HARDWARE SYNTHESIS OF A PARALLEL JPEG DECODER FROM ITS FUNCTIONAL SPECIFICATION

John Hawkins and Ali E. Abdallah
Centre for Applied Formal Methods,
London South Bank University,
103 Borough Road,
London SE1 0AA, U.K.
John.Hawkins@lsbu.ac.uk, A.Abdallah@lsbu.ac.uk

Abstract: Recent advances in manufacturing programmable logic devices, such as the FPGA, have made it possible to obtain reconfigurable circuits with upwards of one hundred million gates. Although we have such enormously powerful hardware at our fingertips, we are still somewhat lacking in techniques to properly exploit this technology to its full potential. In previous work, we have proposed a development methodology based on transformational programming and process refinement for producing provably correct solutions. Starting with a clear, intuitively correct specification of the problem, in a functional language such as Haskell, we apply a set of formal transformation laws to refine it into a behavioural definition in Handel-C, exposing the implicit parallelism along the way. This definition can then be compiled onto an FPGA. We apply this technique to a non-trivial, real world problem - a JPEG decompression algorithm, and achieve a truly scalable, parallel hardware implementation.

1. INTRODUCTION

Greatly increased efficiency in solutions to real world problems can be achieved through parallelism and implementation in hardware. Unfortunately this comes at a cost; principally in terms of complexity. This complexity, coupled with the increased consequences of making mistakes, can make this a very costly process indeed.

A good example of a class of real world problems to illustrate these issues is that of image compression, particularly the JPEG standard [10, 11]. JPEG decoders and encoders are widely used, but not nearly as widely understood. Developers tend, quite naturally, to rely on tried

and tested library code under normal circumstances. However, in some situations performance requirements force developers to leave behind the comfort of such libraries and look to new implementations.

At this point it is very important to have clear, unambiguous, and if possible, provably correct specifications to work from. We propose that functional programming languages [4], such as Haskell [5], facilitate exactly this. Indeed, we are not the first to consider the functional style a good foundation for a specification of JPEG decompression [7]. Not only do functional languages provide a good framework for specification, but also they give us scope for transformation and refinement not present in imperative languages. Such capabilities allow an efficient and correct implementation to be derived formally, and in part mechanically from the specification, by exploiting known efficient implementations for commonly used patterns of computation. This approach is often broadly referred to as patterns or skeletons [1, 2, 6].

Recent advances in FPGA chip manufacturing, coupled with the emergence of higher level compilation tools, such as Celloxia's Handel-C compiler [8], have together revolutionised hardware design from the exceedingly costly process it once was, to now being within the grasp of even the smallest company or academic institution. The Handel-C language has many desirable features including CSP [9] style communication, and an explicit means for denoting parallelism. However, it is an imperative language (being based on C), and as such, we argue, does not necessarily form a good basis for specifications, nor a good starting point for deriving a parallel algorithm. In this work, we use Handel-C as a target for implementation, deriving code in this language from specifications given in a functional style.

The rest of this work proceeds as follows. In Section 2, we give a brief overview of the notation used, and introduce the concept of refinement to explain how behavioural implementations can be derived from functional specifications. In Section 3 we discuss some issues relevant to the JPEG decompression process. In Section 4 we provide a functional specfciation of a JPEG decompressor. Then, in Section 5, we use this specification to derive a parallel implementation in Handel-C. This paper concludes in Section 6.

2. NOTATION AND REFINEMENT CONCEPTS

Functional Notation. As already noted, functional languages such as Haskell provide an extremely good environment for clear specification of algorithms. Details of functional notation in general can be found in

[4], with more specific information relating to Haskell in [5]. Also, certain aspects and properties of the particular notation we use in this work are explored in [1–3].

Handel-C. Handel-C [8] is a C style language, and fundamentally imperative. Execution progresses by assignment. Communication is effectively a special form of assignment. As previously noted, communication in Handel-C follows the style of CSP. The same operators are used for sending and receiving messages on channels (`!` and `?`), and communication is synchronous - there must be a process willling to send and a process willing to receive on a given channel at the same time for the communication to take place. Additionally, note that channels in Handel-C are typed - this is so the compiler knows how wide to make them. Parallelism in Handel-C can be declared with the `par` keyword. Handel-C has an equivalent of CSP's choice operator in the form of the `prialt` statement.

Refinement. Having stated our specification environment (Haskell) and our target environment (Handel-C) it is now necessary to consider how we are to refine definitions in one to the other. These techniques are explained in more detail elsewhere [1–3], we shall provide only a very brief overview here.

Data Refinement. Given that our implementation in Handel-C will rely on message passing, we need to consider how the types derived from our specification will be communicated. Most interesting to us are list types, and we will examine the alternative refinements for these here. Broadly we have two intuitive strategies for communication of a linear data structure (i.e. a list) - either sequentially or in parallel. We term these techniques streams and vectors respectively.

Streams facilitate a functional, or pipeline parallel scheme. To communicate a list as a stream, we send each value in order along a channel, and then signal the end of transmission (EOT). Although there are a number of possible options for how to signal the end of transmission, we have found the use of a second single bit channel the most widely applicable.

Vectors implement a data parallel scheme. To communicate a list as a vector each item is communicated independently, in parallel, on its own channel. There may be several variations to the vector, depending on the type of the items in the list.

These two structures may then be combined together to form refinements for lists of lists. One example of this is the vector of streams,

which is a parallel composition of n streams, each communicating a sublist independently as a stream. Another example is the stream of vectors, in which at each stage an entire sublist is communicated in a single step, in parallel

Process Refinement. Higher-order functions in our specifications can be refined into Handel-C implementations from a library of processes. We may have more than one implementation for any given higher order function depending on the setting in which we choose to use it (i.e. with streams or vectors). More detail on higher order process refinement can be found in [3].

As noted, the composition operator forms an important part of functional definitions. In terms of processes and parallelism, functional composition maps on to pipelining. Given a process `P` that outputs on a particular channel, and a process `Q` that takes input of the same type on a particular channel, we can pipe the result from one to the other simply by parameterising the name of their respective output and input channels and composing them together in parallel. This simple but powerful scheme can apply to both the stream and vector setting. We can pass in streams and vectors as parameters to processes in exactly the same way as we would simple channels.

3. JPEG DECODING

We shall focus our efforts on a decoder for JPEG's baseline DCT method of compression. This is almost certainly the most commonly used method within the JPEG set of standards.

We shall require the use of restart markers in our compressed data. A JPEG decoder must maintain a set of predictors. The predictors will be modified each time a unit of data is decoded, and their values will affect the decoding of each unit. As such, for every single unit in the compressed file, we require that the previous unit has been at least partially decoded before it in turn can be decoded. This makes for a largely sequential decoding process. Thankfully, the JPEG standard recognises applications in which JPEG images might be communicated over unreliable media, and as such, data may have been lost part way through transmission. To this end, the standard includes the definition of restart markers. Whenever one of these markers is encountered, the predictors can be safely reset. This has the effect of defining a number of sections within the compressed data that can be decoded completely independently of each other.

It is important to clearly consider the hierarchy within a compressed JPEG file, when considering writing the specification for a decoder. To

begin with we have a file. This can be split into two areas, the headers and the compressed scan data. The headers contain information about the compressed data (size, format and so on) as well as tables for dequantization and Huffman decoding.

Where restart markers are used, the scan can be decomposed into a number of independent sections which we shall call intervals. An interval can be further decomposed into one or more minimum coding units (MCUs). The number of MCUs per interval is defined in the headers. The MCU is a collection of units. Each unit, when fully decompressed, will form an 8×8 matrix of samples for a given component (usually one of Y, C_b or C_r for colour images). Generally, the chromincance components will be downsampled to achieve better compression. A typical scheme has an MCU representing a 16×16 block of pixels in the fully decoded output image. Within this, there will have been a unique Y (luminance) value for every pixel. However, each chrominance value will be shared by a 2×2 pixel block. As such, an MCU in this scheme will contain four units of Y samples, followed by one of C_b samples, and one of C_r samples.

4. FUNCTIONAL SPECIFICATION

We may find the following type definitions useful. A unit is an 8×8 matrix of coefficients (before transformation) or samples (after transformation). An MCU is a list of units. These types may therefore be defined as follows:

```
type UnitRow = [Int]
type Unit = [UnitRow]
type MCU = [Unit]
```

Now, to consider the functions that will comprise our decoder. At the highest level we require a function that will take in a list of compressed bytes representing the entire file, and will return an uncompressed image.

```
decodeJpeg :: [Byte] -> Image
decodeJpeg data = decodeScan hdrInfo scanData
 where (scanData,hdrInfo) = decodeHeaders data
```

An `Image` here can be considered as a simple two dimensional array of pixel values. This definition relies on two auxilary defintions. The first decodes the headers in the data, and returns both a `HeaderInfo` object and a list of the remaining data in the file, following the headers.

```
decodeHeaders :: [Byte] -> ([Byte],HeaderInfo)
```

The exact definition of decodeHeaders and the HeaderInfo type will not be shown in full here due to lack of space. Broadly, the header information should include all the numeric parameters and structures required for decoding. The second function, decodeScan, is where the bulk of the decoding effort takes place.

```
decodeScan :: HeaderInfo -> [Byte] -> Image
decodeScan hdrInfo = composeImage hdrInfo .
                     map (decodeInterval hdrInfo) .
                     readIntervals
```

This function is a composition of three stages. In the first, we use the function readIntervals to split the compressed scan data into a list of intervals which can be decoded independently of each other. Next, we map the function decodeInterval to each interval in the list of decoded sections within the image. Finally we apply composeImage to compose these sections together, a function which we shall keep deliberately vague.

The function readIntervals is simple, but crucial in terms of scope for parallelism, as we shall see later. It reads through the input list of bytes, and splits it into sublists based on the occurrence of restart markers. A restart marker will be a single byte with value ff in hex, followed by a value from d0 up to d7. The *encoder* will 'pad' any byte values of ff naturally occurring in the compressed data with a single zero byte to ensure they are never confused with a restart marker. This means that readIntervals can safely split up the compressed data without any greater level of detail than simply examining individual byte values. As such, this task should be very fast.

```
readIntervals :: [Byte] -> [[Byte]]
```

The next function, decodeInterval, will take a list of compressed bytes that form a single interval, and return a list of totally decompressed MCUs that, when reconstructed, will form the corresponding section of the output image. The definition is as follows:

```
decodeInterval :: HeaderInfo -> [Byte] -> [MCU]
decodeInterval hdrInfo
 = map (transformMCU) . intervalToMCUs hdrInfo . bytesToBits
```

Here again we have a composition of three stages. Firstly, given that Huffman decoding works at the bit rather than byte level (due to the use of variable length codes), we employ bytesToBits to transform our input list of bytes into a list of bits. Next we apply intervalToMCUs

which should supply us with a list of MCUs, each, at this stage, containing untransformed coefficients. Finally we map `transformMCU`, such that each MCU is transformed from a list of matrices of coefficients to a list of matrices of samples (Y, C_b, and C_r values). The type of `intervalToMCUs` is as follows:

```
intervalToMCUs :: HeaderInfo -> [Bit] -> [MCU]
```

We shall have to brush somewhat briefly over the goings on inside this function due to lack of space. Suffice to say we shall have a repeated application of a function which reads in an MCU, and maintains the state of the predictors between calls. Reading an MCU is in turn a repeated application of a function which reads in units.

Let us return now to the function `transformMCU`. This takes an MCU, containing units of untransformed coefficients, and returns an MCU containing units of fully decoded sample data. It maps the function `transformUnit` to each unit in the MCU.

```
transformMCU :: HeaderInfo -> MCU -> MCU
transformMCU hdrInfo = map (transformUnit)
```

The `transformUnit` function performs the familiar stages of transforming an 8×8 unit of coefficients into an 8×8 unit of output sample values. Firstly it performs zig-zag reordering, then dequantization (making use of the appropriate quantization table in the `HeaderInfo` structure), and finally applies the inverse discrete cosine transform.

```
transformUnit :: HeaderInfo -> Unit -> Unit
transformUnit hdrInfo = idct. dequantize hdrInfo . zigzag
```

5. IMPLEMENTATION

The majority of interesting functionality in the specification is concealed within the function `decodelnterval`, upon which we shall concentrate in this section. Given that an MCU is a list of units, and the number of units per MCU can be derived from the header information, it should be straightforward to flatten a list of MCUs into units and vice verse. This can be achieved with the functions `unitsToMCUs` and `MCUsToUnits`. Thus, with a little simple program transformation, we can arrive at the following definition:

```
decodeInterval' hdrInfo
 = unitsToMCUs hdrInfo .
   map idct . map (dequantize hdrInfo) . map zigzag .
```

```
MCUsToUnits . intervalToMCUs hdrInfo . bytesToBits
```

We may find the following 'shortcut' useful:

```
intervalToUnits hdrInfo
 = MCUsToUnits . intervalToMCUs hdrInfo . bytesToBits
```

This compositional form is now well suited to process refinement. An overview of the definition of DECODEINTERVAL could therefore proceed as follows. Communication between intermediate stages of the process (and indeed the final output of the process) will be in the form of a stream of vectors. At each stage a whole unit (sixty four values) is communicated in parallel. We have:

```
macro proc DECODEINTERVAL (streamin,vectorout)
{
   StreamOfVectors (64,Int) vsmida, vsmidb, vsmidc, vsmidd;
   par
   {
      INTERVALTOUNITS (smid,vsmida);
      SMAP (vsmida,vsmidb,ZIGZAG);
      SMAP (vsmidb,vsmidc,DEQUANTIZE);
      SMAP (vsmidc,vsmidd,IDCT);
      UNITSTOMCUS (vsmidd,vectorout);
   }
}
```

Now we have a definition for DECODEINTERVAL, we can construct our overall refinement of decodeScan. Let us consider the three stages of decodeScan in turn. Firstly we have readIntervals. A process refinement of this function, READINTERVALS, should take a stream of bytes as input - it needs to process these sequentially. The output, a list of lists of bytes, can be produced as a vector - each interval can be processed independently. At the next stage, we map decodeInterval to each interval produced by readIntervals. As the input to this stage will be a vector, we shall choose VMAP to refine the map in the original specification.

We shall leave the output type of the compose image stage (which forms the output of the decoder as a whole) deliberately vague - we may want it in any one of several forms depending on the process that receives the data. Regardless of the exact structure of the output, the overall outline for the DECODESCAN process can proceed as in Figure 1. The implementation is depicted in Figure 2.

```
macro proc DECODESCAN (streamin,vectorout)
{
   VectorOfStreams (n,Byte) vectormida;
   StreamOfVectors (n,Byte) vectormidb;
   par
   {
      READINTERVALS (n,streamin,vectormida);
      VMAP(n,vectormida,vectormidb,DECODEINTERVAL);
      VFOLDR (n,vectormidb,vectorout,ADDINTERVAL);
   }
}
```

Figure 1. The DECODESCAN process.

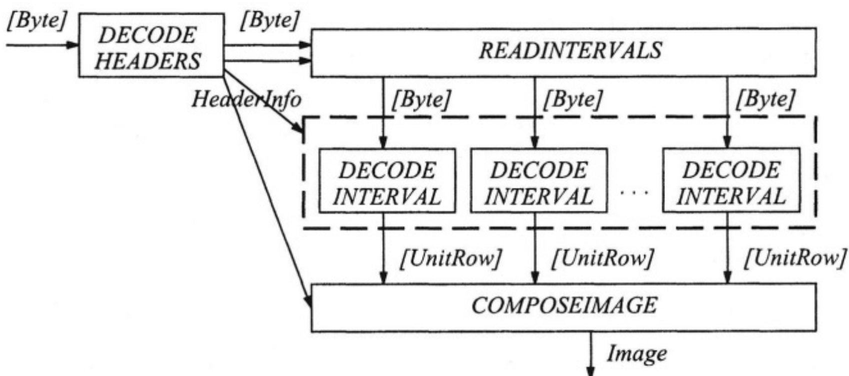

Figure 2. The JPEG decoder process network

6. CONCLUSION

We have presented a framework in which non-trivial algorithms can be specified in a clear, well structured environment, and then transformed formally, and in part mechanically, into an efficient behavioural implementation.

We have illustrated this with the development of a JPEG decoding algorithm, starting from a high level and intuitive specification in Haskell, and using this to derive a parallel Handel-C program that in turn can be compiled into a circuit design for an FPGA.

Given that the intervals (defined by the restart markers) in the compressed data are decoded independently of each other in parallel, our

implementation is scalable, and as we are required to deal with larger problem sizes (effectively higher resolution images) we simply need to add more processing elements. Effectively this means we should use an FPGA with more gates, or combine more than one FPGA together, and the resulting execution time should not be greatly increased. This assumes, of course, that higher resolution images will contain more restart markers.

It is important to point out that restart markers are optional in the official JPEG specification, and the benefits of the implementation presented here on an image encoded without restart markers would be somewhat limited. It is worth noting however, that several newer compression standards derived from JPEG (including most notably MPEG-2, the worldwide standard for digital television), have adopted a version of restart markers which are mandatory.

REFERENCES

[1] A.E. Abdallah, Derivation of Parallel Algorithms from Functional Specifications to CSP Processes, in Bernhard Möller, ed., *Mathematics of Program Construction,* LNCS **947**, (Springer Verlag, 1995) 67-96

[2] A. E. Abdallah, Functional Process Modelling, in K Hammond and G. Michealson (eds), *Research Directions in Parallel Functional Programming,* (Springer Verlag, October 1999). pp339-360.

[3] A. E. Abdallah and J. Hawkins, Calculational Design of Special Purpose Parallel Algorithms, in *Proceedings of 7th IEEE International Conference on Electronics, Circuits and Systems* (ICECS 2000), Lebanon, (IEEE, December 2000). pp261-267.

[4] R. S. Bird and P. Wadler, *Introduction to Functional Programming,* (Prentice-Hall, 1988).

[5] R. S. Bird *Introduction to Functional Programming Using Haskell,* (Prentice-Hall, 1998).

[6] M. I. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation,* in Research Monographs in Parallel and Distributed Computing, (Pitman 1989).

[7] J. Fokker, Functional Specification of the JPEG algorithm, and an Implementation for Free, in R.C. Veltkamp and E.H.Blake, (eds), *Programming Paradigms in Graphics, Proceedings of the Eurographics workshop in Maastricht,* The Netherlands, September 1995. (Wien, Springer 1995). pp. 102-120.

[8] *Handel-C Documentation,* Available from Celoxica (`http://www.celoxica.com/`).

[9] C. A. R. Hoare, *Communicating Sequential Processes.* (Prentice-Hall, 1985).

[10] International Standards Organisation, *Digital Compression and Coding of Continuous Still Tone Images.* Draft International Standard DIS 10918-1.

[11] G. Wallace, The JPEG Still Picture Compression Standard, in *Communications of the ACM,* 43, 4, 1991. Draft International Standard DIS 10918-1.