Phil Katz [30]. Using a combination of LZSS [31] and Huffman Coding [32], it compresses data losslessly. It is widely used, and a common example is the ZIP file format in computers. While many different implementations of the DEFLATE algorithm exist (i.e. pkzip, zlib, gzip), it is important to note that PNG uses only zlib [33]. The implementations have different headers, and zlib, which starts with 78, is the only recognizable format by PNG decoders.

Brief descriptions are provided below for the two main coding algorithms used in DEFLATE. To implement DEFLATE these details are not required, but they make it easier to understand the DEFLATE function used in PNG.

2.3.1 LZSS Dictionary Coding

Dictionary coding is a lossless compression technique where matches are searched for in a data set. The data set can be predefined or generated. LZSS creates a new dictionary entry every time it encounters a new character or string. When the same string is repeated further in the array, it is substituted with the distance from the starting point and the length of the string [31]. For example, the string [I am Sam Sam I am], it is transformed to [I am Sam (5,3) (0,4)]. The strings 'I am' and 'Sam' have not been encountered before, so they are stored in the dictionary, but as they are repeated they can be directly substituted as numbers which saves characters.

DEFLATE will always store dictionary strings as hash tables. DEFLATE also uses a sliding window

of 2^{15}=32768 characters. This means that it will retain up to 32768 possible hash tables, and characters which are behind the current character by 32768 positions will be discarded. This is done so that DEFLATE does not need to search the entire dictionary for each string, which significantly saves compression time. After this value is exceeded, new Huffman codes are generated. This value can be changed to powers of 2, provided there is enough buffer space, and larger sliding window sizes increase compression level.

2.3.2 Huffman Coding

Huffman coding is a lossless prefix coding technique, which tries to encode frequently repeated characters with the least number of bits [32]. Prefix coding ensures that there exists no other code segment with the same initial segment as another code word. It employs a greedy strategy and generates a near optimal code, which is unique. Huffman coding generates binary characters only.

Here is a simple example to demonstrate Huffman coding. Consider an output from the LZSS encoder as (5,3) with a 60% chance of appearing, (0,4) with 30% chance and (0,1) with a 10% chance. We would encode (5,3) as 0, (0,4) as 10 and (0,1) as 11. No two code segments have the same prefix here. If an output stream is 0001110, the only possible decoded output is (5,3), (5,3), (5,3), (0,1), (0,4). The code segments and their literal values are transmitted along with the Huffman stream. DEFLATE has other technical details which can be found on the official page [34].

2.3.3 The CRC computation algorithm

Cyclic Redundancy Check (CRC) is required for every chunk in the PNG stream [35]. PNG universally uses only the Adler-32 checksum, as it is the default option in zlib. To compute the checksum two individual 16 bit checksums are computed and their results are concatenated into a 32 bit integer.

$A = (1 + D_1 + D_2 + ... + D_n)\ mod\ 65521$

$B = ((1 + D_1) + (1 + D_1 + D_2) + ... + (1 + D_1 + D_2 + ... + D_n))\ mod\ 65521$

$= (nD_1 + (n-1)D_2 + (n-2)D_3 + ... + D_n + n)\ mod\ 65521$

$Adler - 32(D) = B \times 65536 + A$

Here A and B denote the two 16 bit partial checksums. $D_1, D_2, ...D_n$ denote the decimal version of the characters or the array in ASCII format. For example, the character 'C' would be 67 when used as an input to the Adler-32 checksum. The checksum values must then be converted to hexadecimal. This results in an eight hexadecimal CRC value. The Adler-32 checksum will usually require the entire buffer of which the CRC needs to be computed. However, a slight modification can allow it to be computed in a rolling manner which works well with the SystemC version of small memory buffers.

2.4 APNG Modeling Details with Parallelism

Our initial model of the APNG encoder is modeled by creating separate modules for every computationally intensive unit (Figure 8). The modules are the Color Splitter, the Subtract filter, the Up filter, the Average filter, the Paeth filter, the Comparator, the Compressor and the APNG Encoder. Three identical modules of the filters are instantiated for the three color channels sent by the color splitter. In total twelve filters are instantiated. These modules are rearranged when they are mapped onto the checkerboard model, while their functionality remains the same.

Note that our modeling aims at operating most operations in parallel. This enables parallel execution on architectures with parallel computation units and thus results in shorter execution time. In other words we are modeling a parallel implementation of an APNG encoder. The APNG encoder communicates using two data types- Row and Interlaced_Row (Table 7). The Row data type holds the red, green or blue color intensities of a row. The size of its data parameter is approximately the width of the image. The Interlaced_Row data type holds a much larger data parameter which is three times the length of the image plus one. Both these data types contain additional parameters. The Row data types also carries the filter type signifying which filtering scheme is used on it. The Interlaced_Row carries the length of relevant compressed data in its data parameter. It also carries the CRC of the IDAT and fdAT chunks.

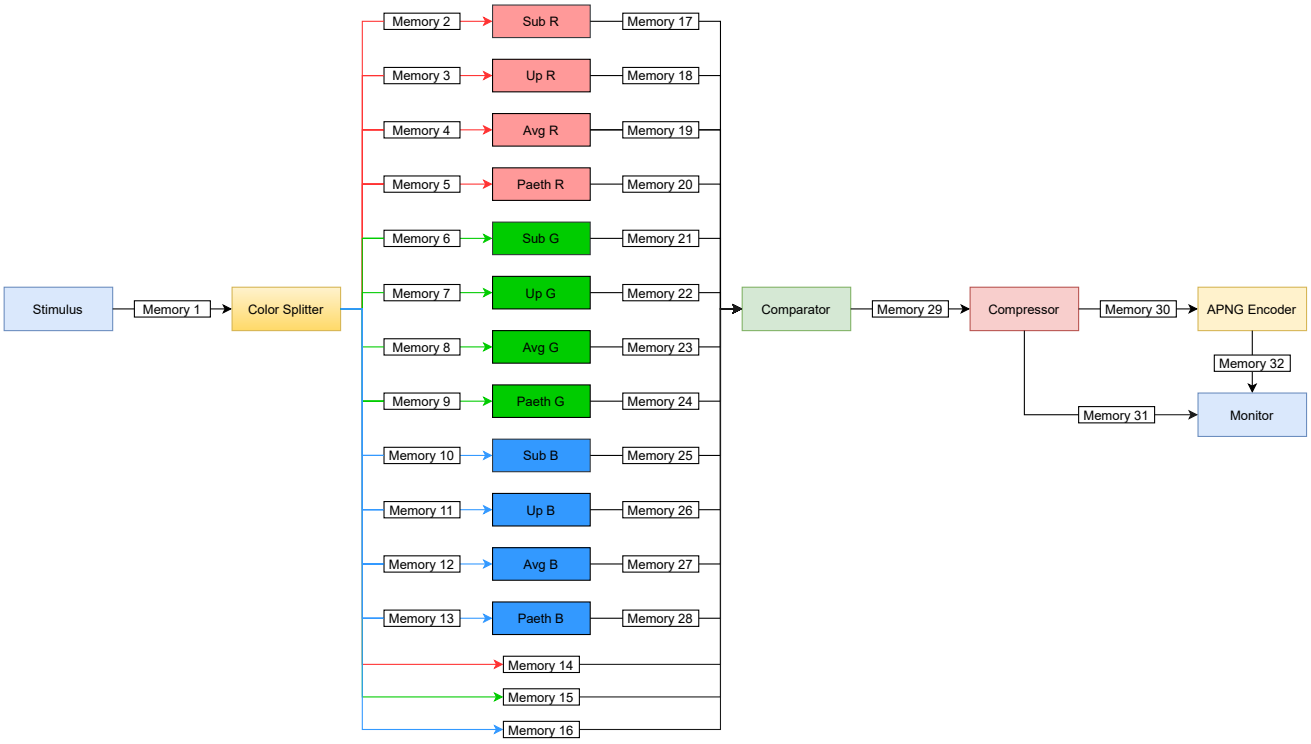

Figure 8: Our parallel APNG encoder as a TLM-2.0 model with memories between modules.

Small memories are used to communicate between adjacent modules. Row and Interlaced_Row are the only two data types which are communicated using the SystemC TLM-2.0 b_transport function. They can be thought of as the packets used to transfer pixel data between modules. Only the portable pixel map (.ppm) used by Linux computers is readable by the APNG encoder. Other formats can be converted to .ppm using ImageMagick or other image processing software [36]. This limitation exists because reading multiple image formats would require more complex decoders.

2.4.1 Stimulus Module

The APNG model takes input from a stimulus module which reads the .ppm image using basic C++ file I/O and sends it to the color splitter module using the Interlaced_Row data type via a memory. The .ppm image format contains the interlaced RGB data after a fixed offset which is calculated using the height and width parameters set in the preprocessor macros section. Each Interlaced_Row holds exactly one row of the input .ppm image, which is sent to the Color Splitter through a memory connected using sockets.

2.4.2 Color Splitter Module

The Color Splitter is the module which splits the interlaced RGB color stream that comes from the stimulus module into the individual color components and sends them to the individual filters. It rearranges the interlaced pixel data sent by the Stimulus into individual R, G, and B channels stored in the Row data type. The individual Rows carrying the R, G, and B pixel data are then sent to the four filters and additionally sent as unfiltered values to the Comparator module. The color splitter has fifteen output sockets of which twelve are connected to the filters through the memory units. The remaining three red, green and blue outputs are directly connected to the Comparator module.

Table 7: Table for APNG data structures

Data Structure	Approximate Size	Important Parameters	Used by Modules
Row	Image Width	Pixel Data, Filter Type	Stimulus, Color Splitter, Compressor, APNG Encoder, Monitor
Interlaced_Row	3*Image Width	Pixel Data, length, IDAT & fdAT CRC	Color Splitter, Filters, Comparator

2.4.3 Filtering Modules

The five filters operate completely in parallel. Each filter performs the computation only on a single row at any given time. Filtered values are computed using the mathematical equations which describe them, by using multiple for-loops to calculate the value of each individual filtered pixel. Filtering is performed on each row containing the pixel values. For the Up, Avg and Paeth filters the computed result is stored in a row buffer for the next row which will require the previous pixel values. Once these values are computed, the filter makes a computation of the absolute sum of pixel values. This value is stored in the Row data structure as the Row_Sum value for the row of a specific color. The row is then sent to the Comparator module through a memory.

2.4.4 Comparator Module

The Comparator receives the Row data types from all of the filters and the Color Splitter. First it computes the Row_Sum of the unfiltered R,G and B values. Next, it performs a comparison operation, and using multiple "if else statements" chooses the filtered row with the smallest Row_Sum to send to the Compressor. The selected row is then sent to the Compressor, after setting a Filter_Type parameter in the Row data type. The Filter_Type takes a value from 0 to 4. A value of 0 refers to Unfiltered, 1 to Subtract filter, 2 to Up filter, 3 to Average filter and lastly 4 to Paeth filter. This is performed individually for the three colors. The rows are then sent to the Compressor using another small memory in between the two modules to perform the transactions.

2.4.5 Compressor Module

Using the three Rows from the Comparator, the Compressor creates an Interlaced_Row which contains the Filter_Type at the start, followed by the pixels of the three rows, interlaced such that the pixels appear as R,G,B triplets. This entire Interlaced_Row is then compressed using zlib library functions.

The procedure for compression works as follows. First a z_stream data type is created. This is the basic compression data type used by zlib which stores the Huffman characters. Next, two arrays 'in' and 'out' are created. The 'in' array takes the next array of values to be compressed. The output buffer is 'out' which contains the compressed values. It is important to note that the output buffer may often have nothing in it, as the z_stream data type may wait to output the values at once. To check what length of the 'out' buffer has usable data, another variable 'have' is initialized.

The compression work is performed using a call to the 'ret=deflate(strm, flush)' function. The 'flush' parameter indicates whether all of the pending data stored in the z_stream data structure should be written to the output buffer when the deflate function is called. Ideally, this value should be Z_NO_FLUSH except in the case of the very last row, where it should be Z_FINISH. Other options such as Z_SYNC_FLUSH exist, but these are not required in the case of PNG. Using two "do while loops", the 'in' buffer is continuously updated with the next Interlaced_Row coming from the Comparator module. The outer loop uses the condition that while the 'flush' parameter is not Z_FINISH, compression is continued. The inner loop checks whether the output buffer is full, in which case the data is simultaneously written to the PNG file and also sent to the APNG encoder module to create the APNG image. Compression stops when Z_FINISH is set as the value for the 'flush parameter', which is determined, by using a counter, whether it is the last row. The amount of time that the 'out' buffer has useful data in it is completely variable and changes between images. To ensure that

the APNG encoder receives the correct number of output rows, a variable called Last_Row is initialized in the Interlaced_Row data type. When the two "while loops" finish, the value of Last_Row is set to 1, so that the APNG encoder knows that the entire data has been received. This implementation of DEFLATE for APNG encoding in SystemC is a highly modified version of the zpipe.c example by Mark Adler, an author of the zlib library [37].

The IDAT CRC and fdAT CRC are computed while the two "do while loops" are running. Initially, the CRCs are set to 0. Every time new compressed data is created, the CRC is updated using the data in the 'out' buffer as well as the amount of valid data using 'have'. The continuous update of the CRC is possible by creating a CRC table, computing and storing newly encountered values in it. This avoids creating a large array to store the entire IDAT/fdAT chunk (which invalidates the idea of small memory buffers) and then find the CRC value. The fdAT CRC is sent to the APNG Encoder using a parameter 'fdAT_CRC'. CRC computation is performed using a modified version of the function which uses a large buffer provided by the PNG developers [35].

When the Compressor thread starts, it first creates and sends the IHDR chunk to the PNG thread of the monitor module. The IHDR chunk requires only the resolution of the image. It then performs the above listed compression procedure to generate the IDAT chunk, which is sent to the PNG thread of the monitor module. This process repeats over a variable number of times, and the Compressor also sends the 'have' variable to signify the length of the compressed data generated by the deflate function. The IDAT chunk is sent through the Interlaced_Row data type. Lastly, the threads write out the IEND chunk when deflate has finished producing output. These steps are repeated for every .ppm image such that for every .ppm image a compressed PNG file is produced from the Compressor module.

2.4.6 APNG Encoder Module

This module produces the APNG image chunks. It creates the various chunks required for APNG such as the IHDR, acTL, and fcTL for IDAT at the start. These chunks require the resolution of the image, number of frames, and frame rate which are set in the pre-processor macros section of the source code. The chunks are passed to the APNG thread of the monitor module which performs the writing of these chunks on the APNG file. Then, using the compressed data from the Compressor module, APNG Encoder creates the IDAT chunk for the very first frame it receives. From the next frame onward, fcTL and fdAT chunks are created and sent to the APNG thread of the monitor module. When the Compressor module has no more data to send, the APNG Encoder sends the final IEND chunk to the monitor module.

2.4.7 Monitor Module

The final module "Monitor Module" produces both PNG and the APNG images. It uses two separate threads running in parallel called PNG thread and APNG thread. This is performed so that both PNG images and the APNG image can be generated in parallel as well as to avoid any accidental writing of an APNG chunk to the PNG file and vice versa. The two threads first create appropriate names and then write the chunks to their respective files as they are sent by the Compressor and APNG Encoder module.

3 Grid of Processing Cells (GPC) Architecture

One instance of the Grid of Processing Cells (GPC) is the checkerboard architecture [7] (Figure 3) with off-chip memories connected to the checkerboard. The off-chip memories transfer data to the checkerboard for computation. They are also connected to the stimulus and monitor modules which function as the I/O devices.

3.1 Checkerboard Architecture

The proposed checkerboard architecture (Figure 9) is a new type of multiprocessor architecture which could be a suitable replacement for shared memory processors (SMP). The idea behind the architecture came from Professor Dömer [7] and modeling was carried out by the CECS group [9]. The checkerboard model is similar to a distributed memory computer with individual cores with local memory which can be accessed also by their neighbours. The main difference is that the memory size is small and the cores themselves only perform computation comparable to that of a desktop CPU. The small memories are referred to as on-chip memories, and are expected to be as fast as the cache in a multi-core computer.

An interesting question is whether smaller distributed memories can be a viable alternative to one large shared memory. The answer depends on a variety of factors, such as the application itself and its ability to be re-coded to support an efficient multiprocessor implementation, delays caused by the multiplexers namely the memory multiplexer and the core multiplexer, the contention in memory access between a small memory and its four neighbours, etc. A detailed presentation and analysis of timing results is presented in the next section.

The off-chip memories facilitate interaction of the checkerboard architecture with the outside world, and they can be considered as the method of sending Input/Output (I/O) to it. For our 4×4 checkerboard the size of each off-chip memory is up to 0x20000000 bytes. Since a single off-chip memory contains multiple ports, a memory multiplexer is required to choose the correct signal coming from one of the cores. The stimulus and the monitor are connected to the other side of the off chip memory. Only these two modules perform I/O operations. The checkerboard is restricted to performing only computations.

The interior part of the GPC contains the 4×4 checkerboard model (Figure 9). It has sixteen individual cores and memory units, along with their respective multiplexers, the core multiplexer and the memory multiplexer. The term cell is used to describe, the four modules joined together (Figure 10). The checkerboard can be scaled to any number of even cells, as long as the user can instantiate the required modules. No change of the source code pertaining to the cells is required. This work has been described in greater detail in the M.S. thesis by Yutong Wang [38]. It was found that the GPC is scalable to an even number of cells such as 8×8 or 16×16 but odd numbers where excluded as they add further complexity to the cell types.

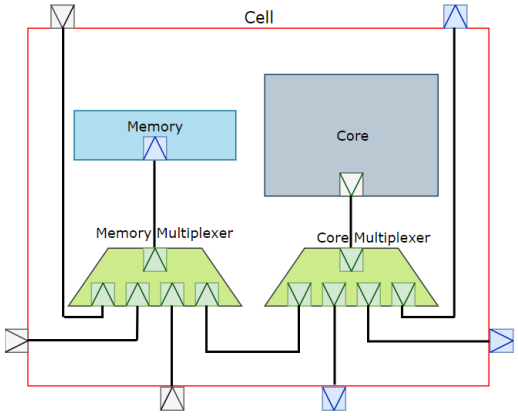

Figure 10: The interior components of a cell. Note that the number of sockets may vary on the multiplexers and on the exterior of the cell.

Our checkerboard is modeled using SystemC TLM-2.0 which provides the complete features of C++ and TLM-2.0. The model specifically uses TLM-2.0, rather than channels, because it is memory accurate. Also, the multiplexers can have delays and hence timing can be simulated.

3.2 The Cell Architecture

The description and modeling details of the core, memory, core multiplexer, and memory multiplexer

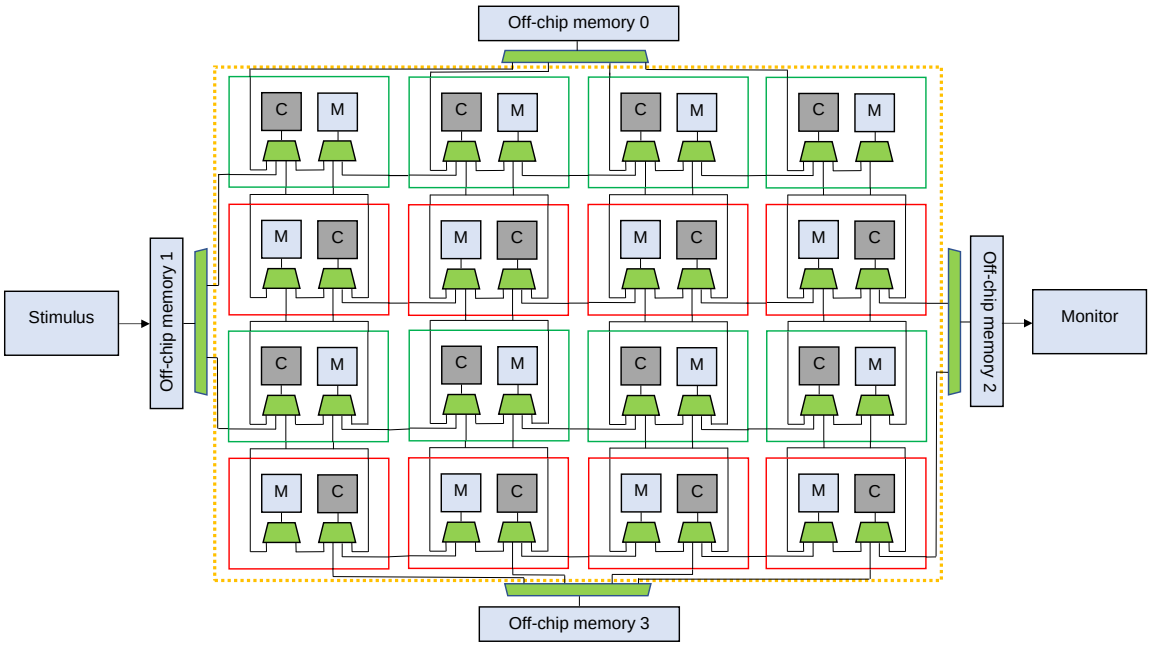

Figure 9: Detailed 4×4 checkerboard model [9]. The stimulus and monitor can be moved to different off-chip memory locations if necessary. Signals exist between every core and its neighbour, as well as cores adjacent to the stimulus and monitor.

modules are provided in this section.

3.2.1 Core Module

The core module is the primary computation component of the cell. It contains a single socket through which it is connected to the core multiplexer. It is a C++ class containing frequently used communication and general arithmetic functions. The primary communication functions are the SystemC blocking transport (b_transport), which is used by TLM-2.0, and synchronization functions to prevent possible race conditions when interacting with other cores. The most vital component of the core responsible for efficient communication to other cores is the SystemC event (sc_event) object which is a member of the SystemC core library. Each core has its own signal which is passed by reference to adjacent cores. A total of four events are available for use by the core. Whenever communication is necessary it calls by reference the neighbouring core's signal and transfers data safely without the possibility of any race conditions arising.

The communication scheme (Figure 11) uses counters instantiated in the common memory shared between the cores. When a core reads or writes, it signals the other core to wait until it completes its operation, and modifies the received/sent counter variable in the memory. It then proceeds to use a b_transport call to receive or send the data, and sets the sent/received counter that it has finished receiving/writing the data. A few other functions are also implemented. When instantiating specific cells present in the checkerboard the core class and its functionality are inherited. Instantiated cells are then provided additional cell specific functions. Each core contains a main thread which is used to perform the actual computation, and from where the communication functions can be called.

3.2.2 Core Modeling Details

The core module uses a single socket named CoreBus which is directly bound to the core multiplexer. The core has six variables which are passed to it as references, namely the four signals which are used

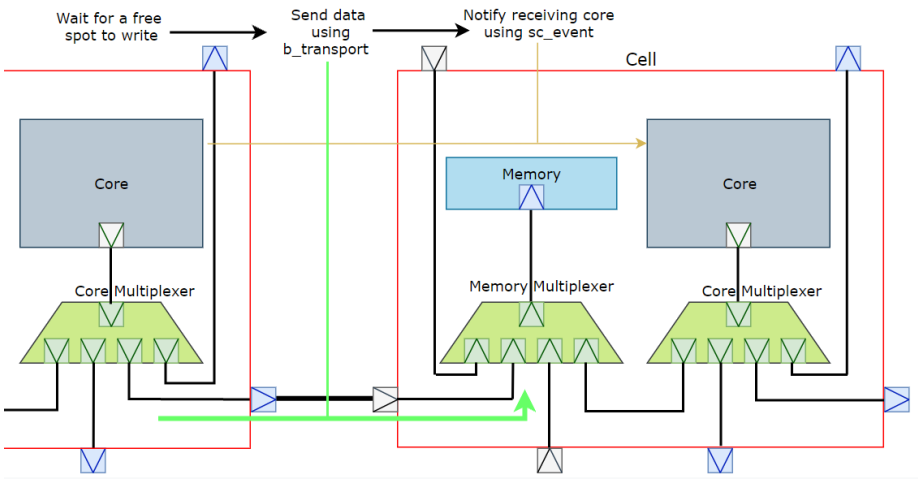

Figure 11: Signal based communication between adjacent cells.

to communicate to other cores and another two integer variables x and y which identify its position on the checkerboard. Among the communication functions, the primary function is memAccess() (Input parameters are not listed here unless it is simple). The memAccess() function serves as a wrapper to the real communication function b_transport. Before performing a b_transport call, several different variables must be set. These include whether to read or write the data, the location of where to store or send the data, where it must be read from or placed in memory, the size of the processed data and whether direct memory interface should be turned on or not. The memAccess() takes all of these parameters and conveniently combines them into a single function. Synchronized communication between cores is performed mainly through the two functions PushRow() and PopRow(). The use of these two functions is made possible through the presence of counters in the memory. Only the communication of the Row and Interlaced_Row data type is used throughout the SystemC model, and hence the functions are named as such. Two counters are used in the memory, the receive counter and sent counter. The receive counter is incremented when there is a free spot to write and the b_transport is performed. The working mechanism is similar to the bounded buffer problem.

PushRow() first checks the value of the received counter and then proceeds to send data, if there is a free spot in the buffer or waits in a while loop if there is no space. Whenever there is a change in the received counter value, it checks the loop condition again and it proceeds to either write or wait depending on the space available.

PopRow() functions similar to PushRow(), except it reads the data and checks the sent counter instead. Each of the counter increments are always followed by an sc_event statement to immediately notify the adjacent core the change in counter values. The sc_events are passed by references so that both cores share the event. When the checkerboard model is initialized all the counters between cores must be set to 0. For this, the checkerboard uses the SystemC core library command to wait one second, which is wait(SC_ZERO_TIME). This way every counter starts at 0 to prevent the possibility of any core writing or reading ahead of the others.

A few other functions are also present in the core module, such as LoadRow() and StoreRow(). These are mainly wrapper functions for receiving and sending of the Row data type. The last function Convert_To_Byte_Format() converts unsigned long numbers (such as the CRC) used by various modules of the APNG encoder into the correct byte format which is written onto the APNG file.

3.2.3 Memory Module

Each core has its own memory to store its data. The size of this memory is up to 0x08000000 bytes. Each

memory has only one port, to keep the architecture as real as possible to modern computers. The address space of each local memory module starts at 0x00000000. The memory can only have its contents modified by the four neighbouring cores, and it uses TLM-2.0 communication protocol.

3.2.4 Memory Modeling Details

The memory used throughout the SystemC model is known as the 'Memory1p' which stands for a one socket memory. Memory in actual hardware has only one socket and is connected to the bus or multiplexer so that the SystemC model resembles a real computer scenario. The constructor of the one socket memory accepts the size of the memory required so it can be chosen when instantiated both in the cell for the small local memory (on-chip memory) and in the top module for the off-chip memory. The on-chip memories in the cell have a size of up to 0x08000000 and the off-chip memories are of size up to 0x20000000. These values can be changed in the pre-processor macros section in case a smaller memory size is required.

3.2.5 Core Multiplexer Module

In order to communicate with the neighbours of each core, a core multiplexer is needed to translate addresses. The core multiplexer contains one socket to communicate to the core and four sockets to connect to the adjacent memory multiplexers. The issue of address translation arises from the default memory sizes. Since each address has a space from 0x00000000 to 0x08000000, the core would specifically need to choose which memory it wants to either read or write to. However, it would be a better choice if the core just chooses an address to write to and the core multiplexer then chooses the correct memory to write to. This would also be the more realistic solution as modern operating systems use page tables similar to the core multiplexer to translate the addresses from logical to physical. The core multiplexer forwards b_transport calls after performing the address translation. The global address space used by the checkerboard architecture refers to the four off chip memories on the outside and the local address space refers to the small memories near the cores.

3.2.6 Core Multiplexer Modeling Details

The core multiplexer is the module responsible for the conversion of the logical addresses seen by the core module into the physical address used by the memories. The core multiplexer (referred to as CoreMux in the source code) is connected to its own core and memory multiplexer directly. Connection to the three neighbouring memory multiplexers is accomplished through hierarchical binding of sockets on the exterior of the cell.

Our checkerboard architecture uses 32-bit addresses (Figure 12). The address calculation works as follows. The most significant bit (MSB) denotes whether the address is global or local. Global addresses always have 1 on their MSB and local addresses always have 0. Once an address has been determined as local, bit 30 and bit 29 are checked to find the row of the cell. Bits 28 and 27 are then used to determine the column address of the cell.

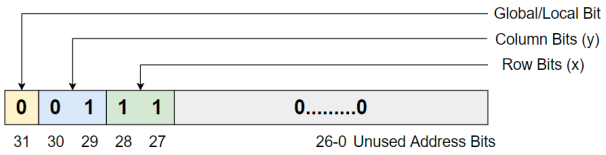

Figure 12: The address calculation method. This address translates to 0x38000000 which implies a local communication to Core 13

Table 8 provides the details on which off-chip memory is accessed.

Table 8: Table for global addresses

Hex Address	Bit 30	Bit 29	Accessed off chip memory
0x80000000	0	0	Off chip memory 0
0xA0000000	0	1	Off chip memory 1
0xC0000000	1	0	Off chip memory 2
0xE0000000	1	1	Off chip memory 3

If the memory is local, which is the case when the MSB is set to 0 the following methodology is used (Table 9). The bits 30 and 29 are used to find the column of the core to which communication must be established. The cells on the even and odd rows differ

in the position of the memory multiplexers. ”if and else” statements are used to determine whether the row is even or odd, using the column address. If the column address matches the core multiplexer's column (which is passed by reference at the elaboration phase), it is confirmed that the targeted memory is either itself or on the opposite side. In this case, bits 28 and 27 are used to check which row the core is trying to communicate to. An if statement is used to check if the core is trying to write to its own memory or not. Depending on this a decision is made on whether the b_transport is forwarded to its own memory or to the opposite one. If the column address differs, then it is checked to determine whether the core wants to write to the top or bottom memory. In this case it is not required to check bits 28 and 27.

3.2.7 Memory Multiplexer Module

The memory multiplexer module is connected to the small memory unit and to its adjacent core multiplexers. Its only purpose is to forward b_transport calls from adjacent core multiplexers to its memory module. It has a varying number of sockets which are dynamically generated depending on its location on the checkerboard.

The cell is a single module which contains the above listed modules and a varying number of sockets depending on its position on the checkerboard. The sockets are pass through sockets which the multiplexers use to connect to the multiplexers of other cells. The cell is the base class from which every instantiated cell inherits. It is responsible for providing the position of the four components inside it and their connections to other cells.

3.2.8 Memory Multiplexer and Cell Modeling Details

The modeling of the memory and the cell is the most challenging aspect of the entire architecture. The challenge arises from the sockets on the edges of cells and their hierarchical binding. In SystemC, sockets cannot be left unbound. Otherwise an elaboration error occurs, as each socket must be bound to a memory. Ideally only a single core module and memory multiplexer module should exist but as the detailed architecture (Figure 9) shows, this is not possible since extra sockets are present on the left and right side as well as on the sides. Another complication is that there can be two types of cells depending on whether the cell is present on an odd or even row. If it is even, the memory and its multiplexer is on the right side with the core and its multiplexer on the left. If it is odd, the entire cell is flipped. Because of this reason, eight different cell types are possible in the 4×4 checkerboard as well as memory multiplexers. Figure 13 shows an example of top left cell on the checkerboard. The initial solution to this is to use the SystemC multi-pass through sockets, which supports unbound sockets. This solution seems to work and the program compiled with no errors or warnings. It is found that a memory could be accessed by only its own core, however other cores were not able to do so. This issue is due to the hierarchical binding of the memory multiplexer's multi pass through sockets to the other core multiplexer of adjacent cores.

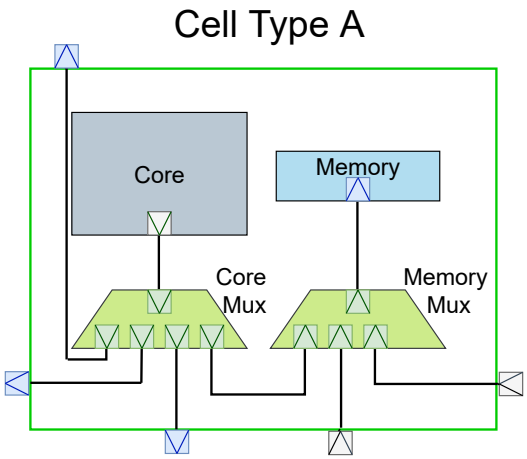

Figure 13: The interior components of a cell. Note that the number of ports may vary on the multiplexers and on exterior of the cell.

Two solutions are implemented. First is the creation of eight different cells and memory multiplexers and the second is to connect unbound sockets to fake initiator sockets in the off-chip memory. Both solutions do not use the multi-pass through socket and instead use regular SystemC TLM-2.0 target and initiator sockets. While the second solution works well, it is not used because of the potential problems that

Table 9: Table for local addresses

Hex Address	Bit 30	Bit 29	Bit 28	Bit 27	Accessed Core
0x00000000	0	0	0	0	Core 00
0x08000000	0	0	0	1	Core 01
0x10000000	0	0	1	0	Core 02
0x18000000	0	0	1	1	Core 03
0x20000000	0	1	0	0	Core 10
0x28000000	0	1	0	1	Core 11
0x30000000	0	1	1	0	Core 12
0x38000000	0	1	1	1	Core 13
0x40000000	1	0	0	0	Core 20
0x48000000	1	0	0	1	Core 21
0x50000000	1	0	1	0	Core 22
0x58000000	1	0	1	1	Core 23
0x60000000	1	1	0	0	Core 30
0x68000000	1	1	0	1	Core 31
0x70000000	1	1	1	0	Core 32
0x78000000	1	1	1	1	Core 33

may arise in the future due to the connection which is extra and unnecessary, and hence the first solution is chosen. The code to create eight different versions of the same cells (first solution) is highly repetitive. It works efficiently but it could be improved through the use of some C++ features.

The options of shortening the code are the use of pre-processor macros, dynamic creation of class objects (the sockets on the cell and the memory multiplexer), inheritance, and templates. Among these options the best solution is the dynamic creation of sockets. Note that the same procedure is used for both the cell and memory multiplexer, because the sockets on the cell simply act as pass through sockets for the real sockets created by the memory multiplexer module. The eight different cells generated are derived from the module Cell_Multi_Type. They are named alphabetically as cell type A,B,C,D,E,F,G and H. Figure 14 shows the complete set of different cell possibilities. Cell types A, B, E, F are colored green in the figure as their core is on the left side of the cell, whereas cell types C, D, G, and H are colored red to show that the core is on the right side.

The dynamic cell type creation is as follows (Figure 14). First the module starts with pointers initialized to the four possible sockets that could exist. Then in the constructor of the module if-else statements are used to check where the cell is on the checkerboard architecture. Depending on the location, sockets are created using the C++ operator 'new'. The 'x' and 'y' parameters passed as references are useful here. Eight if conditions are used to find out and create the new sockets required by the cell. First the 'y' value is checked to determine if it is on an odd or even row. If it is even, then the 'left' and 'down' sockets are created. If the 'y' value is not 0, then the 'top' socket is created. This is because cells on the top edge do not have top sockets for the memory multiplexers.

Next the 'x' value is checked to determine if it is the last cell on the row. If it is then it has only two sockets, and no new sockets are created. Cell type B is created in this method if a top socket was not generated, else it is a cell type F. If it is not the last cell on the row, an extra 'right' socket is created and cell type A is the

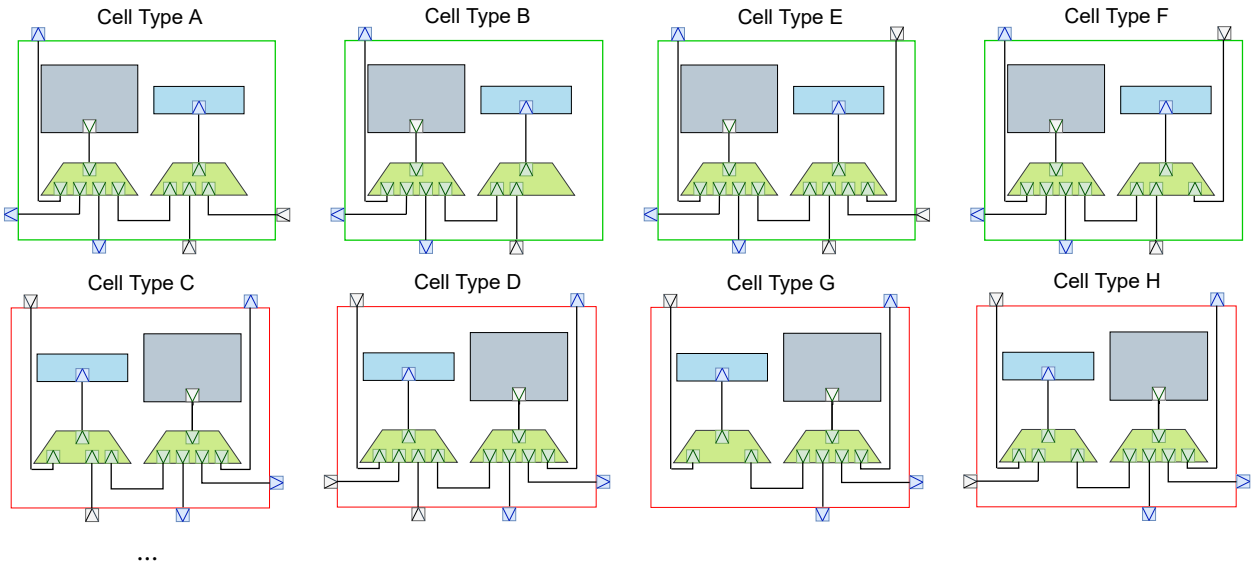

Figure 14: The eight different possible cell types, derived from the cell multi type module.

result if the top socket was not created, otherwise the output is cell type E.

A similar procedure is applied to cells on odd rows. First the 'top' and 'right' sockets are created. If the 'y' value represents the last row on the checkerboard, then the 'down' socket is not created, otherwise it is initialized using 'new'. Finally the 'x' value is checked to see if it is the first cell in the current row. If it is then the 'left' socket is not required. Cell type G is the final cell type if the down socket was not created, otherwise the result is cell type C. If 'x' is not 0 then the final 'left' socket is generated. This creates cell type D if all sockets were created else the final cell is cell type H. Table 10 shows the cell and memory multiplexer socket generation process. To avoid any future problems the pointers of uninitialized sockets are set to 'NULL' . This effective solution reduces the number of lines drastically and also enables communication between the cores. The SystemC model is now also scalable to any even number of cells with this method, as any future cells will be one of these eight types (Figure 14). Detailed discussion of this is present in [38].

3.3 Off-chip Memory Modeling Details

The dynamic creation of sockets turns out to be better than expected and a similar procedure is adopted for the off-chip memories. This modeling also uses templates so that any number of cells can connect to the off chip memories. The off-chip memory is created as a template class which takes the number of sockets as a parameter. Then an array of socket pointers is initialized. In the constructor these sockets are dynamically created using 'new'. The off-chip memory's constructor also takes an additional input parameter which is the memory size. Now the memory multiplexer sockets can be bound in the top module by referencing its sockets as array members. The memory size can be changed at initialization time of the off-chip memory to a different size.

Table 10: Table for cell types

Even or Odd Row	x value	y value	Ports created	Cell/Multiplexer Type
Even	Last Column	0	Left, Down	B
Even	Last Column	Not 0	Left, Down , Up	F
Even	Not Last Column	0	Left, Down, Right	A
Even	Not Last Column	Not 0	Left, Down, Right, Up	B
Odd	0	Not Last Row	Up, Right	G
Odd	0	Last Row	Up, Right, Down	H
Odd	Not 0	Last Row	Up, Right, Left	C
Odd	Not 0	Not Last Row	Up, Right, Left, Down	D

4 Mapping APNG on Checkerboard

For experimental evaluation, it is necessary to map the APNG encoder application on the checkerboard architecture. The mapping on the checkerboard results in change of communication between modules. Since some modules can no longer be connected to a large number of other modules, the data which needs to go to a specific module must be forwarded through neighbouring modules. This is the case for the color splitter and the various filters surrounding it. Forwarding increases the communication cost for each cell. To reduce the cost incurred by forwarding, it is better to send data which can also used by an adjacent cell. For example, sending the three R, G, and B color values to the Up filter through the Subtract filter would be a good choice as the unfiltered values are used by both filters. In most of the cases, each APNG module becomes the main function of a checkerboard cell.

4.1 Checkerboard Mappings

Two checkerboard mappings have been implemented, a simpler initial mapping in which cores perform individual filtering on the three colors on the same core, and an improved mapping which splits the colors between cores to filters. In the initial mapping nine cores are involved in encoding with three cores forwarding data. The improved mapping has every core utilized in APNG encoding.

4.1.1 Initial Checkerboard Mapping

The main idea behind the initial mapping is to avoid the use of excessive forwarding (Figure 15). For instance, the subtract filter receives the red, green, and blue values from the color splitter. Next, it performs the computation and sends both the filtered values and the unfiltered values to the Up filter. The Up filter performs its own computation and sends the filtered Sub, Up and unfiltered values to the Paeth filter. The Paeth filter then computes the Paeth filtered output, and sends it to the Comparator while also forwarding both outputs from the Up filter and Sub filter. The same process is performed in the Average filter route. At the Comparator the least sum filtered row is chosen and sent to the Compressor. The compressed output is forwarded through neighboring cells to the right and written to the output file using the monitor. It is also sent to the APNG Encoder module which creates chunks necessary for APNG encoding and forwards it to the second monitor thread.

4.1.2 Improved Checkerboard Mapping

While the initial mapping works, it could be further improved by splitting the filtering work of each core to three cores with each core filtering a different color component. This is because some filters, such as the Paeth filter take too much time to compute. Since there are sixteen cores available, it is possible to map every core to one module (Figure 16). This mapping should technically be faster as the filter computation is now approximately three times faster. As the checker-

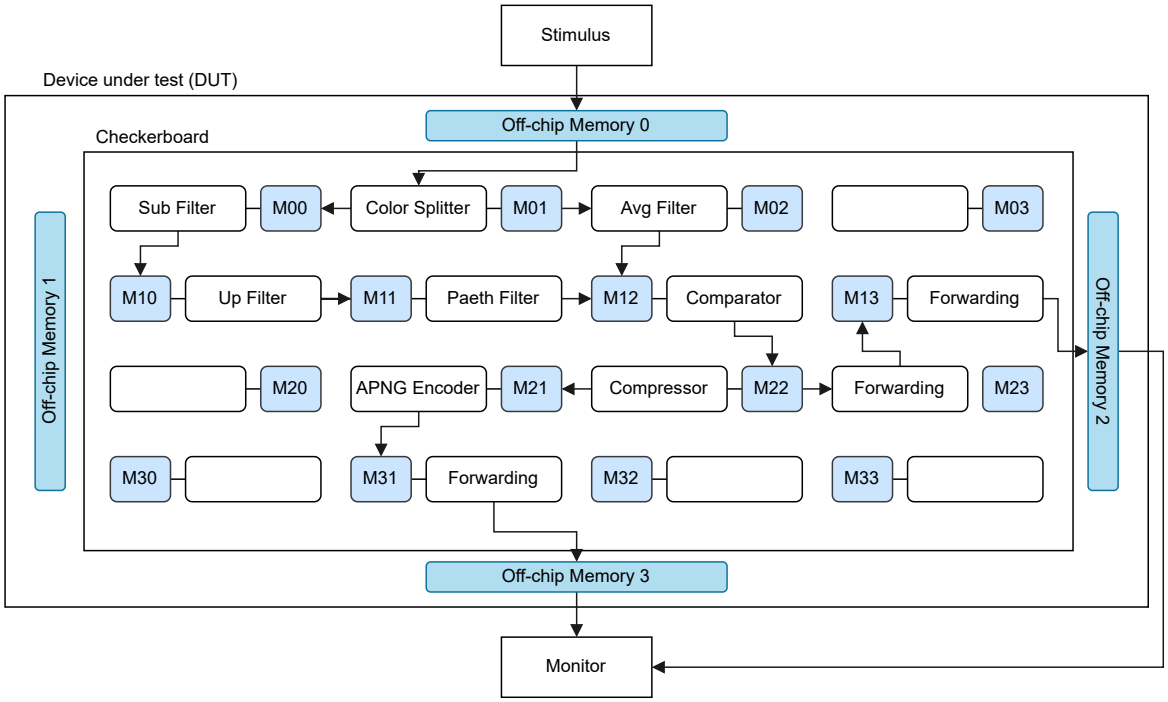

Figure 15: An initial mapping that was successfully implemented on the checkerboard.

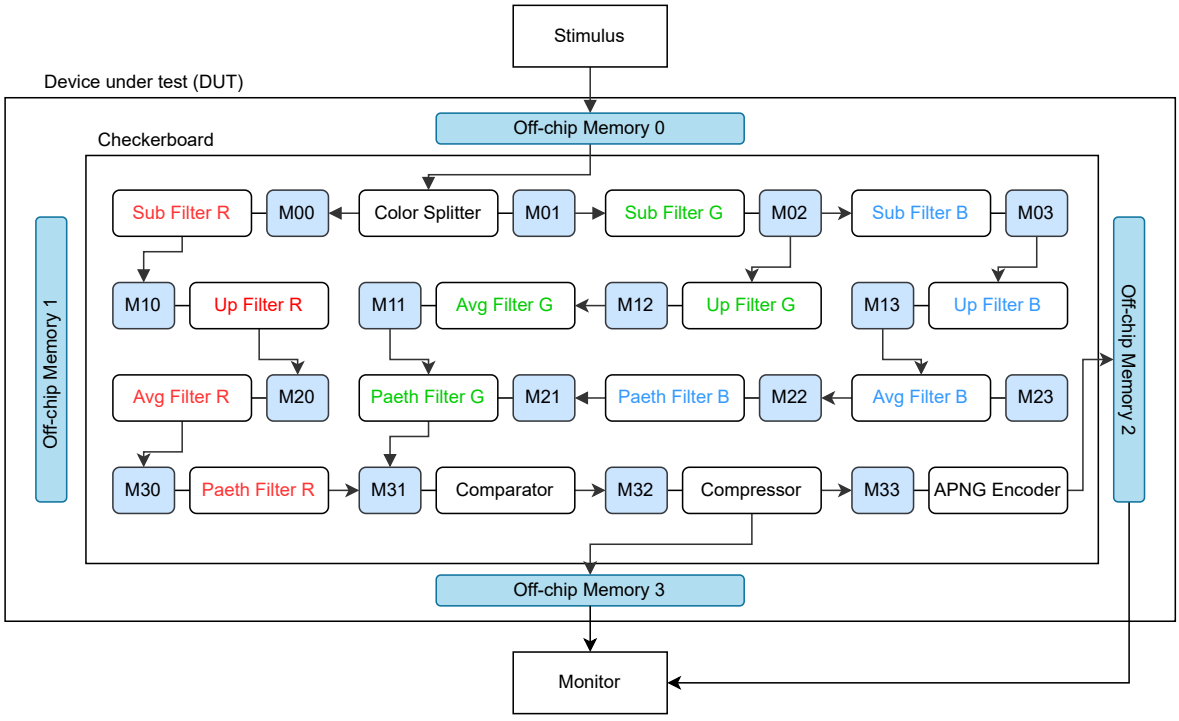

Figure 16: The improved mapping which splits computation more evenly.

board model is pipelined, the cycle time becomes the computational delay of the module which takes the longest time. This concept is presented in the chapter based on pipelining in [21]. The exact computational delay of each module is presented in the experimental results section.

4.2 Models for comparison

To provide further comparisons, a single core model and two shared memory processor models are created in SystemC. These models use a single large global memory with a templated memory multiplexer connecting the cores to the memory. The memory is of the same speed as the off-chip memory in the checkerboard model. The single core model (Figure 17) performs every one of the computations in only one core and is therefore not pipelined. The shared memory models work very similar to the checkerboard architecture, using the same communication functions to transfer pixel data between cores, but have a greater amount of contention.

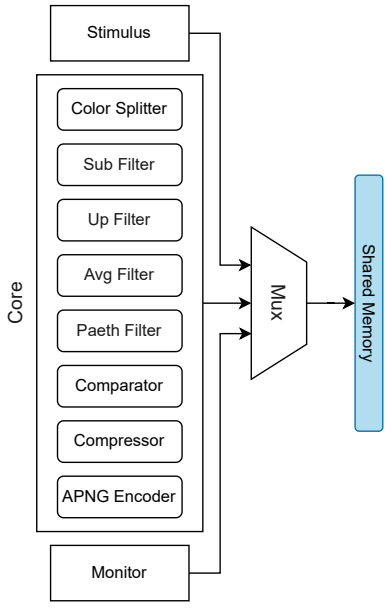

Figure 17: Single core model with no pipelining. Some contention is present due to sharing the multiplexer with the stimulus and monitor modules.

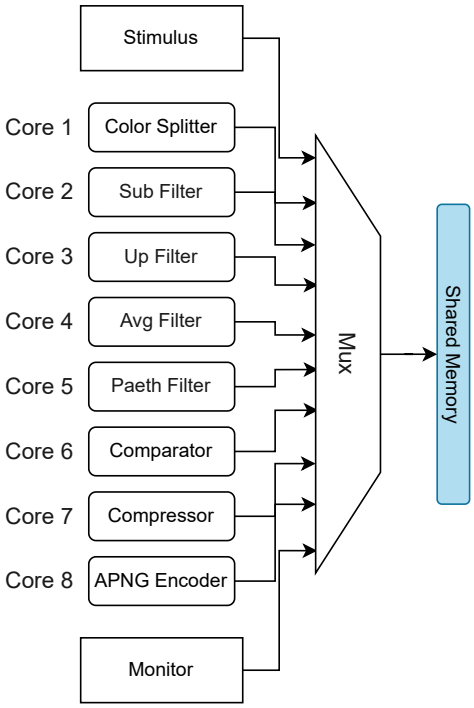

Figure 18: Shared memory processor with 8 cores. The filters for the R,G and B pixels are combined in this case.

The SMP with 8 cores (Figure 18) performs the filtering operation on the different color channels in the same core. The computation time per core is increased, which subsequently reduces communication per unit time. Doubling the number of cores provides a 16 core model (Figure 19) in which the cores perform less work but communication is increased per unit time leading to higher chances of contention. This model is similar to the better checkerboard mapping and the 8 core SMP matching the initial checkerboard mapping.

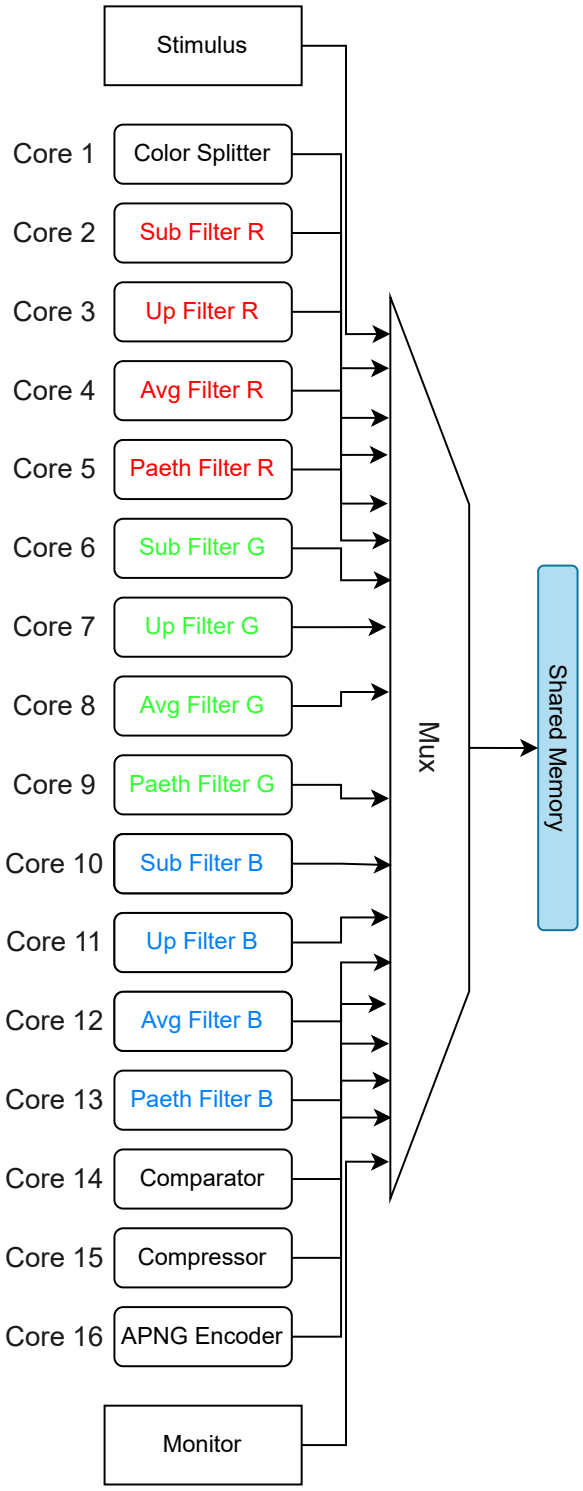

Figure 19: Shared memory processor with 16 cores, with the computation work split between multiple cores.

5 Experimental Results

With a total of five different models for APNG encoding implemented in SystemC, it is possible to compare execution times and benchmark the performance of the modeled architectures.

5.1 Modeling of Delays

To measure experimental results, it is desirable to reflect timing in the models. There are three types of timing in the model - (1) memory access delays, (2) delays caused due to contention between cores, and (3) computation delays. The computation time of each module is estimated by running the gprof tool on Linux. Timing is first computed by running the encoder a few times and then averaged to get an approximate computation time per pixel. The number of pixels also changes for different resolution images which is why this calculation is done. These values are measured on an i5-1135G7 at 2.40 Ghz CPU with 8 cores. SystemC is single threaded, so only 1 core is actually used during the execution of the program [39]. These values are dependent on the processor clock rate, if the processor is faster or slower then the values are scaled linearly. The observed timing for computational delays is shown in Table 11. These delays are measured by running the APNG encoder model in Figure 7 on 30 images of the UCI Engineering Hall with a resolution of 2704 $\times$ 1502 for five times. A sample APNG frame is shown in Figure 20. The time of each module is averaged and rounded. This value is divided by 30 (number of frames) to get the time each module spends per frame. Time per frame is further divided by 3 (as there are R,G and B color channels) and image resolution to obtain the per pixel delay. These values change every time the encoder is run on the same set of images, and even more so when they are run on different images. Therefore they are rounded and are only approximations of the computation delays. A detail to be noted in this table is that the filtering operations are the most time consuming in the whole encoder. Therefore it is more important to make the filters operate in parallel compared to the other modules, which was the reason for creating the improved mapping.

Next the communication delays must be considered

Table 11: Table for computation delays measured at 2.4 Ghz on 30 images of 2704 $\times$ 1502 resolution of the UCI Engineering Hall

Module Name	Total time for encoder	Time per frame	Time per pixel
Color Splitter	4s	0.133s	11ns
Subtract Filter	30s	1.000s	82ns
Up Filter	33s	1.100s	88ns
Average Filter	50s	1.667s	137ns
Paeth Filter	102s	3.400s	274ns
Comparator	8s	0.267s	21ns
Compressor	14s	0.467s	38ns
APNG Encoder	1s	0.033s	3ns

(Table 12). For this purpose the memory access times have been referred from [21]. The off-chip memory is assumed to be made of slower DRAM, and the on-chip memory is assumed to be made of high speed SRAM. [21] does not have much information about multiplexer delays, so a realistic 4x1 multiplexer has been considered such as the AD8170/AD8174, which has an approximate fast switching time of 10 ns [40].

Table 12: Table for communication delays

Delay Name	Time in Nanoseconds
Off-chip Memory Read Time	50
Off-chip Memory Write Time	50
On-chip Memory Read Time	2.5
On-chip Memory Write Time	2.5
Multiplexer Switching Time	10

Another important metric is the contention time, which is how much time each core spends waiting for access to the memory. Memory accesses have been modeled so that only a single core can access a memory at any time when contention is turned on [41]. This value is based on the communication delays that have been computed, as they control how long it takes for a core to access the main memory. Contention does not have any specific delay value associated with it defined by the user.

Note that the absolute delay times do not really matter here. We are interested only in the relative timing of the models when compared against each other.

Figure 20: A sample image (Rainer Dömer, EECS 222 course material) compressed by the APNG encoder. The APNG encoder compresses data without any loss.

5.2 Measurement of Delays

To measure delays we create three global variables of the type sc_time called Computation_Time, Read_Time, Write_Time and Contention_Time. These variables provide an accurate value of how the time in each SystemC model was spent, and are present in every model. Using them it is possible to also come up with formulas to compute the total execution time in some cases. The Computation_Time is the summation of time spent by every core on the values listed in Table 11, and is meant to represent how much time a real computer would have spent in compressing the data provided to it. Read_Time and Write_Time are other important variables which keep track of the total time spent by the model accessing the memory. Added

together they are called as the Communication_Time or the RW_Time. The Checkerboard model does this in a more detailed way explained below. Lastly the Contention_Time is the total time spent by every core waiting to access the memory. This time can easily exceed the execution time of the model if there are a lot of cores attempting to access the memory at the same time, as each core will be waiting to access the memory and all of these times add up to get the Contention_Time.

For the Checkerboard models these variables have also been instantiated for each and every core as well as for the off-chip multiplexers. This information is useful as it provides a way to find which core takes the longest to complete its task of computation and communication, and the longest stage of the pipeline can be identified. Another reason for this is that it gives an idea of how contended each individual memory is, and whether the modeling of the checkerboard is successful in addressing the important underlying issue of memory contention leading to the memory bottleneck.

5.3 Simulated Timing Results

With the three types of possible back-annotated, it is possible to obtain results on the five SystemC models as well as derive formulas predicting the total execution time in some of the models. There is a one second reset delay in every model, this is mainly provided to model the effect that when a computer is turned on the memory values have random values and need to be initialized to zero. Table 13 shows the variation in model timings as the clock rate is gradually increased, thereby decreasing computation time per module and increasing the frequency of memory access requests. The test images are 2704×1502 PPM images of the UCI Engineering Hall with 10 frames which are compressed and added to the APNG file. The five SystemC models shown are the single core, 8 core shared memory, 16 core shared memory, initial checkerboard mapping, and the improved checkerboard mapping. These models are compared with each other at different processor clock rates to observe the decrease in execution time as the computation speed of the processor increases. The clock rates that have been chosen start at 0.25 Ghz and are doubled until it reaches 8 Ghz. These values are reasonable assumptions as these have been the approximate CPU clock rates for the past two decades. The Pentium-II processor had an approximate clock rate of 266 Mhz [42] in the year 1997. Modern CPUs today have a clock rate of 4-5 Ghz and 8 Ghz may be reachable soon.

An important assumption made in the case of the five models is that no cache memory exists. The reasoning for this assumption is that we want to observe memory contention directly, undisturbed by caching behaviour. Also, the implementation of a cache is quite complicated and will be added later in a future work involving replacement of cores with a real virtual core which uses an instruction set simulator.

Table 13 shows the comparison of the three types of delays and the overall execution time. The table shows the simulated variation in execution time over the years as processor speed rises, and how the delay values are affected. Time spent in computation linearly reduces, but the contention time goes up in every model, to varying degrees. Time spent in accessing memory remains nearly constant, which matches the real world case in which memory access speeds are not as greatly improved as processor speeds. The increase in contention time is the reason why execution time start to show less improvement (Table 13). Computation time varies slightly for the single core model as part of the source code involved with timing was changed more compared to the other models.

Table 14 shows the execution time and other delays when either the computation or contention delay is turned off. This is useful in observing what the hypothetical increase in performance would be if only communication delays existed. The results in this table also provide a way of finding out if any co-relation exists between the three delays and whether the execution time can be estimated from them.

Using the results in Table 14 it is possible to make some empirical estimates on the execution times. For example, in the single core model the execution time when both computation and contention delays are switched off is approximately half of the communication time (RW time). This is because the communication from Stimulus to the Core and Core to the Monitor can happen at the same time (Figure 17), which results in only half the time spent for communication. When contention is turned on, it becomes the communication time with an additional one second caused due to the reset delay. This is also observed in the case of the SMP models. Again in the single core case, when contention is turned off but computation is turned on, it is observable that the execution time becomes the summation of the computation time and half of the communication time. When all three delays are present the execution time for the single core model becomes-

$$Execution\ time = Computation\ time + (Communication\ time + Contention\ time)/2 + 1$$

This formula does not give the exact execution time because it does not consider the multiplexer switching time or the presence of overhead in the encoder. However the predicted execution time is still quite accurate. This empirical formula is possible because of the single core model is simple and has no parallelism or pipelining.

5.4 Observations and Comparison

Plotting the values from Table 13 provides insight on the change in execution time as the clock rate is doubled (Figure 21). For the single core model it is seen that the decrease in computation speed leads to great increase in speed up, until a certain point where diminishing returns are observed. This becomes noticeable at around 4 Ghz. The plot for the single core resembles a multiplicative inverse. The shared memory processors start off with low execution time, but they start to stagnate at around 1 Ghz. The lack of cache severely limits the performance of the SMP models (which we intentionally omitted as mentioned above).

The initial checkerboard model starts with an execution time similar to the 8 core SMP model but quickly decreases in execution time as the clock rate is increased. In fact it performs slightly better than the improved checkerboard at 8 Ghz. The improved checkerboard is the fastest model at low clock rates, but at higher clock rates it becomes only about as fast as the initial checkerboard. To understand this better it is necessary to know the factor affecting execution time on the checkerboard. This important component which determines the checkerboard model's execution time is the longest stage in the pipeline. Using the per-core timings it is possible to make an estimate on total execution time of the checkerboard models. At 1 Ghz, the computation time of core 11 in the initial mapping is about 79 seconds, this high computation time is caused by the Paeth filter. The total execution time of the model is 84 seconds. Therefore core 11 takes as much as 94 % of the execution time. The total execution will have some overhead but the longest stage is still a nearly accurate measure of how long it will take to execute the model. For the improved checkerboard core 21 is the longest stage in the pipeline because it performs the computation of a Paeth filter component and also performs forwarding of the computed filter values to the Comparator through memory 31. These delays must be added as the pipelining is not present for these operations. The computation takes about 27 seconds and the memory accesses takes 3 seconds, which gives 30 seconds for total time of 30 seconds. The total execution time is 34 seconds, so 88 % of the time is represented by the longest pipeline stage. As the core clock rates reach 8 Ghz, the longest stage in the pipeline is no longer core or on-chip memories, but the off-chip memory 3 on the bottom of model. This memory has contention from core 32 and the Monitor module which leads to it taking about 17 seconds as the communication time. Since the model execution time are 19 seconds this memory now becomes the largest stage in the pipeline taking about 90% of the time.

It is now possible to visualize the memory bottleneck. The increase in number of processors does not seem to cause a decrease in execution speed if the memory is shared. The contention creates a situation where the cores keep competing and even if the computation time is negligible, the communication time along with the contention time result in a situation of

Table 13: Table for simulated timing results

Model Name	Execution Time	Computation Time	RW Time	Contention Time
Clock rate of 0.25 Ghz				
Single Core	746.693s	705.599s	46.195s	33.972s
SMP with 8 Cores	350.038s	707.231s	185.237s	378.815s
SMP with 16 Cores	265.042s	707.231s	185.220s	471.747s
Initial Checkerboard	324.495s	707.231s	57.528s	4.564s
Improved Checkerboard	115.103s	707.231s	58.030s	2.648s
Clock rate of 0.5 Ghz				
Single Core	393.895s	352.799s	46.195s	33.972s
SMP with 8 Cores	207.695s	353.615s	185.236s	392.710s
SMP with 16 Cores	208.789s	353.615s	185.221s	529.793s
Initial Checkerboard	164.488s	353.615s	57.528s	4.564s
Improved Checkerboard	59.593s	353.615s	58.030s	2.480s
Clock rate of 1 Ghz				
Single Core	217.495s	176.400s	46.195s	33.973s
SMP with 8 Cores	186.233s	176.808s	185.231s	428.378s
SMP with 16 Cores	190.746s	176.808s	185.219s	587.320s
Initial Checkerboard	84.484s	176.808s	57.529s	4.681s
Improved Checkerboard	32.269s	176.808s	58.030s	2.750s
Clock rate of 2 Ghz				
Single Core	129.295s	88.200s	46.195s	33.973s
SMP with 8 Cores	186.233s	88.404s	185.231s	457.149s
SMP with 16 Cores	187.222s	88.404s	185.219s	595.828s
Initial Checkerboard	44.482s	88.404s	57.529s	6.013s
Improved Checkerboard	21.020s	88.404s	58.030s	4.096s
Clock rate of 4 Ghz				
Single Core	85.196s	44.100s	46.195s	33.973s
SMP with 8 Cores	186.233s	44.202s	185.231s	455.554s
SMP with 16 Cores	186.720s	44.202s	185.218s	619.839s
Initial Checkerboard	24.482	44.202	57.529	8.362s
Improved Checkerboard	19.002	44.202	58.031	6.837s
Clock rate of 8 Ghz				
Single Core	63.146s	22.050s	46.195s	33.973s
SMP with 8 Cores	186.232s	22.101s	185.231s	434.962s
SMP with 16 Cores	186.470s	22.101s	185.218s	644.136s
Initial Checkerboard	18.005s	22.101s	57.530s	16.140s
Improved Checkerboard	19.002s	22.101s	58.031s	7.308s

stagnant performance. To improve execution time architectures which have very limited contention should be considered. This results in a sustainable decrease in execution speed as the clock rate is increased.

Figure 22 shows the increase in contention as the clock rate increases. When the clock rate increases processors can perform more instructions per second, thereby reducing computation time. This increases the rate at which memory is accessed by the cores so that they can process more data, which leads to a rise in contention time. Contention decreases slightly for the 8 core model, possibly because the increase in clock rate makes the cores access data in a more scheduled way. But this is not the case for the 16 core CPU, as the contention is steadily increasing.

For the single core model contention exists because the cores are accessing the memory along with the stimulus and monitor. This contention is however minimal, and the checkerboard models have even less contention. The initial checkerboard mapping suf-

Table 14: Table for the increase in timing due to added contention and computation delays at 4 Ghz

Timings included	Execution Time	Computation Time	RW Time	Contention Time
Single Core at 4 Ghz				
None	24.115s	0s	46.195s	0s
Only contention	47.195s	0s	46.195s	46.167s
Only computation	68.215s	44.100s	46.195s	0s
Both	85.196s	44.100s	46.195s	33.973s
Shared Memory Processor with 8 cores at 4 Ghz				
None	64.081s	0s	185.231s	0s
Only contention	186.232s	0s	185.231s	296.704s
Only computation	64.582s	44.202s	185.231s	0s
Both	186.233s	44.202s	185.231s	455.554s
Shared Memory Processor with 16 cores at 4 Ghz				
None	51.165s	0s	185.217s	0s
Only contention	186.219s	0s	185.217s	483.542s
Only computation	57.538s	44.202s	185.219s	0s
Both	186.720s	44.202s	185.218s	619.839s
Checkerboard Initial Mapping at 4 Ghz				
None	17.999s	0s	57.530s	0s
Only contention	18.002s	0s	57.530s	16.560s
Only computation	24.466s	44.202s	57.529s	0s
Both	24.482s	44.202s	57.529s	8.362s
Checkerboard Improved Mapping at 4 Ghz				
None	18.998s	0s	58.031s	0s
Only contention	19.001s	0s	58.031s	7.979s
Only computation	18.999s	44.202s	58.031s	0s
Both	19.002s	44.202s	58.031s	6.837s

fers from a linear increase in contention, but the improved checkerboard remains constant. Therefore the checkerboard model is a good alternative to shared memory processors as clock rate increases, as its contention is much less.

6 Conclusion and Future Work

The increase in processor speeds over the years has resulted in much faster computers but this trend has been hampered due to slower memory speed increases. The newly proposed checkerboard architecture may be one possible way to mitigate the effects of slower memory access speeds. This would require a substantial amount of effort from engineers from different aspects of computer engineering. The entirety of CPU architectures may need to be modified as almost all of the CPUs in the market are based on shared memory. Writing programs for this type of architecture will be challenging and as seen in the case of the initial and improved mapping, the improved mapping was slower than the initial mapping at high clock rates. Manually assigning a thread to each core can be time consuming and not ideal as the performance varies depending on many factors. The compiler for such an architecture would need to be much more advanced to make the decision on which core should be assigned a specific thread. In other words, the compiler must also be responsible for the mapping operation presented in Section 3.

Before implementing this architecture on a real die it will be necessary to perform further virtual prototyping. A good next step would be to replace the current cores with high-level RISC-V virtual CPUs. These cores can be borrowed from [43] and they have working caches. With such an architecture, the de-

Figure 21: APNG Encoder execution time scaling with increase in clock rate for the five SystemC models.

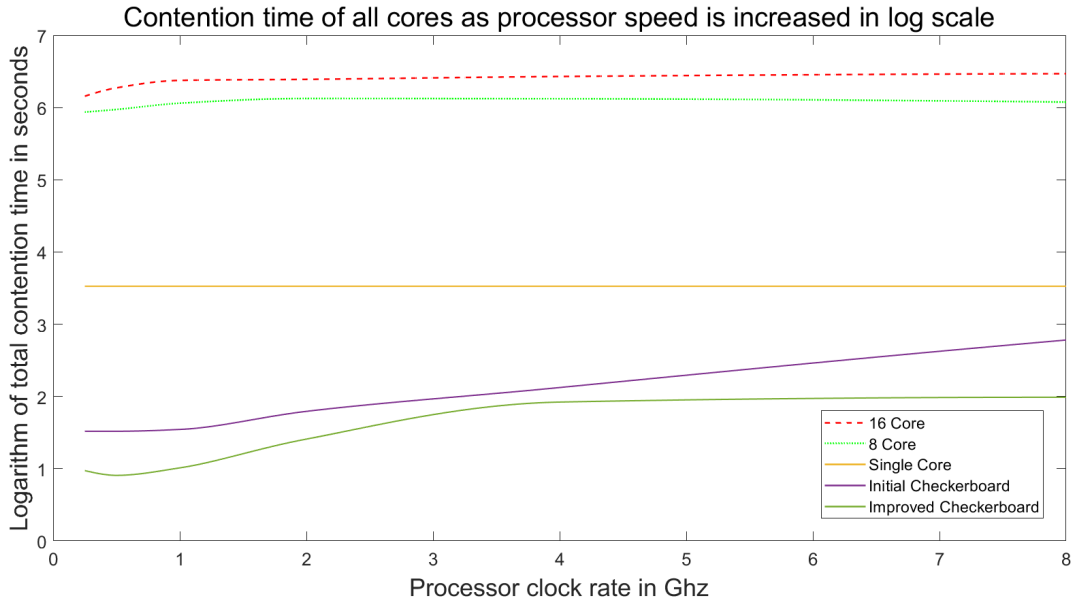

Figure 22: Contention time scaling with increase in clock rate for the five SystemC models.

tailed performance of the checkerboard architecture can be compared with the shared memory processor which will now include a cache. Then a more definitive statement can be made at which clock rate contention becomes severe enough that the execution time of most programs becomes limited.

References

[1] Andreas Gerstlauer, Christian Haubelt, Andy D Pimentel, Todor P Stefanov, Daniel D Gajski, and Jürgen Teich. Electronic system-level synthesis methodologies. *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, 28(10):1517–1530, 2009.

[2] Daniel D Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification language and methodology*. Springer Science & Business Media, 2012.

[3] Wolfgang Müller, Wolfgang Rosenstiel, and Jürgen Ruf. *SystemC: methodologies and applications*. Springer Science & Business, 2007.

[4] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D Gajski. *System design: a practical guide with SpecC*. Springer Science & Business Media, 2001.

[5] Preeti Ranjan Panda. SystemC: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 75–80, 2001.

[6] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (IEEE Cat. No. 03TH8721)*, pages 19–24. IEEE, 2003.

[7] Rainer Dömer. A Grid of Processing Cells (GPC) with Local Memories. Technical Report CECS-TR-22-01, Center for Embedded Computer Systems, April 2022.

[8] Vivek Govindasamy. Mapping of an APNG Encoder to the Grid of Processing Cells Architecture. Master's thesis, UC Irvine, 2022.

[9] Arya Daroui, Vivek Govindasamy, Yutong Wang, and Rainer Dömer. Modeling of a 4-by-4 checkerboard GPC in SystemC. Personal communication, October 7 2021.

[10] APNG specification. https://wiki.mozilla.org/APNG_Specification.

[11] ISO functional specification. https://www.iso.org/standard/29581.html.

[12] Android mobile phone camera specifications. https://developer.android.com/guide/topics/media/media-formats.

[13] W3 Functional Specification. https://www.w3.org/Graphics/JPEG/itu-t81.pdf.

[14] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, RFC 1950, May, 1996.

[15] John Von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[16] William Stallings. *Computer organization and architecture: designing for performance*. Pearson Education India, 2003.

[17] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[18] Gene Frantz. Digital signal processor trends. *IEEE micro*, 20(6):52–59, 2000.

[19] Sally A McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162, 2004.

[20] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[21] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[22] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A Case for {NUMA-aware} Contention Management on Multicore Systems. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[23] Vivek G. A SystemC Model of a PNG encoder. Technical report, 2020.

[24] PNG chunk details. https://www.w3.org/TR/PNG-Chunks.html.

[25] William Wesley Peterson and Daniel T Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[26] Theresa Maxino. Revisiting Fletcher and Adler checksums. 2006.

[27] Offical LibPNG webpage. http://www.libpng.org/.

[28] Filter selection method 1, official w3 PNG webpage. https://www.w3.org/TR/2003/REC-PNG-20031110/#12Filter-selection.

[29] John Miano. *Compressed image file formats: Jpeg, png, gif, xbm, bmp*. Addison-Wesley Professional, 1999.

[30] Official PKZIP webpage. https://www.pkware.com/pkzip.

[31] James A Storer and Thomas G Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.

[32] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[33] Official zlib webpage. https://zlib.net/.

[34] Official *DEFLATE* webpage. https://tools.ietf.org/html/rfc1951#section-Abstract.

[35] Portable Network Graphics CRC code. https://www.w3.org/TR/PNG-CRCAppendix.html.

[36] Michael Still. *The definitive guide to ImageMagick*. Apress, 2006.

[37] Mark Adler. Example of using DEFLATE. https://zlib.net/zpipe.c%20II/24333503.PDF.

[38] Yutong Wang. Scalable GPC Architectures. Master's thesis, UC Irvine, 2022.

[39] Guantao Liu, Tim Schmidt, and Rainer Dömer. RISC compiler and simulator, beta release V0.3.0: Out-of-order parallel simulatable SystemC subset. 2016.

[40] AD8170 and AD8174 Multiplexer datasheet. https://www.mouser.com/datasheet/2/609/AD8170_8174-1502101.pdf.

[41] Emad Malekzadeh Arasteh and Rainer Dömer. Improving Parallelism in System Level Models by Assessing PDES Performance. In *2021 Forum on specification & Design Languages (FDL)*, pages 01–07. IEEE, 2021.

[42] Pentium II Datasheet. http://datasheets.chipdb.org/Intel/x86/Pentium%20II/24333503.PDF.

[43] Vladimir Herdt, Daniel Große, Hoang M Le, and Rolf Drechsler. Extensible and configurable RISC-V based virtual prototype. In *2018 Forum on Specification & Design Languages (FDL)*, pages 5–16. IEEE, 2018.