Generation of a Fast JPEG 2000 Encoder using SPIRAL

Hao Shen

Kongens Lyngby 2008
This thesis was prepared at department of Informatics Mathematical Modelling, Technical University of Denmark (DTU) in collaboration with department of Electrical and Computer Engineering, at Carnegie Mellon University (CMU), in partial fulfillment of the requirements for acquiring the Master degree in engineering. The thesis is advised by Christian W. Probst at DTU, Franz Franchetti and Markus Püschel at CMU.

The thesis deals with different levels of representation of the bit-plane coding algorithm and its transformation and optimization that exploit the structure of the algorithm to better match the hardware platform. The main focus is on extensions of high level language (Operator Language) used in SPIRAL and the presentation and generation EBCOT algorithm in the of the JPEG 2000 encoder. The code compiler is also extended to generate the optimized code.

Kongens Lyngby, August 2008

Hao Shen
Domain specific program generators reduce the need for tuning or rewriting performance libraries when new platforms are released.

This thesis extends SPIRAL’s framework to automatically generate the entropy coding part in JPEG 2000 encoder - EBCOT (Embedded Block Coding with Optimal Truncation), which is the most time-consuming component in the entire encoding process. The EBCOT algorithm is initially expressed in the high level representation of Operator Language (OL), which is an extension of the Signal Processing Language (SPL). Breakdown rules and manipulation rules are then applied to generate and optimize formula on the implementation level. The formula is then translated into C-like code and code level optimization is applied. Finally, the C-like code is optimized by the spiral code compiler and is unparsed into C code. For encoding JPEG 2000 images, parallelization is used due to the fact that blocks are independently encoded, this exploits the multi-core functionality of the most recent commercial processors. Vectorization is also used as an option to fully use the capabilities of short vector instruction set (Streaming SIMD Extensions) existing on the latest Intel Processors.

A special adapted version of EBCOT is used to replace the open source reference implementation JasPer and the encoding performance compares and even outperforms that of Intel Performance Primitive.
Acknowledgements

I want to thank all person who supported me to make this work possible.

In particular, I want to thank my advisor Christian W. Probst at DTU for his input and feedback. I also want to thank my advisors Franz Franchetti and Markus Püschel at CMU for giving me the chance of working on my thesis with the SPIRAL team in Pittsburgh and their advice and ideas throughout the project.

Additionally, I want to thank Frédéric de Mesmay and Yevgen Voronenko from the SPIRAL team for their essential help on the SPIRAL platform.

I would also like to thank Karl Sigurjónsson for his comments on the report.

I want to thank my family and especially my parents for always believing in me and their support in all the years.

Hao Shen
Contents

Preface i

Summary iii

Acknowledgements v

1 Introduction 1

2 Background 5

2.1 Hardware vs. Algorithm 5

2.2 Performance Library – Building and Tuning 6

2.3 JPEG 2000 Algorithm 8

2.4 Hardware Development Trend 8

2.5 Automatic Performance Tuning and SPIRAL 11

3 SPIRAL and Operator Language 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Signal Processing Language</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Breakdown Rule and Manipulation/Rewrite Rule</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Ruletree and Formula</td>
<td>19</td>
</tr>
<tr>
<td>3.4 Templates/Codegen and Code</td>
<td>20</td>
</tr>
<tr>
<td>3.5 Optimization</td>
<td>21</td>
</tr>
<tr>
<td>3.6 The Operator Language</td>
<td>22</td>
</tr>
<tr>
<td>4 JPEG 2000 Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>4.1 JPEG 2000 Encoding Process</td>
<td>27</td>
</tr>
<tr>
<td>5 Generation of JPEG 2000 Encoder</td>
<td>31</td>
</tr>
<tr>
<td>5.1 Approach</td>
<td>31</td>
</tr>
<tr>
<td>5.2 JPEG 2000 Reference and Performance Implementations</td>
<td>32</td>
</tr>
<tr>
<td>5.3 Runtime Profile for Jasper Implementation</td>
<td>33</td>
</tr>
<tr>
<td>6 Entropy Encoding in JPEG 2000</td>
<td>35</td>
</tr>
<tr>
<td>6.1 Tier-1 Coding</td>
<td>36</td>
</tr>
<tr>
<td>6.2 Tier-2 Coding</td>
<td>50</td>
</tr>
<tr>
<td>7 JPEG 2000 Entropy Coding in Operator Language</td>
<td>51</td>
</tr>
<tr>
<td>7.1 EBCOT</td>
<td>51</td>
</tr>
<tr>
<td>7.2 Operators Language and Extension for EBCOT</td>
<td>52</td>
</tr>
<tr>
<td>7.3 Prototype Implementation</td>
<td>56</td>
</tr>
<tr>
<td>7.4 Optimized Implementations</td>
<td>63</td>
</tr>
</tbody>
</table>
JPEG 2000 is a standard for image compression developed by the Joint Photographic Experts Group committee in the year 2000 and published as an ISO standard, ISO/IEC 15444-1. JPEG 2000 provides many advantages, such as better compression performance, multiple resolution representation and progressive transmission, just to name a few, over its predecessors. It is quite suitable for anything from scientific applications, such as medical imaging and geographic imagery to consumer applications such as digital camera imaging.

However, the major drawback of the JPEG 2000 standard compared to current widely used JPEG standard is that the coding algorithm is much more complex and the computational needs are much higher.

The complexity of the JPEG 2000 algorithm and its potential wide application demands a high performance library. However, the complexity of the algorithm itself also makes it a challenging task to map it to standard hardware efficiently.

A current trend in CPU development, due to the factor that the processor frequency cannot easily be increased, major vendors are instead providing multiple cores on a single chip to increase the overall performance of the processor. Another development in general purpose microprocessors is that the power of the short vector SIMD (single instruction, multiple data) extension has been greatly enhanced, to improve the performance of multi-media applications. However,
using these hardware features currently still needs programmers to explicitly
use the assembly or vendor APIs, to tune and adapt the algorithms towards the
new platforms. It is even harder to generally optimize the algorithms to take
advantage of the special processor capabilities.

Using compiler for automatic performance tuning would be a good solution
as the source code doesn’t have to be changed for each platform. However,
the main reason why normal compilers cannot automatically optimize for non-
trivial algorithms is that it doesn’t have the high level information needed to
match algorithms to the current hardware. Using a high-level language which
could easily map the algorithm to the features in the hardware will require a
code generator to generate lower level code for compilation. However, most code
generation systems are only for software engineering, and thus not designed for
performance.

The SPIRAL project developed by Carnegie Mellon University is an automatic
algorithm generation and optimization system, using high level language presen-
tation and mathematics algebra to transform and adapt algorithms to various
platforms. However, the domain of algorithms that SPIRAL can generate is
currently limited to only digital signal processing algorithms (transforms).

This thesis provides an extension of the newly developed language for SPIRAL-
the Operator Language. This extension is done so that the most consuming part
of the JPEG 2000 encoding algorithm - EBCOT can be expressed, generated
and optimized using to SPIRAL’s framework. The extension greatly extends
the capability of the Operator Language and the optimization capabilities of
the compiler, thus enabling the automatic performance tuning capabilities of
SPIRAL on a bigger set, and more general algorithms.

Synopsis

Chapter 1 introduces JPEG 2000 and the reason why a performance implementa-
tion of such algorithms is needed but difficult to implement. The performance
tuning system - SPIRAL is briefly introduced.

Chapter 2 discussed the hardware development trend, its implication on perform-
ance library implementation. The method of automatic performance tuning
is introduced with the example of SPIRAL.

In Chapter 3, the architecture of SPIRAL platform particularly the high level
language - Signal Processing Language and its extension Operator Language are
introduced.
Chapter 4 describes the encoding process of JPEG 2000 encoder algorithm and Chapter 5 describes the approach to generate the JPEG 2000 encoder.

Chapter 6 detailed describe the entropy coding algorithm and the algorithm that is going to be generated and tuned - EBCOT used in JPEG 2000.

Chapter 7 is the main chapter for the thesis, an extension of Operator Language is introduced to make it possible to express the EBCOT algorithm described in Chapter 6. New operator language operators, breakdown rules and rewrite rules are added, optimizations on the code level and algorithm level are discussed.

Chapter 8 shows the performance comparison between the reference implementation, a fast hand-tuned library implementation and the generated implementation.

The source code snippet generated by SPIRAL can be found in Appendix A.
2.1 Hardware vs. Algorithm

The Moore’s law [15] in hardware states the trend that the number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially, doubling approximately every 18 months. The law has been observed to be accurate for more than three decades, and the consequence is that current mainstream single processor workstation computers have a theoretical peak performance of more than 10 GigaFLOPS. However, it’s increasingly harder to harness the peak performance, except for the most simple computational tasks. This is due to the fact that modern computers are not just faster but vastly more complex. For example, nowadays a cache miss can be 10 - 100 times more expensive than a floating point multiplication. Generally speaking, the performance of numerical code now depends crucially on the number of processor cores and their layout, use of the platform’s memory hierarchy, register sets, available special instruction sets – in particular vector instructions - and other, often undocumented, microprocessor features.

Also, the most important changes in the current trend of hardware development is that parallelism has entered mainstream computing, due to the fact that frequency of a single processor can no longer easily be increased due to physical limitations, heat and power consumption. The performance increase is gained by adding multiple processor cores, tightly integrated into one chip. This can
be seen from recent introduction of Dual-Core, Quad-Core and even eight core processors by major vendors such as IBM and Intel. In addition, each core provides further parallelism through SIMD (single instruction, multiple data) short vector instructions that were already introduced a few years ago.

For high performance computation and consumer multi-media applications, there is constantly an infinite demand for optimization, but the ubiquitous and different forms of parallelism, specifically at the instruction level and multi-core level, will pose an enormous burden on developers of high-performance libraries. For example, most current software packages do not yet efficiently utilize the power of modern parallel computing systems. Even before, writing fast code was very difficult due to deep memory hierarchies and the general complexity of micro architectures that made performance platform-dependent and hard to predict. Further, computing platforms constantly change, which requires developers to permanently reimplement or retune the same functionality if highest performance is desired. This situation will worsen considerably as, in addition to the above problems, developer will have to produce code that takes advantage of different levels of parallelism, e.g., multi-threading and vectorization.

2.2 Performance Library – Building and Tuning

Writing high-performance libraries usually relies on a small group of experienced people, and the algorithms are usually tuned to a few specific platforms. The code may well possibly be written and tuned in assembly language in such a manner that it matches and utilizes what is offered most by the specific platforms such as instruction sets (especially vector instruction sets), memory hierarchy and cache access patterns. This restricts the complexity and scope of algorithms that can be successfully tuned because of the effort involved. These libraries have to be reimplemented and retuned every time a new platform comes out.

For performance tuning, a commonly used mechanism is to work on “representative short runs” or benchmarking runs. In other words, the scientific program is run with meaningful input for a short period of time. Based on the performance observed, the program is changed by some experienced people and executed again. This process repeats until the performance is acceptable. After that, the program is tuned and ready for production run. The “representative” or meaningful input is usually decided by the scientists who developed the scientific program or who have in-depth domain knowledge. This tuning process is usually time consuming and not cost-effective. On the other hand, due to the fact that these applications frequently have large computational demands and thus need to be run on large-scale parallel computers, even a small percentage improvement in the execution time will reduce the cost dramatically. Alternatively, with improved execution time, the program can also achieve better
results such as higher resolution, better precision or use a larger data set.

An important method that is commonly used in performance tuning is to adjust the configurations to achieve better performance. A configuration is a set of parameters such as data layout alignment. These configurations are possibly quite specific to the algorithm implementation to be tuned rather than general to the algorithm. Also, sometimes the parameters inside the configurations do not necessarily correspond to parameters in a specific library, thus a more “general” implementation is used for the tuning. This will have negative impact on the performance, as illustrated in the next paragraph.

Specialized algorithm implementation, which refers to implementations of an algorithm that have some fixed parameter (the transformation size for instance), most will have better performance than a general implementation. The main reason is that specialized implementations give more chance to apply algorithm transformation and optimizations that might not hold for the entire domain of an algorithm.

In principle, Compilers are an ideal solution to do automatic performance tuning since the source code does not need to be rewritten for each platform. However, high-performance library routines are carefully hand-tuned, frequently directly in assembly language, since today’s compiler often generate inefficient code even for simple problems. The reason is mainly because important performance improvement can only be attained by transformations and optimizations that are beyond the capability of today’s compilers or that reply on the algorithm information that is difficult to extract from a high-level language.

Thus, a natural choice to solve the automatic tuning problem, considering all of factors above, would be a system that acts like a compiler, that could automatically generate the specialized algorithm implementation from a high level language. This high level language will be able to describe and represent the algorithm in a way that it would easy to parameterize, possible to transform and map to different hardware features. These parameters and transforms acts as choices of implementations and the system can search from the alternatives and find the best for specific platform.

However, there are few high level code generation framework out there built for performance. And a few of the performance platform that uses code generation technique, such as FFTW (Fastest Fourier Transform in the West) or ATLAS (Automatically Tuned Linear Algebra Software), has very limited domain and not easily expandable to encompass a bigger domain of algorithm.
2.3 JPEG 2000 Algorithm

JPEG 2000 [14] is a wavelet-based image compression standard. It was created by the Joint Photographic Experts Group committee in the year 2000 with the intention of superseding their original discrete cosine transform-based JPEG standard (created about 1991).

The JPEG 2000 standard supports lossy and lossless compression of single-component (e.g., grayscale) and multi-component (e.g., color) imagery. In addition to this basic compression functionality, however, numerous other features are provided, including: 1) progressive recovery of an image by fidelity or resolution; 2) region of interest coding, whereby different parts of an image can be coded with differing fidelity; 3) random access to particular regions of an image without needing to decode the entire code stream; 4) a flexible file format with provisions for specifying opacity information and image sequences; and 5) good error resilience. Due to its excellent coding performance and many attractive features, JPEG 2000 has a very large potential application base. Some possible application areas include: image archiving, Internet, web browsing, document imaging, digital photography, medical imaging, remote sensing, and desktop publishing.

However, JPEG 2000 is much complicated than other traditionally used image compression algorithm and it has notably higher computational and memory demands for coding and decoding process, mainly due to the introduced coding block.

2.4 Hardware Development Trend

2.4.1 Multi Core

For a long time, frequency has been the mostly important factor for microprocessor performance. The reason is, if other things being equal, the more tick goes per second, the more work will get done. As the complexity of computer architecture grows, other factors such as cache structure, pipeline, special instruction etc, all became more and more important to the performance. Moreover, just a few years ago, the CPU clock rate reached its peak, for both CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer) processors. The reason why CPU frequency stalled is that it was hitting some of the physical limitations which will cause heat dissipation and data synchronization problems. Plus, it’s not worth the cost in terms of power consumption. A report by Intel [17] indicated that underclocking a core by 20 percent saves half of the power while only sacrificing just 13 percent of the performance.
The solution of major vendors to this problem is to combine two or more independent cores into a single package composed of a single integrated circuit. Cores in a multicore device may share a single high level cache or may own their own caches, Figure 2.1 shows a typical two core architecture with the cores own their separate level 1 cache while sharing a larger level 2 cache. This type of architecture provides a natural hardware parallelism but the performance gained by using multicore processor depends heavily on the problem, the algorithm used, as well as the implementation in software.

Symmetric Multiprocessing (SMP) support is built into most operating systems. But adjustments to existing software are required to maximize utilization of the computing resource provided by multi-core processors. That is because multi-threading is not transparent to the majority of programming languages. In order to use the multi-threading capabilities of the multi-core processors, the algorithms need to be transformed and adapted to the Process/Thread model, and implemented using the vendor or operating system dependent APIs. Examples of these APIs are Win Thread in Windows API, PThread, MPI, OpenMP etc. Mapping and adapting of algorithms is non-trivial and there are many factors that need to taken into consideration in this process, such as synchronization, data sharing and racing. In a word, it’s not easy to fully utilize the power of more and more complicated computer systems.

### 2.4.2 SIMD Technology

In computing, SIMD (Single Instruction, Multiple Data) is a technique employed to achieve data level parallelism, as in a vector processor. These vector instructions were first made popular in large-scale supercomputers. Smaller-scale SIMD
operations have now become widespread in personal computer hardware. Major vendors of general purpose microprocessors started with adding SIMD extensions to their instruction set architectures (ISA) to improve the performance for certain applications, especially the multi-media kernels. Although initially developed for the acceleration of multi-media applications, these extensions have the potential to speed up digital signal processing kernels as well.

All SIMD extensions are currently based on the packing of large registers with smaller 8-bit, 16-bit or 32-bit data types. Once data are packed into the larger register, operations are performed in parallel on the separate data items within the vector register. Currently, most integer, single-precision floating point and double-precision arithmetic and logical operations are supported.

Figure 2.2 shows an example of typical SIMD instruction, a pair of packed data operands are executed in parallel with the same operator.

Currently, application developers have three common methods for accessing the vector hardware within the system to accelerate their applications. They can invoke vendor supplied media processing libraries (such as the Intel Performance Primitive Library), use assembly language for those vector instructions to rewrite key portion of the applications, or code in high-level language and use vendor supplied macros that make available the functionality of the media-processing primitives through a simple function call like interface.
2.4.2.1 Algorithm Vectorization

Vectorizing compilers were developed for the architecture with vector computers. These compilers can automatically generate vector code from normal high-level language such as C. As the vectorizing compiler technology originates from completely different machines and new restrictions are found in the short vector SIMD extensions, the capabilities of these compilers are limited. Currently the best compiler on a general purpose processor platform with vector instruction set extension can only automatically vectorize the simplest arithmetic algorithms.

2.5 Automatic Performance Tuning and SPIRAL

SPIRAL [16] (www.spiral.net), developed at Carnegie Mellon University, is a program generation system focused mainly on digital signal processing transforms on both hardware and software. It automatically generates high-performance code tuned to a given platform by finding the domain-specific mathematical structure of the algorithm that best matches the computer’s micro-architecture. For a given transform, SPIRAL uses a domain-specific language to generate various different algorithms and to optimize them at a high level of abstraction using a rewriting system. Optimization includes simple unrolling and vectorization and/or parallelization for multi-core platforms. Search and learning techniques are used to guide this automatic exploration of algorithm and implementation space to find the fastest code. Experimental results in [16] show that the code generated by SPIRAL competes with, and sometimes outperforms, the best available human tuned transform library code.

The problem that Spiral is to solve can be stated as automatically generate and optimize software implementation for linear digital signal processing transforms that are tuned to a target hardware platform. For a specific transform, the aim is to find the implementation $I$ with the minimized cost (e.g. execution time) among all the possible implementations $\mathcal{I}$. The two sub-problems are how to generate the possible implementations and finding the implementation with minimized cost.

SPIRAL exploits the specific domain of linear DSP transforms by representing algorithms in a concise mathematical language called signal processing language (SPL), which uses only a few constructs. Further, it is possible to generate these SPL formulas recursively using a small set of rules to obtain a large formula space. These formulas, in turn, can be transformed into code.

SPIRAL’s architecture, shown in Figure 2.3, can be viewed as a solver for the optimization problem. Spiral uses breakdown rules and manipulation rules to
Figure 2.3: The program generator Spiral.
recursively generate the set of implementation for an algorithm. Breakdown rules specify how a transform can be computed by other transforms, usually in a smaller size. Manipulation rules turn one transform into another that is equivalent but possibly maps better to the platform’s hardware. These manipulations are performed on in the Algorithm Level and are all presented as SPL. In the Implementation Level, SPLs are translated into high level languages such as C or Verilog. Code Optimization is also done at this level. The generated code is then compiled into executable and performance is evaluated. The result of the evaluation is then fed back to the Search/Learning block to control the choice of algorithm and coding implementation options both at Algorithm Level and Implementation Level.

SPIRAL is a framework designed for automatic code generation and automatic performance tuning. However, as SPIRAL was previously designed for digital signal transforms, the high level language (SPL or its extension OL) cannot express the algorithm of those such as the coding algorithm in JPEG 2000. The main contribution of this thesis is to extend SPIRAL’s framework to enable it to encompass a broader range of algorithms. This is attained by adding new operators and compiler rules to the Operator Language so that it would describe the coding algorithm in JPEG 2000, designing of the rewriting system to generate and optimize the algorithm in a high level of abstraction, and a compiler to generate the actual code.
SPIRAL [16] is a research project developed by Carnegie Mellon University on automatic code generation, code optimization, and platform adaptation. The traditional domain of SPIRAL is numerical problems, namely digital signal processing algorithms, or more specifically, linear signal transforms. SPIRAL addresses the general problem: How to enable machines to automatically produce high quality code for a given platform? In other words, how can the processes that human experts use to produce highly optimized code be automated and possibly improved through the use of automated tools?

SPIRAL's solution formulates the problem of automatically generating optimal code as an optimization problem over the space of alternative algorithms and implementations of the same transform. To solve this optimization problem using an automated system, the mathematical structure of the algorithm domain is exploited. Specifically, SPIRAL uses a formal high level language and the transformation of the language to efficiently generate many potential alternative algorithms for a given transform and to translate them into code. Then, SPIRAL uses search and learning techniques to traverse the set of these alternative implementations for the same given transform to find the one that is best tuned to the desired platform while visiting only a small number of alternatives.
Figure 2.3 shows an overview how spiral generate, implement and optimize a transform and how each step correspond to SPIRAL’s architecture. A transform $T_n$ is presented by Signal Processing Language (SPL). It is generated recursively using breakdown rules and get a Formula for the transform, that corresponds to formula Generation at the Algorithm Level. Then manipulation rules are applied to the transform to get some equivalent formula. Rules and formula are also presented in SPL.

The formula is essentially a statement of combination of SPL blocks that can not be expanded any further called terminals. Each SPL terminal has a template of code that is used to generate the C-like code. Once a formula is chosen, it is fed into implementation level, and code for the entire transform is generated (Implementation). Then the Code Implementation block performs various standard and less standard optimizations at the code level, e.g., common sub expression elimination and code reordering for locality. These optimizations are necessary as standard compilers are often not efficient when used for automatically generated code, in particular, for large blocks of straightline code.

After formula generation and code generation, the third conceptual key block in SPIRAL is the evaluation level block, which is responsible for measuring the performance of the generated code and for feeding the result into the Search/Learning block. The evaluation level fulfils three main functions: 1) compilation of the source code into machine code; 2) optional verification of the generated code; and 3) measurement of performance of the generated code. The performance metric can be runtime of the compiled code, or it can be some other statistics about the number of arithmetic operations, the instruction count, the number of cache misses, etc. This information will enable the feedback loop to automatically explore algorithm and implementation alternatives. This feedback loop provides SPIRAL with the “intelligence” that produces very fast code.

### 3.1 Signal Processing Language

The SPL: Signal Processing Language is a language suitable to express products of structured sparse matrices using a small set of constructs and symbols. Table 3.1 provides a grammar for SPL in Backus-Naur form (BNF) as the disjoint union of different choices of rules (separated by vertical line “|”) to generate valid SPL expressions. Symbols marked by ⟨⟩ are non-terminal, ⟨spl⟩ is the initial non-terminal, and all the other symbols are terminals. The elements of SPL are called formulas. There are the following SPL constructs:

**Generic matrices.** SPL provides constructs to represent generic matrices, generic permutation matrices, and generic sparse matrices. However, most matrices occurring within transform have additional structures, these generic
matrices are rarely used except the diagonal matrices. There are written as 
\[ \text{diag}(a_0, ..., a_{n-1}) \], where the arguments list contains the diagonal entries of the matrix.

**Symbols.** Frequent occurring classes of matrices are represented by parametrised symbols. Examples include the \( n \times n \) identity matrix \( I_n \), the matrix \( J_n \) is obtained from the identity matrix by reversing the columns (or rows), \( n \times n \) zero matrix \( O_n \). The stride permutation matrix \( L^n_k \), which reads the input at stride \( k \) and stores it at stride 1, is defined by its corresponding permutation:

\[
L^n_k : i(n/k) + j \to jk + i, 0 \leq i \leq k, 0 \leq j \leq n/k;
\]

Another example is the butterfly matrix \( F_2 \), which is equal to the \( 2 \times 2 \) DFT matrix, but not considered a transform, as transforms are generally considered non-terminal.

\[
F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}
\]

**Transforms.** SPL expresses various transforms such as \( \text{DFT}_n \), \( \text{DCT-2}_n \), and \( \text{Filt}_n(h[z]) \). In particular, only those formulas that do not contain transforms can be translated into code. Also, the set of transforms and the set of symbols available in SPL are user extensible.

**Matrix constructs.** SPL constructs can be used to form structured matrices from a set of given SPL matrices. Examples include the product of matrices \( AB \) (or \( A \cdot B \)), the direct sum \( \oplus \) and the tensor or Kronecker product \( \otimes \) of two matrices \( A \) and \( B \), defined, respectively, by

\[
A \oplus B = \begin{bmatrix} A \\ B \end{bmatrix}
\]

\[
A \otimes B = [a_{k,i}B], \text{where} A = [a_{k,i}],
\]

### 3.2 Breakdown Rule and Manipulation/Rewrite Rule

In Spiral, an algorithm for a transform \( T_n \) is generated recursively using breakdown rules and manipulation rules. Breakdown rules are recursions for transforms, i.e., specify how to compute a transform from other transforms of the same or a different type and of the same or smaller size. Spiral will use a set of rules to recursively expand a transform \( T_n \) until no further expansion is possible.
\[
(spl) := \langle \text{generic} \rangle \langle \text{symbol} \rangle \langle \text{transform} \rangle
\]
\[
(spl) \cdot (spl)
\]
\[
(spl) \oplus (spl)
\]
\[
(spl) \otimes (spl)
\]
\[
\ldots
\]
\[
\langle \text{generic} \rangle := \text{diag}(a_0, \ldots, a_{n-1})\ldots
\]
\[
\langle \text{symbol} \rangle := I_n|J_n|L^n_k|F_2|\ldots
\]
\[
\langle \text{transform} \rangle := \text{DFT}_n|\text{DCT-2}_n|\text{WHT}_n|\text{Filt}_n(h[z])\ldots
\]

Table 3.1: Definition of the most Important SPL constructs in BNF

to obtain a completely expanded formula \( T_n \in \mathcal{F} \). This formula specifies one algorithm for \( T_n \).

Finally, the terminal breakdown rules are used to terminate the base cases, which usually means transforms of the size 2. The right hand side of the terminal breakdown rule does not contain any transforms. Examples include for the trigonometric transforms

\[
\text{DFT}_2 \rightarrow F_2,
\]
\[
\text{DCT-2}_2 \rightarrow \text{diag}(1, 1/\sqrt{2})F_2
\]
\[
\text{DCT-4}_4 \rightarrow J_2R_{13\pi/8}
\]

A manipulation rule is a matrix equation in which both sides are SPL formulas, neither of which contains any transforms. These rules are used to manipulate the structure of an SPL formula that has been fully expanded using breakdown rules. Examples involving the tensor product include

\[
A_m \otimes B_m \rightarrow (A_m \otimes I_n)(B_n \otimes I_m)
\]
\[
(B_n \otimes A_m) \rightarrow L_n^{mn}(A_m \otimes B_n)L_m^{mn}
\]

The first rule is referred to as the multiplicative property of the tensor product.

Manipulation rules for the stride permutation include the following

\[
(L_m^{mn})^{-1} \rightarrow L_m^{mn}
\]
\[
L_m^{kn}L_n^{km} \rightarrow L_n^{km}L_m^{kn} \rightarrow L_m^{kn}L_n^{km}
\]
\[
L_n^{km} \rightarrow (L_n^{kn} \otimes I_m)(I_k \otimes L_m^{mn})
\]
\[
L_k^{km} \rightarrow (I_k \otimes L_m^{mn})(L_n^{kn} \otimes I_m)
\]
3.3 Ruletree and Formula

Recursively applying rules to a given transform to obtain a fully expanded formula leads conceptually to a tree, which in SPIRAL is called ruletree. Each node of the tree contains the transform and the rule applied at this node. As a simple example, consider the $\text{DCT-2}_4$, expanded first using the rule

$$\text{DCT-2}_4 \rightarrow L_2^4(\text{DCT-2}_2 \oplus \text{DCT-4})(F_2 \otimes I_2)(I_2 \oplus J_2)$$  \hspace{1cm} (3.1)

and then completely expanded using the base case rule mentioned above.

A ruletree is always expected to be expanded. A ruletree clearly shows which are used to expand the transform and, thus, uniquely defines an algorithm to compute the transform.

Ruletrees are a convenient representation of SPL formulas: they keep the relevant information for creating the formula, they are storage efficient, and they can be manipulated easily, e.g., by changing the expansion of a subtree. The ability to manipulate ruletrees easily gives much convenience for search.

Expanding a ruletree by recursively applying the specified rules top-down, yields a completely expanded (SPL) formula, or simply a formula. Both a ruletree and the formula specify the same fast algorithm for the transform, but in a different form. The information about the intermediate expressions of the transform is lost in the formula, but the formula captures the structure and the dataflow of the computation, and hence can be mapped to code. As an example, the completely expanded formula corresponding to $\text{DCT-2}_4$ is given by

$$\text{DCT-2}_4 \rightarrow L_2^4(\text{diag}(1, 1/\sqrt{2})F_2 \oplus J_2 R_{13\pi/8})(F_2 \otimes I_2)(I_2 \oplus J_2)$$  \hspace{1cm} (3.2)

The formula (3.1) can not be translated into code in SPIRAL because it is not fully expanded: its right hand side contains the transforms $\text{DCT-2}_2$ and $\text{DCT-4}_2$, which are non-terminals. In contrast, (3.2) is a fully expanded formula since it expresses $\text{DCT-2}_4$ exclusively in terms of terminal SPL constructs. A fully expanded formula can be translated into code.

The above framework defines a formal language that is a subset of SPL. The non-terminal symbols are the transforms, the rules are the breakdown rules available in SPIRAL, and the generated language consists of those formulas that are fast algorithms for the transforms.

Alternatively, we can regard this framework as a term rewriting system. The terms are the formulas, the variables are the transform sizes (or, more general, the transform parameters), the constants are all other SPL constructs, and the
rules the breakdown rules. The transform algorithms are those formulas in normal form.

In theory, the termination of the breakdown rules cannot be guaranteed as there are potential infinite loops. In practice, termination is ensured by only including translations rules that translate transforms of higher complexity into transforms of lower complexity.

3.4 Templates/Codegen and Code

The translation of SPL formulas to code is defined through templates. A template consists of a parameterized formula construct $A$, a set of conditions on the formula parameters, and a C-like code fragment. Templates serve four main purposes in SPRIAL: 1) they specify how to translate formulas into code; 2) they are a tool for experimenting with different ways of mapping a formula into code; 3) they enable the extension of SPL with additional constructs that may be needed to express new DSP algorithms or transforms not yet included in SPRIAL; and 4) they facilitate extending the SPL compiler to generate special code types such as code with vector instructions.

Each template is written as a separate function implementing a parameterized SPL construct with its own scope of variables. However, when incorporated into the generated code, the variables local to different templates are given unique names to disambiguate them and to incorporate them into one common namespace. The template code is specialized by substituting all of the template parameters (e.g., size and str in $I_{str}^{size}$ by their respective values).

Currently SPRIAL can generate code for both instantiations and parameterized libraries, this thesis only focus on algorithm instantiations. Take transforms as example, the transform size and other parameters are assumed to be fixed in the formula generation process. This makes the specialization of the initial code generated from the formula straightforward. Different output languages such C, Fortran, C++ and Java are fully or partially supported, and in this thesis the implementation is available in C.

**Standard Code Generation** The SPL compiler translates a given SPL program describing a formula into C code. The first stage of the compiler traverses the SPL expression tree top-down, recursively matches subtrees with templates, and generates by specializing the template parameters with the values obtained from the formula.

Next, based on the local unrolling tags and the global unrolling threshold, the compiler identifies loops that should be unrolled and marks them accordingly in
the intermediate representation. Loops marked for unrolling are fully unrolled. A reasonably large degree of unrolling is usually very beneficial, as it creates many opportunities for optimizations.

3.5 Optimization

Code level optimizations are applied to the unrolled code. The SPIRAL compiler can do the following optimization to the code:

**Static single assignment form** All of the optimizations considered are scalar optimizations that operate on code converted to static single assignment form, in which each scalar variable is assigned only once to simplify the required analysis.

**Array scalarization** C compilers are very conservative when dealing with array references. For example, the loop unrolling stage can produce many array references with constant indices. During array scalarization, all such occurrences are replaced by scalar temporary variables, which will be potentially placed in registers.

**Algebraic simplification** This part of the optimizer performs constant folding and canonicalization, which support the efforts of other optimization passes.

Constants are canonicalized by converting them to be non-negative and by using unary negation where necessary. Expressions are canonicalized similarly by pulling unary negation as far out as possible. For example, $-x - y$ is translated to $-(x + y)$, and $-(x) \ast y$ becomes $-(x \ast y)$. Unary operators will usually combine with additive operators in the surrounding context and disappear through simplification.

**Copy propagation** Copy propagation replaces occurrences of the variable on the left hand side of a given “simple” assignment statement with the right hand side of that assignment, if the right hand side is either a constant, a scalar, or a unary negation of a scalar or a constant.

Recall that unary negation expressions are often created during algebraic simplification due to canonicalization. Copy propagation will move them so that they can combine with additive operators in the new context during further algebraic simplification.

**Common subexpression elimination** Common subexpression elimination tries to discover multiple occurrences of the same expression; it makes sure that these are computed only once. SPIRAL can optionally be configured to treat subscripted array references as expressions and, therefore, as eligible for
elimination.

**Optimization Strategy** The different optimizations described above have mutually beneficial relationships. For instance, algebraic simplification can bolster copy propagation, and copy propagation can then create new opportunities for algebraic simplifications. Alternating between these two optimization passes, the code will eventually reach a fixed point, where it is changed no further. Different optimization strategies can have different impact on the final performance of the code thus each algorithm can have its own compiler strategy.

SPIRAL’s implementation strategy is to loop over these different optimization passes in the manner prescribed, and to terminate once an entire iteration fails to change the code.

### 3.6 The Operator Language

SPIRAL’s approach for algorithm generation and optimization is based on matrix algebra, which restrict that every construct is linear. This gives the advantage of rewrite but hinges on the inherent structure of transforms and their algorithms.

The Operator Language (OL) is a super set of Signal Processing Language (SPL) that enables the extension beyond transforms by viewing matrices as operators that take an input vector $x$ to produce an output vector $y$. These operators are in addition linear. The product of matrices is now equivalent to the composition “$\circ$” of operators and the tensor product of matrices becomes a tensor product of operators. The transform breakdown rule

$$
\text{DFT}_n \rightarrow (\text{DFT}_k \otimes I_m) T^n_m (I_k \otimes \text{DFT}_m) L^n_k, \quad n = km
$$

now becomes a structured decomposition of the operator DFT into other operators:

$$
\text{DFT}_n \rightarrow (\text{DFT}_k \otimes I_m) \circ T^n_m \circ (I_k \otimes \text{DFT}_m) \circ L^n_k, \quad n = km
$$

The Operator Language has two elements: *operators* and *operations*. The definitions followed are all over the complex numbers. A generalization to other domains (reals, integers) is straightforward. Vectors in are $\mathbb{C}^n$ denoted with $x, y, x_i, y_j,$ etc. Sometimes, the vector is a matrix, stored linearized in memory. In this case upper-case letters are used as $A$. The $n$th component of the vector $x$ is denoted with $x_n$. 
### Table 3.2: Base operators definitions.

<table>
<thead>
<tr>
<th>name</th>
<th>definition</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear transform</td>
<td>$M : \mathbb{R}^n \to \mathbb{R}^m; x \mapsto Mx$</td>
<td>linear transform defined by an $m \times n$ matrix</td>
</tr>
<tr>
<td>stride</td>
<td>$L_{nm} : \mathbb{R}^{mn} \to \mathbb{R}^{mn}; M \mapsto M^T$</td>
<td>transposition of a $m \times n$ matrix</td>
</tr>
<tr>
<td>vector sum</td>
<td>$\Sigma_n : \mathbb{R}^n \to \mathbb{R}; x \mapsto \sum_{i=0}^{n-1} x_i$</td>
<td>returns the sum of all components</td>
</tr>
<tr>
<td>point-wise mul-</td>
<td>$(\cdot)_{m \times n \rightarrow m \times m} : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m$</td>
<td>multiplies components of $x$ and $y$</td>
</tr>
<tr>
<td>point-wise mul-</td>
<td>$(\oplus)_{m \times n \rightarrow m \times n} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m$</td>
<td>places $x$ and $y$ side by side</td>
</tr>
<tr>
<td>concatenation</td>
<td>$(\otimes)_{m \times n \rightarrow m \times n} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m$</td>
<td>multiplies each component of $x$ with $y$</td>
</tr>
<tr>
<td>Kronecker product</td>
<td>$\mathbb{R}^{mn}; (x, y) \mapsto x_0y \oplus \cdots \oplus x_{m-1}y$</td>
<td></td>
</tr>
</tbody>
</table>

**Operators** An operator of arity $(r, s)$ maps $r$ vectors to $s$ vectors. Formally,

$$A : \mathbb{C}^{n_0} \times \cdots \times \mathbb{C}^{n_{k-1}} \to \mathbb{C}^{N_0} \times \cdots \times \mathbb{C}^{N_{l-1}}$$

$$(x_0, x_1, \ldots, x_{k-1}) \to (y_0, y_1, \ldots, y_{l-1})$$

This definition includes transforms but also many other functions such as matrix-matrix multiplication ($n = 2, N = 1$).

The operator $A$ has the arity $(k, l)$ and the signature $((n_0, \ldots, n_{k-1}), (N_0, \ldots, N_{l-1}))$ with $(n_0, \ldots, n_{k-1})$ called domain and $(N_0, \ldots, N_{l-1})$ called range. Linearity is not required.

Any linear transform is an operator with arity $(1, 1)$. For example, the DFT$_n : x \to y = $ DFT$_n x$ is defined as formula (3.3) and has signature $((n), (n))$.

The signature of an operator $A$ can also be expressed explicitly, by writing, for example, $A_{k \times m \rightarrow N}$ if the signature is $((k, m), (N))$. Operators with the same domain and range can be expressed in a simpler way, for example, $M_n$ for $M_{n \rightarrow n}$ and $M_{m \times n}$ for $M_{m \times n \rightarrow m \times n}$.

**Operations** Operations or higher-order operators are functions on operators.

The composition of $A$ and $B$ is defined if the domain of $B$ is equal to the range of $A$:

$$(B \circ A)(x, y) = B(A(x, y))$$

The Cartesian product of two operators is the operation that “selects” arguments and “pairs” results. For example,

$$(A \times B)(x, y, z, t) = A(x, y) \times B(z, t)$$
Operator formulas. The first two sections in Table 3.3 introduce operations on vectors and tuples of vectors (only pairs are shown for simplicity). These are standard in mathematics and computer science. The third section shows the formalism we use to construct more complex, structured operators using ·, ⊕, ×, ◦ and ⊗. Among those, · and × have no equivalent for linear transforms. We restrict the definitions to arities (2, 1) and (2, 2); the generalization to arbitrary arities is straightforward. These definitions borrow ideas from multilinear algebra, but use it for not necessarily multilinear operators.

Identity operator. Here are the definitions of all identity operators up to arity (2, 2); identity operators with higher arities are defined analogously.

\[
\begin{align*}
I_n &: x \rightarrow x \\
I_{m \times n} &: (x, y) \rightarrow (x, y)
\end{align*}
\]

\[
\begin{align*}
I_{m \times n \rightarrow mn} &: (x, y) \rightarrow xy \\
I_{m \times n \rightarrow mn \times mn} &: (x \otimes y, x \otimes y)
\end{align*}
\]

OL Rewriting. The general rewriting rules for SPL are certainly still valid for OL where they can be applied to the SPL subset. As OL introduces new operations, there are extra rewriting rules, the most general ones are shown in Table 3.4.

Similar to SPL rewriting, rewrite rules can be specified for specific algorithms and with consideration with the target platform.

OL Grammar Definition. Table 3.5 shows the grammar for OL in BNF, introducing the new operations ◦ and ×. All the SPL constructs are seen as operators that are linear and whose input and output is single vector.
<table>
<thead>
<tr>
<th>name</th>
<th>definition</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>$\mathbf{x} + \mathbf{y} = (x_0 + y_0, \ldots, x_{m-1} + y_{m-1})$</td>
<td>usual vector addition</td>
</tr>
<tr>
<td>multiplication</td>
<td>$\mathbf{x} \cdot \mathbf{y} = (x_0y_0, \ldots, x_{m-1}y_{m-1})$</td>
<td>component-wise vector multiplication</td>
</tr>
<tr>
<td>direct sum</td>
<td>$\mathbf{x} \oplus (x_0, \ldots, x_{m-1}, y_0, \ldots, y_{n-1})\mathbf{y}$</td>
<td>glues the vectors together</td>
</tr>
<tr>
<td>Cartesian product</td>
<td>$\mathbf{x} \times \mathbf{y} = (\mathbf{x}, \mathbf{y})$</td>
<td>pairs vectors into tuple</td>
</tr>
<tr>
<td>tensor product</td>
<td>$\mathbf{x} \otimes \mathbf{y} = \mathbf{x}<em>0\mathbf{y} + \ldots + \mathbf{x}</em>{n-1}\mathbf{y}$</td>
<td>multiplies $\mathbf{y}$ with each component of $\mathbf{x}$</td>
</tr>
<tr>
<td>addition</td>
<td>$(\mathbf{x}, \mathbf{y}) + (\mathbf{u}, \mathbf{v}) = (\mathbf{x} + \mathbf{u}, \mathbf{y} + \mathbf{v})$</td>
<td>component-wise addition</td>
</tr>
<tr>
<td>multiplication</td>
<td>$(\mathbf{x}, \mathbf{y}) \cdot (\mathbf{u}, \mathbf{v}) = (\mathbf{x} \cdot \mathbf{u}, \mathbf{y} \cdot \mathbf{u})$</td>
<td>component-wise multiplication</td>
</tr>
<tr>
<td>direct sum</td>
<td>$(\mathbf{x}, \mathbf{y}) \oplus (\mathbf{u}, \mathbf{v}) = (\mathbf{x} \oplus \mathbf{u}, \mathbf{y} \oplus \mathbf{v})$</td>
<td>component-wise direct sum</td>
</tr>
<tr>
<td>Cartesian product</td>
<td>$(\mathbf{x}, \mathbf{y}) \times (\mathbf{u}, \mathbf{v}) = (\mathbf{x}, \mathbf{y}, \mathbf{u}, \mathbf{v})$</td>
<td>glues both pairs together</td>
</tr>
<tr>
<td>tensor product</td>
<td>$(\mathbf{x}, \mathbf{y}) \otimes (\mathbf{u}, \mathbf{v}) = (\mathbf{x} \otimes \mathbf{u}, \mathbf{y} \otimes \mathbf{v})$</td>
<td>component-wise tensor product</td>
</tr>
<tr>
<td>addition</td>
<td>$(\mathbf{M} + \mathbf{N})(\mathbf{x}, \mathbf{y}) = \mathbf{M}(\mathbf{x}, \mathbf{y}) + \mathbf{N}(\mathbf{x}, \mathbf{y})$</td>
<td>component-wise addition of the result</td>
</tr>
<tr>
<td>multiplication</td>
<td>$(\mathbf{M} \cdot \mathbf{N})(\mathbf{x}, \mathbf{y}) = \mathbf{M}(\mathbf{x}, \mathbf{y}) \cdot \mathbf{N}(\mathbf{x}, \mathbf{y})$</td>
<td>component-wise multiplication of the result</td>
</tr>
<tr>
<td>direct sum</td>
<td>$(\mathbf{M} \oplus \mathbf{N})(\mathbf{x} \oplus \mathbf{u}, \mathbf{y} \oplus \mathbf{v}) = \mathbf{M}(\mathbf{x}, \mathbf{y}) \cdot \mathbf{N}(\mathbf{u}, \mathbf{v})$</td>
<td>break argument and glue result</td>
</tr>
<tr>
<td>Cartesian product</td>
<td>$(\mathbf{M} \times \mathbf{N})(\mathbf{x}, \mathbf{y}, \mathbf{u}, \mathbf{v}) = \mathbf{M}(\mathbf{x}, \mathbf{y}) \cdot \mathbf{N}(\mathbf{u}, \mathbf{v})$</td>
<td>selects arguments and pairs result</td>
</tr>
<tr>
<td>composition</td>
<td>$(\mathbf{M} \circ \mathbf{N})(\mathbf{x}, \mathbf{y}) = \mathbf{M}(\mathbf{N}(\mathbf{x}, \mathbf{y}))$</td>
<td>usual composition</td>
</tr>
<tr>
<td>iterative composition</td>
<td>$(\prod_{i=0}^{n-1} \mathbf{M}_i)(\mathbf{x}, \mathbf{y}) = (\mathbf{M}<em>0 \circ \ldots \circ \mathbf{M}</em>{n-1})(\mathbf{x}, \mathbf{y})$</td>
<td>compose parameterized operators</td>
</tr>
</tbody>
</table>

Table 3.3: Operations definitions for vectors, pairs of vectors, and for operators. $\mathbf{M}$ and $\mathbf{N}$ have arity $(2,1)$ or $(2,2)$, an all $\mathbf{M}_i$ have the same signature $((m, n), (m, n))$.
\[(A \times B) \circ (C \times D) = (A \circ C) \times (B \circ D)\]
\[(\otimes)_{p \times q \to pq} A = (\otimes)_{p \times 1 \to p} \otimes (\otimes)_{1 \times q \to q} \otimes A\]
\[(\otimes)_{p \times q \to pq} \otimes A_{m \times n \to k} = (L^p_{pq} \otimes I^k) \circ ((\otimes)_{1 \times q \to q} \otimes (\otimes)_{p \times 1 \to p} \otimes A)\]
\[(\otimes)_{p \times q \to pq} \otimes A_{m \times n \to k} = L^{pk}_{pq} \circ (A_{m \times n \to k} \otimes (\otimes)_{p \times q \to pq}) \circ (L^m_{pq} \times L^q_{pq})\]
\[(\cdot)_{p \times p \to p} \otimes A_{m \times n \to k} = L^p_{pk} \circ (A_{m \times n \to k} \otimes (\cdot)_{p \times p \to p}) \circ (L^m_{pq} \times L^q_{pq})\]

\[L^{k_{mn}} = (L^k_{mn} \otimes I^m)(I_k \otimes L^m_{mn}) = L^{kn}_{mn} L^{kmn}\]

\[L^{k_{m}} = (I_k \otimes L^m_{mn})(L^k_{mn} \otimes I_m) = L^{kn}_{mn} L^{kmn}\]

\[A \otimes B = (A \otimes I_m)(I_n \otimes B) = (I_n \otimes B)(A \otimes I_m), A \text{ and } B \text{ matrices}\]

**Table 3.4:** OL rewriting rules.

\[
\langle \text{ol} \rangle := \langle \text{ol} \rangle \circ \langle \text{ol} \rangle \\
\langle \text{ol} \rangle \times \langle \text{ol} \rangle \\
\langle \text{spl} \rangle \\

...\]

\[
\langle \text{spl} \rangle := \langle \text{generic} \rangle | \langle \text{symbol} \rangle | \langle \text{transform} \rangle \\
\langle \text{spl} \rangle \cdot \langle \text{spl} \rangle \\
\langle \text{spl} \rangle \oplus \langle \text{spl} \rangle \\
\langle \text{spl} \rangle \otimes \langle \text{spl} \rangle \\

...\]

\[
\langle \text{generic} \rangle := \text{diag}(a_0, ..., a_{n-1})... \\
\langle \text{symbol} \rangle := I_n | J_n | L^2_k | F_2... \\
\langle \text{transform} \rangle := \text{DFT}_n | \text{DCT-2}_n | \text{WHT}_n | \text{Filt}_n(h[z])... \\
\]

**Table 3.5:** Definition of the most Important OL constructs in BNF
Chapter 4

JPEG 2000 Algorithm

4.1 JPEG 2000 Encoding Process

The JPEG 2000 compression standard is composed of the stages shown in the flow graph in Figure 4.1. Here only a high level explanation of each stage is provided, in corresponds to the standard defined in ISO/IEC 15444-1 [14].

The first stage of JPEG 2000 encoding is pre-processing. Per-processing actually contains three sub-stages.

The image to be encoded might be larger than the amount of memory available to the encoder. To solve this problem, JPEG 2000 allows for an optional tiling. In tiling, the input image is partitioned into rectangular and non-overlapping tiles of equal size (except possibly those on the border). Each tile is independently compressed using its set of compression parameters. This is defined in Data ordering (Annex B). Also, an image is divided of one ore more components, and each component consists of a two dimensional array of samples representing the luminosity of the component of the point. These samples are presented by integers, either signed or unsigned, and can have up to 38 bit per sample.

The following two blocks are described in DC level shifting and component transformations (Annex G). As JPEG 2000 uses high-pass filtering,
it expects its input sample data to have a nominal dynamic range centered at about zero. The DC level shifting ensure that this expectation is met. If the original $B$-bit image sample values are unsigned quantities, an offset of $-2^{B-1}$ is added so that the samples have a signed representation in the range of $-2^{B-1} \leq x[n] \leq 2^{B-1}$. If the data is signed, no adjustment is performed.

Color images are commonly represented in RGB format, in which three component planes are used, each describes the luminosity in red, green and blue. As the discrete wavelet transformation is applied to each component, these components should be as independent as possible. Since the color space of YC_rC_b are statistically less dependently than RGB color space, JPEG 2000 introduced one reversible and another irreversible component transform, used for lossless and lossy encoding respectively. These component transformation are basically just matrix transformations.

JPEG 2000 uses discrete wavelet transformation (DWT) to decompose each image tile into subbands shown in Figure 4.2, defined in the standard **Discrete wavelet transformation of tile components (Annex F)**. DWT is performed by filtering on each row and column of the pre-processed image tile with a high-pass and a low-pass filter. The sample is firstly symmetrically extended to ensure that data beyond edge is possible. After the DWT, the result of each filtering is downsampled by 2 so that the sample rate remain the same as the input.
In JPEG 2000, multiple stages of DWT are performed, resulting in multiple image resolution. In Figure 4.2 the Forward discrete wavelet transformation translate DC-level shifted tile component samples $I(x,y)$ into a set of sub-bands with coefficients $a_b(u_b,v_b)$, which depends on the paramter $N_L$, representing a number of iteration (number of decomposition level), create $(3 \times N_L) + 1$ sub-bands (LL, HL, LH, HH). The four quadrants are defined as:

- **LL**: low subbands for row and column filtering
- **HL**: high subbands for row filtering and low subbands for column filtering
- **LH**: low subbands for row filtering and high subbands for column filtering
- **HH**: high subbands for row and column filtering

In the **Quantization (Annex E)** stage after the Forward Wavelet Transform, each of the transform coefficients $a_b(u,v)$ of the sub-band $b$ is quantized to the value $q_b(u,v)$ according to the following equation:

$$q_b(u,v) = \text{sign}(a_b(u,v)) \left\lfloor \frac{\text{sign}(a_b(u,v))}{\Delta_b} \right\rfloor$$

Where $\Delta_b$ is the quantization step size.

The final block in the compression stage is the entropy coder. It is composed of Tier-1 and Tier-2. Tier-1 can be further broken down to EBCOT (EBCOT stands for Embedded Block Coding with Optimal Truncation) and MQ-Coder. Tier-2 is responsible for efficiently representing layers and block summary information for each code-block, including the code stream generated by Tier-1. It is also been defined in **Data ordering (Annex B)**.
Coefficient bit modeling (Annex D) is the part which describes how the coefficients are arranged into code-blocks, bit-planes and coding passes.

The coefficients are associated with different sub-bands arising from the transform applied. These coefficients are then arranged into rectangular blocks within each sub-band, called code-blocks. These code-blocks are then coded a bit-plane at a time starting from the most significant bit-plane with a non-zero element to the least significant bit-plane. For each bit-plane in a code-block, a special code-block scan pattern is used for each of three coding passes. Each coefficient bit in the bit-plane is coded in only one of the three coding passes. The coding passes are called significance propagation, magnitude refinement, and cleanup. For each pass contexts are created which are provided to the arithmetic coder, CX, along with the bit stream, CD.

Arithmetic coding (Annex C) defines the lossless arithmetic entropy coding. In this part, the list of context (CX) and decision (D) pair are processed to produce compressed data output. Both CX and D are provided by the model unit defined in Annex D, CX selects the probability estimate to use during the coding of D.

Arithmetic coding is a variable-length source encoding technique, the sequence of input symbols are represented by an interval of real numbers between 0 and 1. The encoded result is actually an interval, and the longer the input is, the smaller the interval it gets. The entire sequence of input can be decoded by any number in that interval.

What differs most from arithmetic coding is that binary arithmetic coding output a sequence of 0s or 1s as the result. Binary arithmetic coding scales the interval when it’s getting too small and this solves the problem of the precision of the final result that could happen to arithmetic coding. The advantage of the binary arithmetic coding is that it is an incremental encoding and decoding process, that is, the encoder need not wait till the end of the encoding of the last symbol of the message before it can output the encoded data.

QM-coder is the adaptive binary arithmetic coding algorithm. It follows the same principle of arithmetic coding but designed for simplicity and speed. The input can be only a stream of 0s and 1s, it maps the input bits into more probable symbol and less probable symbol. It would do the conditional exchange of the more probable symbol and less probable symbol is some conditions are met, this way, the output stream can be maintained shorter.

The MQ-coder used in JPEG 2000 standard is a variation of the QM-coder.
Chapter 5

Generation of JPEG 2000 Encoder

5.1 Approach

JPEG 2000 is a very complicated standard, and is defined as standard in ISO 1.29.15444 Part 1 Coding System (ISO/IEC FCD15444-1) [14]. As the aim is to determine whether automatic algorithm generation and performance tuning can generate a good performance library for algorithms such as JPEG 2000, it’s sufficiently enough to optimize the most time-consuming part of the JPEG 2000 encoding algorithm. The maximum expected improvement of the overall system is a bit limited according to Amdahl’s law [1], still we could get considerable performance gain by optimizing the coding algorithm in JPEG 2000, which is shown to take more than half of the encoding time through early experiment. Also, generating time-consuming code blocks could be useful in terms of making the library flexible and more reusable.

JasPer is an open source reference implementation of JPEG 2000 written in C, and is available on multiple platforms. It serves the base for the generated code in the terms of that the generated coding algorithm can be plugged in JasPer and then the encoder is benchmarked against the best library to compare the performance.
5.2 JPEG 2000 Reference and Performance Implementations

5.2.1 JJ2K

JJ2000 is the Java reference implementation (Part 5) of JPEG 2000. This project was the result of a partnership between EPFL, Canon Research France and Ericsson. It ended in September 2001 with a complete implementation of the normative parts of the JPEG 2000 core coding system (Part 1).

5.2.2 JasPer

JasPer is a project to create a reference implementation of the codec specified in the JPEG-2000 Part-1 standard, (ie. ISO/IEC 15444-1). The project was started by Image Power, Inc. and the University of British Columbia. It consists of a C library and some sample applications useful for testing the codec.

5.2.3 Intel C++ Compiler with Intel Integrated Performance Primitives

**Intel C++ Compiler** (also known as icc or icl) is a group of C/C++ compilers from Intel which is available for Linux, Microsoft Windows and Mac OS X.

Intel C++ Compiler supports compilation for its IA-32, Intel 64, Itanium 2, and XScale processors. The Intel C++ Compiler for x86 and Intel 64 features an automatic vectorizer that can generate SSE SIMD instructions. Also, it supports OpenMP and automatic parallelization for symmetric multiprocessing.

**Intel Performance Primitives (IPP)** is a library of multi-core-ready, optimized software functions for multimedia and data processing applications, produced by Intel.

The library supports Intel and AMD processors and is available for Windows, Linux and Mac OS X operating systems. Intel IPP is a lightweight library that exposes data type, data structure and other options in the naming convention of each operation. By providing a huge number of functions, and keeping the interfaces lightweight, IPP is designed to provide building blocks for multimedia applications and data processing applications.

Intel IPP functions are mainly divided into three major groups: Signal (with linear array or vector data), Image (with 2D arrays for typical color spaces) and
Matrix (with nxm arrays for matrix operations). It includes the range of Video Decode/Encode, Audio Decode/Encode, JPEG/JPEG2000, Computer Vision, Cryptography, Data Compression, Image Color Conversion, Image Processing, Ray Tracing/Rendering, Signal Processing, Speech Coding, Speech Recognition, String Processing, Vector/Matrix Mathematics. The library takes advantage of processor advances including short vector instructions and is optimized for multi-core processors.

Intel Performance Primitives Library contains two sections of functions that are concerned with JPEG 2000. One is the wavelet transform functions that are specified for JPEG 2000, and the other for the entropy encoding and decoding procedures. IPP for Windows and Linux comes with example of JPEG 2000 encoder and decoder with limited options.

5.3 Runtime Profile for Jasper Implementation

For performance comparison, the tests will be based on C/C++ implementations of JPEG 2000 encoding algorithm. As IPP is only open source, the coding algorithm is generated and plugged in to the JasPer reference implementation, and the final performance (run time) is compared against the fast IPP implementation.

The runtime profile of JPEG 2000 encoding with JasPer implementation over 8 different pictures in both lossy and lossless mode is shown in Figure 5.1. This is done by putting time stamps between each of the components. It shows that Tier-1 coding takes the most of the time of the entire encoding process.
Figure 5.1: JasPer JPEG 2000 Encoder Breakdown Profile.
Entropy encoding losslessly compresses the data by reducing its redundancy. The coding algorithm in JPEG 2000 can be divided into two parts: Tier-1 and Tier-2. Tier-1 mainly comprises of the bit-plane coding (BPC) algorithm EBCOT and the binary arithmetic coding (BAC) algorithm MQ-coder. (Figure 6.1). EBCOT stands for Embedded Block Coding with Optimal Truncation, it encodes each bit-plane by applying three coding passes in order and each bit is coded in only one of the coding passes. The three coding passes in the order in which they are performed on each bit-plane are significant propagation pass, magnitude refinement pass, and cleanup pass. The pairs in the stream consist

![Tier-1 Breakdown Block Diagram](image)

**Figure 6.1**: Tier-1 Breakdown Block Diagram.
of the context and decision, and are coded using a variation of context adaptive BAC, called the MQ-coder. The context is used to select the estimate probability to adjust the interval to code the decision into the output bit stream. Tier-2 efficiently represents the layer and block summary information for each-block in a structure called Tag Tree.

Tier-1 and Tier-2 coding algorithms are defined in Part 1 of International Standard ISO/IEC 15444-1, JPEG 2000 Image Coding System, specifically in Annex D (Coefficient bit modelling) and Annex C (Arithmetic entropy coding)[14], respectively. This chapter briefly describes what is defined in the standard and how to interpret the standard by first defining a few terms, coding operators and coding passes.

6.1 Tier-1 Coding

The input of Tier-1 coding is the quantized coefficient bits generated by the discrete wavelet transformation. These coefficients associated according to the sub-bands from the result of the transform are further arranged into rectangular blocks called “code blocks”. Each of the code blocks is independently coded. The coding is performed starting from the most significant bit-plane with non-zero element to the least significant plane.

All quantized transform coefficients, \( q[i, j] \), where \((u, v)\) denotes the coordinate on that code-block, are signed values even when the original color components are unsigned. These coefficients are conceptually expressed in a sign-magnitude representation. For a particular sub-band, there is a maximum number of magnitude bits, \( M_b \).

\[
M_b = \log_2(\max(|q[i,j]|)) \quad \text{where} \quad i \leq m, j \leq n
\]

As context-based arithmetic coding is employed in EBCOT, meaning that context selection is necessary. Generally speaking, context selection is performed by examining coefficient or state information for the 4-connected or 8-connected neighbors of a sample as shown in Table 6.1. Also, each bit is encoded in only once and once only in one of three coding passes, the encoding of the current bit will depends on the coding in the previous pass, and even on the coding of bit on the same position but more significant bit-plane. Thus, states are maintained to keep track of these information to help to decide the encoding of the current bit.
<table>
<thead>
<tr>
<th>Code</th>
<th>Decision(D)</th>
<th>Context(CX)</th>
<th>Brief explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>-</td>
<td>-</td>
<td>Go to next coefficient or column</td>
</tr>
<tr>
<td>C1</td>
<td>newly significant?</td>
<td>Table D1</td>
<td>Decode significant bit of the current coefficient</td>
</tr>
<tr>
<td>C2</td>
<td>sign bit</td>
<td>Table D3</td>
<td>Decode sign bit of current coefficient</td>
</tr>
<tr>
<td>C3</td>
<td>current magnitude bit</td>
<td>Table D4</td>
<td>Decode magnitude refinement pass bit of current coefficient</td>
</tr>
<tr>
<td>C4</td>
<td>0</td>
<td>Run-length context label</td>
<td>Decode run-length of four zeros</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>Decode run-length not of four zeros</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>First coefficient is first with non-zero bin</td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td>01</td>
<td>UNIFORM</td>
<td>Second coefficient is first with non-zero bin</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Third coefficient is first with non-zero bin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Fourth coefficient is first with non-zero bin</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: 4, 8-connected Context

<table>
<thead>
<tr>
<th>Decision</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>Is this the first significance bit-plane for the code-block?</td>
</tr>
<tr>
<td>D1</td>
<td>Is the current coefficient significant?</td>
</tr>
<tr>
<td>D2</td>
<td>Is the context bin zero?</td>
</tr>
<tr>
<td>D3</td>
<td>Did the current coefficient just become significant</td>
</tr>
<tr>
<td>D4</td>
<td>Are there more coefficients in the significant propagation?</td>
</tr>
<tr>
<td>D5</td>
<td>Is the coefficient insignificant?</td>
</tr>
<tr>
<td>D6</td>
<td>Was the coefficient coded in the last significance propagation?</td>
</tr>
<tr>
<td>D7</td>
<td>Are there more coefficients in the magnitude refinement pass?</td>
</tr>
<tr>
<td>D8</td>
<td>Are four contiguous undecoded coefficients in a column each with a 0 context?</td>
</tr>
<tr>
<td>D9</td>
<td>Is the coefficient significant?</td>
</tr>
<tr>
<td>D10</td>
<td>Are there more coefficients remaining of the four column coefficients?</td>
</tr>
<tr>
<td>D11</td>
<td>Are the four contiguous bits all zero?</td>
</tr>
<tr>
<td>D12</td>
<td>Are there more coefficients in the cleanup pass?</td>
</tr>
</tbody>
</table>

Table 6.2: Coding in the context model flow chart in Figure 6.2.

Table 6.3: Decision in the context model flow chart in Figure 6.2.
Figure 6.2: Flow chart for all coding passes on a code block bit-plane defined by ISO/IEC 15444-1. defined in [14]
6.1.1 Basic terms and definitions

The ISO standard describes the encoding in a way that is easy to see the process flow and for conceptual understanding but difficult to translate to implementation (Figure 6.2, Table 6.2 and Table 6.3). To make it easier to understand in the perspective in order to implement the complex algorithm, the following terms are defined, explained and represented:

**Code-Block** $q$: A code-block is a two-dimensional array that consists of signed integers (wavelet coefficients with or without quantization). Each code-block has width $m$ and height $n$ that specify its size. $q[i,j]$ denote the value at coordinate $(i,j)$ in the code-block.

**Sign Array** $\chi$: is a two-dimensional array representing the signs of the elements of a code-block. It has the exact same dimensions as the code-block. Each element $\chi[i,j]$ represents the sign information of the corresponding element $q[i,j]$ in the code-block as follows.

$$\chi[i,j] = \begin{cases} 1 & \text{if } q[i,j] < 0 \\ 0 & \text{else} \end{cases}$$

**Magnitude Array** $v$: is a two-dimensional array of unsigned integers. The dimension of this array is exactly the same as the dimension of the code-block. Each element of $v$ represents the absolute value of the integer at the corresponding location in the code-block, that is, $v[i,j] = |q[i,j]|$.

**Bit-Plane** $v_b$: The magnitude array $v$ can be also conceptually viewed as a three-dimensional array, splitting the unsigned integer into a one-dimensional sequence of bits (0 or 1) to form the third dimension. Each particular bit-wise position $b$ of bits of every element of $v$ constitutes a single bit-plane $v_b$. Thus $v$ can be viewed a one-dimensional array consisting of a number of bit-planes $v_b$. The notation $v_b[i,j]$ is used to denote the $b$th bit of $v[i,j]$.

**State Variables** $\sigma, \sigma'$ and $\eta$: Three two-dimensional binary arrays are created to indicate the coding states of each element in the code-block during the coding process. These arrays have exact the same dimension as the code-block. Initially, each of the elements of these arrays are set to zero. Once the coding process starts, the values of the two variables $\sigma[i,j]$ and $\sigma'[i,j]$ change to one when certain conditions stratifies, but are never changed back to zero until the entire code-block is encoded. On the other hand, the values of $\eta[i,j]$ reset to 0 right after completion of coding of each bit-plane. Brief interpretations of these three variables are as follows:
Table 6.4: Code-block scan pattern example with width=8, height=6, the numbers in the table signify bit encoding order.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>17</td>
<td>21</td>
<td>25</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>26</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td>19</td>
<td>23</td>
<td>27</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>34</td>
<td>36</td>
<td>38</td>
<td>40</td>
<td>42</td>
<td>44</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>35</td>
<td>37</td>
<td>39</td>
<td>41</td>
<td>43</td>
<td>45</td>
<td>47</td>
<td></td>
</tr>
</tbody>
</table>

- $\sigma_{i,j} = 1$ indicates that the first nonzero bit of $v_{i,j}$ at row $i$ and column $j$ has been coded, that corresponds to the reverse of the significance bit in the standard (D1, D5 and D9 in Table 6.3). It is assumed to be significant ($\sigma = 0$) if so far on different bit-plane the same position $(i,j)$, a 1 valued bit ($v_b[i,j] = 1$) hasn’t been seen. Once a 1 value bit is seen, the state will turned to insignificant ($\sigma = 1$). If on the current bit-plane $\sigma = 0$ and $v_b[i,j] = 1$, it’s is called “newly significant”, which is corresponds to D3 in Table 6.3.

- $\sigma'_{i,j} = 1$ indicates that a magnitude refinement coding operation (defined in the next section) has been applied to $v_{i,j}$, this is only used in C3 in Table 6.2.

- $\eta_{i,j} = 1$ indicates that zero coding operation (defined in the next section) has been applied to $v_{i,j}$ in the previous significant propagation pass. it corresponds to D6 in the standard in Table 6.3;

**Preferred Neighbourhood:** An element $p[m,n]$ in the code-block is said to be in a preferred neighbourhood if at least one of its eight adjacent neighbors has $\sigma$ value equal to 1, one equivalent expression is $\sum \sigma_H + \sum \sigma_V + \sum \sigma_D \geq 0$

**Scan Pattern:** Scan pattern defines the order of encoding or decoding the bit-planes of a code-block. All of the three coding passes (significance, refinement, and cleanup) scan the samples each bit-plane of the code block in the same fixed order, shown as an example in Table 6.4. The code block is conceptually partitioned into horizontal stripes, each having a nominal height of four samples. If the code block height is not a multiple of four, the height of the bottom stripe will be less than this nominal value. As shown in the Table, the stripes are scanned from top to bottom. Columns are scanned from left to right within a stripe and scanned from top to bottom within a column.
6.1.2 Coding Operations

Coding operation produces the pair of context and decision value depend on the current coding position and its context value, along with the states. There are four possible coding operations used in EBCOT to generate the values of context (CX) and decision (D) as intermediate data before the BAC. They are: Zero Coding, Sign Coding, Magnitude Refinement Coding and Run-Length Coding. The former three coding operations encode one bit and generate one context and decision pair, while the Run-Length Coding can generate up to three pairs and encode up to four bits.

The context values can be seen as symbols but are usually represented by integers, the decision value is binary - either zero or one. The standard does not mandate the value of the symbols as they are only used intermittently between the EBCOT and the MQ-coder.

Exactly when or where these operations are applied is subject to current coding pass, the location of the current element, and the context and values of the state variables. Each coding pass will traverse the current bit-plane, in each bit position decides whether and which coding operations will be applied, update the values of the state variables and moves on to the next bit position according to the scan pattern, before reaching the end of the current bit-plane and continue with the next.

In this section all of these coding operations are introduced in manner that is close to what the standard defines it and complement with a bit mathematics formula to . Of all the description below, the index of the current bit-plane is assumed to be \( b \).

**Zero Coding:** For zero coding operation, the decision bit D is equal to current bit-plane value \( v_{b}[i, j] \), context value CX is selected from one of the three zero coding context tables related to the relevant sub-band (LL, LH, HL or HH) the code-block belongs to. The sub-band are divided by the previous wavelet transformation. There are nine entries in each context table. They are derived using the values of the significance states \( \sigma \) of the eight surrounding neighbors of the current coefficient bit, \( v_{b}[i, j] \). As shown in the Table 6.5, each entry depends on the current sub-band, how many and which neighbours of \( v_{b}[i, j] \) are significant.

\[
\begin{align*}
CX &= T_{ZC}(\sum \sigma[i, j]_H, \sum \sigma[i, j]_V, \sum \sigma[i, j]_D, \text{band}) \\
D &= v_{b}[i, j]
\end{align*}
\]
The $T_{ZC}$ denotes the zero coding context look up table and can be seen as a function that takes the input of $\sum \sigma[i, j]_H, \sum \sigma[i, j]_V, \sum \sigma[i, j]_D$, and the band the code-block belongs to.

**Magnitude Refinement Coding (MRC):** For magnitude refinement coding, decision value $D$ at position $(i, j)$ in the $b$th bit-plane is simply equal to the bit value $v_b[i, j]$. The value of context $CX$ is determined by $\sigma'[m, n]$ and the sum of its eight adjacent values of the state variable $\sigma$ is in Table 6.7. As the $\sigma'$ value is initially zero and turned to be one if a refinement coding is applied. Its value will 0 when if it is the first refinement bit that is significant (true in the table). Also, the $\times$ symbol in the table (and follows) means “don’t care”. If $\sigma' = 1$ at the current position, which indicates that it is not the first magnitude refinement for this element, then $CX = 16$ regardless of the neighbouring significance values ($\sigma$).

$$\begin{align*}
\text{First refinement } (\sigma') & \quad \sum \sigma[i, j]_H + \sum \sigma[i, j]_V + \sum \sigma[i, j]_D & \text{Context} \\
\text{false (1)} & \times & 16 \\
\text{true (0)} & \geq 1 & 15 \\
\text{true (0)} & 0 & 14
\end{align*}$$

**Table 6.7: Magnitude Refinement Coding Context Lookup Table**

**Sign Coding:** The context $CX$ value for the sign coding are determined by a horizontal reference value $H$ and a vertical reference value $V$, described in Table 6.8. The XORbit $\bar{\chi}$ value is also determined by the $H$ and $V$, and partially determines the decision value $D$.

$$\begin{align*}
\{ & CX = T_{MRC}(\sigma', \sum \sigma[i, j]_H + \sum \sigma[i, j]_V + \sum \sigma[i, j]_D) \\
& D = v_b[i, j]
\end{align*}$$
<table>
<thead>
<tr>
<th>Horizontal Contribution</th>
<th>Vertical Contribution</th>
<th>Context</th>
<th>XORbit $\chi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>13</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 6.8:** Sign Coding Context from Vertical/Horizontal Contributions

<table>
<thead>
<tr>
<th>$V_0/H_0$</th>
<th>$V_1/H_1$</th>
<th>$V/H$ contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>significant, positive</td>
<td>significant, positive</td>
<td>1</td>
</tr>
<tr>
<td>significant, negative</td>
<td>significant, positive</td>
<td>0</td>
</tr>
<tr>
<td>insignificant</td>
<td>significant, positive</td>
<td>1</td>
</tr>
<tr>
<td>significant, positive</td>
<td>significant, negative</td>
<td>0</td>
</tr>
<tr>
<td>significant, negative</td>
<td>significant, negative</td>
<td>-1</td>
</tr>
<tr>
<td>insignificant</td>
<td>significant, negative</td>
<td>-1</td>
</tr>
<tr>
<td>significant, positive</td>
<td>insignificant</td>
<td>1</td>
</tr>
<tr>
<td>significant, negative</td>
<td>insignificant</td>
<td>-1</td>
</tr>
<tr>
<td>insignificant</td>
<td>insignificant</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 6.9:** Contribution of the vertical/horizontal neighbors to sign context

The reference values of $H$ and $V$ indicate three possible situations as follows:

- 0 indicates that both neighbours are insignificant, or both neighbours have opposite signs.
- 1 indicates that one or both neighbours are significant with positive sign.
- -1 indicates that one or both neighbours are significant with negative sign.

And the value of $H, V$ are determined by the sign values $\chi$ (1 being negative and 0 being positive) and significance state $\sigma$ (0 being significant and 1 being insignificant) of the 4-context neighbour, as in Table 6.9.

As shown in Table 6.8, $H$ and $V$ are used together to determine the context (CX) and a binary value $\chi$, which in terms is used to calculate the value of $D$ as $D = \bar{\chi} \oplus \chi[i,j]$, where $\oplus$ represents an exclusive-OR operation.
The values of $H$ and $V$ are obtained by the following equations, using the state variables defined above:

$$H = \min(1, \max(-1, \sigma[i, j-1]) \times (1 - 2\chi[i, j-1]) + \sigma[i, j+1] \times (1 - 2\chi[i, j+1])$$

$$V = \min(1, \max(-1, \sigma[i-1, j]) \times (1 - 2\chi[i-1, j]) + \sigma[i+1, j] \times (1 - 2\chi[i+1, j])$$

$$\begin{cases} 
CX &= T_{SC_{CX}}(H, V) \\
D &= \chi[i, j]\oplus T_{SC_{D}}(H, V)
\end{cases}$$

**Run-Length Coding (RLC):** Unlike the other three coding operations, run-length coding may code from one to four consecutive bits in the current scan pattern stripe. Also, run-length coding has to be applied on the bit on the first row of each strip, so the possible four consecutive bits are in the same column. Exactly how many bits are encoded depends on where the first 1-value bit (if any) is located in the four consecutive bits. If all of them are 0’s, then all four bits are coded. If one (or more) of these four bits is 1, then the first 1 in the scan pattern and any preceding 0’s in between the current scan location are coded.

A run-length coding operation may generate either one or three pairs of context and decision pairs, depending on whether the four consecutive bits are all 0’s or not. The first $D$ is equal to 0 if all four bits are equal to 0; otherwise it is equal to 1. For both cases, $CX$ is equal to a unique run-length context value 17. In other words, a $(CX, D)$ pair with values $(17, 0)$ indicates four consecutive 0 bits, and a $(CX, D)$ pair with values $(17, 1)$ means there is at least one 1 bit in the current scan pattern stripe. In the case that at least one of the four bits in the current scan pattern is 1, two more $D$’s are used with a “UNIFORM” context value 18 to indicate the location of the first 1 bit in the 4-bit scan pattern. All the possibilities of Run-Length Coding and the corresponding generated pair of context and decision values can be found in Table 6.10.

Another significant difference between run-length coding and the other coding operations is that run-length coding might move the next coding position by an

<table>
<thead>
<tr>
<th>$v_b$</th>
<th>Move Position</th>
<th>(Context, Decision)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 × ×</td>
<td>0 (18,1), (17,0), (17,0)</td>
<td></td>
</tr>
<tr>
<td>0 1 ×</td>
<td>1 (18,1), (17,1), (17,0)</td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>2 (18,1), (17,0), (17,1)</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>3 (18,1), (17,1), (17,1)</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>4 (18,0)</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.10: Run Length Coding
offset of 0 to 4 after it is applied, while other coding operations will not change the current coding position.

6.1.3 Coding Passes

There are three coding passes in EBCOT algorithm: significance propagation pass (SPP), magnitude refinement pass (MRP), and cleanup pass (CUP). These three coding passes are applied sequentially on each bit-plane of a code-block except the first bit-plane (the most significant bit-plane), where only with the cleanup pass is applied. The coding order of the bit-plane is from the most significant plane to the least significant plane. In each coding operation, checks are made to determine whether and which coding operation(s) are going to be applied.

Each coding pass completes a run according to the scan pattern in the current bit-plane, the next coding pass restarts the scan pattern from the beginning. Except the first bit-plane, which is only encoded with cleanup pass, all bit-planes are coded in the order of significance propagation pass, magnitude refinement pass, and cleanup pass.

Depending on the coding pass, a bit on a the current bit-plane might be applied by one or more coding operation mentioned before, it could also be skipped (doing nothing). That means the context and decision pair generated by the bit-plane and code block is not constant but context driven.

The values of the bit-planes are unchanged throughout the entire coding process, but the state variables might be changed from the initial values (which is always 0) to 1 after some coding operations, state variables \( \sigma \) and \( \sigma' \) will never change back to 0 while \( \text{eta} \) will be reset to 0 after each bit-plane is encoded.

**Significance Propagation Pass**: Bit positions that have a magnitude of 1 for the first time (i.e., the most significant bit of the corresponding coefficients) are coded in this pass. Figure 6.3 shows the flow diagram of SPP.

This is the first pass applied to every bit-plane of a code-block, except the first bit-plane. Significance propagation pass first applies zero coding if the current scan position \((i, j)\) is in a preferred neighbourhood and \( \sigma[i, j] = 0 \). That is, the current bit is significant and at least one of the neighbours are insignificant. Proceed to the next bit position if zero coding cannot be applied. If the zero coding is applied, the coded bit of the current bit-plane \( \chi[i, j] \) is set to 1. After zero coding is completed, check is made to determine whether sign coding is needed at the current bit position \((i, j)\). If \( v_b[i, j] = 1 \), then it’s considered “newly significant” and sign coding is applied. After this, \( \sigma[m, n] \) is set to be 1
to denote that the bit position is insignificant. The significant propagation pass will continue coding the bits along the scan pattern until all of the bits of the bit-plane are coded.

**Magnitude Refinement Pass:** Bit positions that have not been coded in SPP and that have had magnitude of 1 in previous bit-planes (i.e., the current bit is not the most significant bit of the corresponding sample coefficient) are coded in this pass. Figure 6.4 shows the flow diagram for MRP.

If the current bit is insignificant and hasn’t been coded in the previous significant propagation pass - equivalently the state variables $\sigma[i, j] = 1$ and $\chi[i, j] = 0$, 

---

**Figure 6.3:** Flowchart of Significant Propagation Pass.
then magnitude refinement coding (MRC) is applied on to the current position $(i, j)$ and $\sigma'[i, j]$ is set to 1, meaning magnitude refinement coding has been applied to position $(i, j)$. Like SPP, MRP will continue coding the bits along the scan pattern until all of the bits of the bit-plane are coded.

**Cleanup pass**: Bit positions that have not been coded in either of the two earlier passes are coded in this pass. Cleanup pass also incorporates run-length coding to help in coding a string of zeros.

CUP is applied to every bit-plane of a code-block after completion of MRP, except the first bit-plane, which does not need the SPP or MRP.

In each position $(i, j)$ follows in the scan pattern, CUP first checks whether the current bit has been encoded in the previous two passes - that is, $\eta[i, j] = 0$ (no SPP has been applied) and $\sigma[i, j] = 0$ (no MRP has been applied). If any one of them are not satisfied, it proceed to the next bit position in the bit-plane. Otherwise, check is made which coding operator to apply, from either run-length coding (RLC) or zero coding (ZC). RLC is applied when all the following three conditions are true.
• \( m \) is a multiple of four, including \( m = 0 \). That is, the current bit is on the first row of the strip.

• \( \sigma = 0 \) for the four consecutive locations on the same column, starting from the current scan position.

• \( \sigma = 0 \) for all the adjacent neighbours (8-context neighbours) of the four consecutive bits in the column.

If any one of the above conditions is false, then zero coding (ZC) is applied to the current location.

Depending on whether run-length coding or zero coding is applied in the current location, the number of bits coded may vary. The next bit to be coded is the bit after the last coded bit. Also, run-length coding would not be applied to the last section with fewer than four rows in a scan pattern because there would not be four consecutive bits available in the same column.

After completion of the run-length coding or zero coding, check is made to determine whether we need to apply sign coding (SC) before we move on to code the next bit. Suppose the last coded position is \((i, j)\). If \( v_b[i, j] = 1 \), which indicates the current bit is the most significant bit, then similar to significant propagation pass, sign coding is applied and the significance state \( \sigma \) is set to one. CUP will continue coding the bits along the scan pattern until all of the bits of the bit-plane are coded.

After completion of the cleanup pass for a bit-plane, \( \eta[i, j] \) will be reset to zero for all \( i \) and \( j \) in the bit-plane before moving into the next bit-plane.

### 6.1.4 Arithmetic Coding

The MQ-coder used in JPEG 2000 belongs to a category of entropy coders that are referred as arithmetic coders. Arithmetic code compress data losslessly by presenting the stream of input into another representation by representing the frequent symbol using fewer bit and infrequently used symbol using more bits. Arithmetic coding, binary arithmetic coding, context adaptive arithmetic coding are briefly introduced in Section 4.1. MQ-coder is much a simpler algorithm compared with EBCOT and is not performance critical. Little analysis and optimization is done to the MQ-coder in this thesis.
Figure 6.5: Flowchart of Cleanup Pass.
6.2 Tier-2 Coding

In Tier-2 coding, the information of the compressed codewords generated in the Tier-1 coding step are encoded using a Tag Tree coding mechanism.

Tag Tree is a particular type of quad-tree data structure, which provides the framework for efficiently representing information of the codeblocks and their compressed codewords in JPEG 2000.

After the compressed bits for each code-block are generated by Tier-1 coding, the Tier-2 coding engine efficiently represents the layer and block summary information for each code-block. A layer consists of consecutive bit-plane coding passes from each code-block in a tile, including all the sub-bands of all the components in the tile. The block summary information consists of length of compressed code words of the codeblock, the most significant magnitude bit-plane at which any sample in the code-block is nonzero, as well as the truncation point between the bit-stream layers, among others. The decoder receives this information in an encoded manner in the form of two tag trees. This encoding helps to represent this information in a very compact form without incurring too much overhead in the final compressed file.

Similar to MQ-coder, Tier-2 coding algorithm comprise a little fraction of the entire running time, thus it is not analysed nor implemented in the thesis.
The input of EBCOT is a two-dimensional array of integers that is within a subband which the discrete wavelet transforms generates, and the output is an array of pairs (CX, D). CX is the context value which can been seen as a table index ranged from 0 to 17, it represents context of the bit that encodes including its surrounding bits and states, while D is the binary decision value. The input size of the EBCOT is always a power of 2 with the minimum height and width is 4 and maximum being 1024. Typical choice of code block size is $64 \times 64$ or $32 \times 32$.

The input of the two dimensional array of signed integers can be conceptually separated by the sign bit array (non-negative being 0 and negative being 1), and the magnitude. The magnitude value array can be further seen as a data cube of bits, of the most significant bits at the top and least significant bits at the bottom. EBCOT will start coding from the most significant bit plane to the least significant plane.

EBCOT consists of three coding passes: Significance Propagation Pass, Magnitude Refinement Pass and Cleanup Pass, each pass traverses through the same
bit-plane before going to the next less significant plane. The most non-trivial significant plane essentially only goes through the cleanup pass.

7.2 Operators Language and Extension for EBCOT

As an extension of the Signal Processing Language (SPL), Operator Language (OL) is initially used in the matrix-matrix multiplication. The most significant contribution so far is to enable the algorithm to take multiple vector input and produce multiple vector output, and drop the constraint of linearity in operator. Formal rewriting rules has been developed for Operator Language. However, so far the language is far enough to express the EBCOT algorithm.

This section first analyses EBCOT in a high level, try to determine what is missing and describe the solution adding new operators or constructs.

One thing worth noticing is that also linearity is not a requirement for operators any more, the operators to be introduced are best tried to be maintained linear, this will provides the advantage of using the existing rules in rewriting engine to optimize on the algorithm level.

7.2.1 Memory Access

The input of EBCOT consists of at least the magnitude array and the state array of equal size \([m, n]\). For each coding operation, access to the 9 context values are possibly needed. Gather operator and its extension defines the memory access on linearized memory.

**Gather.** Let \(e^N_k \in \mathbb{C}^{n \times 1}\) be the canonical basis vector with entry 1 in position \(k\) and entry 0 elsewhere. An index mapping function \(f^{N \rightarrow N}\) generates the gather matrix (\([\cdot]\)T is the matrix transposition)

\[
G_{f^{n \rightarrow N}} := [e^N_{f(0)}|e^N_{f(1)}|...|e^N_{f(n-1)}]^T
\]

This implies that for two vectors \(x = (x_0, ... x_{n-1})^T\) and \(y = (y_0, ... y_{N-1})^T\),

\[
y = G_{f^{n \rightarrow N}} x \Leftrightarrow y_i = x_{f(i)},
\]

**9-context Gather.** As 9 context access is the basic access pattern in EBCOT, an extension of the Gather above it provided. The 9-context gather \(Gath_9\) takes the size \((m, n)\) of input as parameters. Input for \(Gath_9(m, n)\) is the full vector integer (for generality) input whose size is \(m \times n\) and the current location \((i, j)\), a natural number vector of size 2. Output is the vector packed with the 9
neighbour values of the current location. We can express the signature of the 9-context gather as below:

\[ \text{Gath}_9(m, n) : \mathbb{Z}^{m*n} \times \mathbb{N}^2 \rightarrow \mathbb{Z}^9 \]

The index mapping function can be informally formulated as:

\[
f(n) = \begin{cases} 
1 & \text{if } n = m \cdot (j - 1) + (i - 1) \\
1 & \text{if } n = m \cdot (j - 1) + i \\
1 & \text{if } n = m \cdot (j - 1) + (i + 1) \\
1 & \text{if } n = m \cdot j + (i - 1) \\
1 & \text{if } n = m \cdot j + i \\
1 & \text{if } n = m \cdot j + (i + 1) \\
1 & \text{if } n = m \cdot (j + 1) + (i - 1) \\
1 & \text{if } n = m \cdot (j + 1) + i \\
1 & \text{if } n = m \cdot (j + 1) + (i + 1) \\
0 & \text{else}
\end{cases}
\]

Specifically, a general gather can be defined as:

\[ \text{Gath}_1(m, n) : \mathbb{Z}^{m*n} \times \mathbb{N}^2 \rightarrow \mathbb{Z} \]

**Gather fix element.** In the formula below, \( G_i \) where \( i = 0..8 \) means \( G_{9(i)} \), that is, Gather \( i \)th element from a vector whose length is 9.

**Vector Gather.** The input and output vector and their order is defined once an OL operator is defined. Vector Gather is introduced to give the flexibility to pick or shuffle the vector input and output. It is defined similarly as the general Gather but only applies on whole vectors rather than element within the input vector.

\[ \mathcal{G}_{f_{n-N}} : [\xi^N_{f(0)} | \xi^N_{f(1)} | \ldots | \xi^N_{f(n-1)}]^T \]

Suppose the signature of the vector gather is \((x_0, \ldots x_{n-1}), (y_0, \ldots y_{N-1})\),

\[ y = \mathcal{G}_{f_{n-N}} x \iff y_i = x_{f(i)}, \]

In the formula before, the index mapping function is written as the names of the input (as in Table 7.1) needed for easy understanding, for example:

\[ \mathcal{G}([\sigma, pos]) \]
takes \( \sigma \) and current position, thus the domain of the above operator would be \((m \times n, 2)\).

Identity \( I \) can be seen as a special vector gather - just pick the only input and serve it as the output. Copy is also a special vector gather, it is expressed as:

\[
\begin{pmatrix}
1 \\
1
\end{pmatrix}: (x, (x, x))
\]

That is, copying the input twice. The 1 being \( I(1) \) and the underline denote that the operator is on the whole vector.

### 7.2.2 Iteration and Selection

**Iterative Compose.** Iterative compose has already existed in Signal Processing Language, and is defined in operator language as:

\[
\prod_{i=0}^{m-1} (A(x_0, ... x_{n-1}))
\]

Note that in order to differentiate from SPL, OL uses \( \prod \) rather than \( \Sigma \). And similar to SPL, the domain and range of the operator \( A \) inside iterative compose have to be the same.

**Selection.** EBCOT coding passes need to choose whether or which coding operation to apply on the current location. Selection operators are defined to make the choice about which operator to be applied. Selection operator is expressed as:

\[
\langle B, C \rangle_A :((u_0, ... u_{M-1}, x_0, ... x_{n-1}), (y_0, ... y_{N-1}))
\]

if \( B \) and \( C \) have signature \((x_0, ... x_{n-1}), (y_0, ... y_{N-1})\)

and \( A \) has signature \((u_0, ... u_{M-1}), 1)\)

The requirement is that \( B \) and \( C \) will need to have the same signature, for example \((x_0, ... x_{n-1}), (y_0, ... y_{N-1})\), \( A \)'s range need to be a boolean value \( \mathbb{Z}_2 \) and the domain could be anything, take the above example, \((u_0, ... u_{M-1})\). The order of vector input will be \( A, B \) or \( C \). Conceptually, \( A \) is evaluated first and if the result is 1 (true), then the entire operator is equivalent to \( B \), if the result being 0 (false), then the operator is equivalent to \( C \).

An alternative of expressing selection is:

\[
\langle B, C \rangle :((1, x_0, ... x_{n-1}), (y_0, ... y_{N-1}))
\]

if \( B \) and \( C \) have signature \((x_0, ... x_{n-1}), (y_0, ... y_{N-1})\)
The difference being that the second form of selection doesn’t take the operator $A$ to evaluate which to select but rather take a boolean input. This operator is used when the decision is a result of the earlier evaluation.

**Conditional Iterative Compose.** For the iterative compose, the number of iteration has to be known when the operator is defined. This would not be the case for the EBCOT as the number of coding operations applied and the bit move in run-length coding is context driven. The conditional iterative compose is expressed by:

$$\prod \langle B \rangle_A : ((u_0, \ldots u_{M-1}, x_0, \ldots x_{n-1}), (x_0, \ldots x_{n-1}))$$

if $B$ has signature $((x_0, \ldots x_{n-1}), (x_0, \ldots x_{n-1}))$

and $A$ has signature $((u_0, \ldots u_{M-1}), 1)$

The constraint further to the selection is that $B$ will need to have the same domain and range, $(x_0, \ldots x_{n-1})$ in the above representation. The operator $A$ will be evaluated before the $B$ operator, if $A$ returns 1 (true) the iteration will continue until $A$’s value turned to be 0 (false).

### 7.2.3 Other Operators

**Lookup Table.** Table lookups are constantly used in the coding operations. It is defined as:

$$T_{name} : ((x_0, \ldots x_{n-1}), (y_0, \ldots y_{N-1}))$$

A general lookup table takes arbitrary number of input (domain) and generate arbitrary output (range). However, without losing generality, the signature can be simplified as $(x, y)$ as the domain and range can be packed into one vector.

Implementation of lookup tables can be simply a conditional code template, or a multi-dimension array reference.

### 7.2.4 New Rewrite Rules

New Rewrite rules are also introduced with the operators defined above, the most important of which are:

$$\langle B, C \rangle_A \circ D = \langle B, C \rangle \circ (A \times D)$$

$$D \circ \langle B, C \rangle_A = \langle D \circ B, D \circ C \rangle_A$$
7.3 Prototype Implementation

The first implementation of the EBCOT is derived directly from the interpretation of the JPEG 2000 standard shown in the last chapter. It serves a prototype and tries to put the pieces together to make it possible to generate the EBCOT algorithm using Operator Language. This is done without any optimization.

One of the most apparent differences between the paradigm of algorithm description of EBCOT and what is described by Operator Language is that the coding operations in EBCOT kept the state information while OL operators are seen as “stateless”. The solution to overcome the gap between these is to include the state as the input and output. The default output will be the same as the input unless the operator changes the values of the states. A “LinkIO” tag is applied on the operator to denote the nature of the input and output for implementation optimization.

Also it’s observed that the coding operators only need to access a small fraction of the inputs and change a even smaller fraction of the output, an embedded module of operators with input reordering is used. Together with the above LinkIO to offer a flexible yet powerful language to be able to describe the complication EBCOT algorithm.

7.3.1 Input and Output

The input of EBCOT algorithm is the code-block values, and the parameters such as the size and the sub-band the code-block belongs to. The code-block can be further broken down to sign plane, magnitude bit plane etc. Combine with the state variables they serve as the input for the OL formula for EBCOT. The output for EBCOT, on the other hand, is a stream of pairs of context value and decision (CX, D). As EBCOT is context driven and the length of the output pair is unknown at compile time, it is stored in a buffer similar to pointer with an integer specifying the length. The complete list of input and output is listed in Table 7.1.

7.3.2 EBCOT Top Level non-terminals

Using the extra operators defined above, EBCOT can be defined as:

$$\text{EBCOT} := \prod_{i=P_0}^0 \text{EBCOTPlane}_i \circ \text{EBCOTInit}$$

where $P_0$ is index for the most significant plane.
<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>symbol</th>
<th>signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sign Plane (bit)</td>
<td>$\chi$</td>
<td>$\mathbb{N}^{m\times n}$</td>
</tr>
<tr>
<td>1</td>
<td>Magnitude bit Plane (bit)</td>
<td>$v^P$</td>
<td>$\mathbb{N}^{m\times n}$</td>
</tr>
<tr>
<td>2</td>
<td>significance bit</td>
<td>$\sigma$</td>
<td>$\mathbb{N}^{m\times n}$</td>
</tr>
<tr>
<td>3</td>
<td>magnitude encoded bit</td>
<td>$\sigma'$</td>
<td>$\mathbb{N}^{m\times n}$</td>
</tr>
<tr>
<td>4</td>
<td>significance encoded bit</td>
<td>$\eta$</td>
<td>$\mathbb{N}^{m\times n}$</td>
</tr>
<tr>
<td>5</td>
<td>Current Coding Bitplane (integer)</td>
<td>$P$</td>
<td>$\mathbb{N}$</td>
</tr>
<tr>
<td>6</td>
<td>Current Coding Position (integer)</td>
<td>$\text{pos}(i, j)$</td>
<td>$\mathbb{N}^2$</td>
</tr>
<tr>
<td>7</td>
<td>Plane Size (integer)</td>
<td>$\text{size}(m, n)$</td>
<td>$\mathbb{N}^2$</td>
</tr>
<tr>
<td>8</td>
<td>Current Code Block Band</td>
<td>$b$</td>
<td>$\mathbb{Z}_4$</td>
</tr>
<tr>
<td>9</td>
<td>Magnitude Plane (integer)</td>
<td>$v$</td>
<td>$\mathbb{N}^{m\times n}$</td>
</tr>
<tr>
<td>10</td>
<td>Current Output Buffer Length</td>
<td>$N$</td>
<td>$\mathbb{N}$</td>
</tr>
<tr>
<td>11</td>
<td>Current Output Buffer Pointer</td>
<td>$BP$</td>
<td>$\mathbb{N}^N$</td>
</tr>
</tbody>
</table>

**Table 7.1: Input and output of EBCOT algorithm**

Bold operators such as **EBCOTPlane** will be further expanded in the following description. Simpler operator such as EBCOTInit (in this case mostly it is just setting initial values for the input) will not be broken down.

EBCOTPlane := **CUP** ◦ PassReset ◦ **MRP** ◦ PassReset ◦ **SPP** ◦ PassReset ◦ PlaneReset

**CUP, MRP, SPP** corresponds to Cleanup Pass, Magnitude Refinement Pass and Significance Propagation Pass, and PassReset and PlaneReset does some cleanup (resetting position and pass independent states etc) between coding passes and bit-planes, respectively.

### 7.3.3 Coding Operations

In each of the EBCOT coding pass: CUP, MRP and SPP, the code plane is traversed according to the Scan Pattern, as described in Section 6.1.1. Certain coding operations are applied depending on the context of the current bit and states. There are four coding operations used in all the three passes: Zero Coding, Sign Coding, Magnitude Refinement Coding, and Run-Length Coding.

Zero Coding, Sign Coding and Magnitude Refinement Coding takes the 9-context values of the current bit and its states and generate one pair of context and decision value (CX, D), these coding operation will not affect the current
coding position. Run-Length Coding, on the other hand, takes up to four input sequential bits and produces one or three pairs of context and decision values. Naturally, Run-Length Coding will move the current coding bit to the last bit that it applies and the following coding operations (if any) is taken from there.

Each coding operations use part of the entire input (using operator $G$), pick up the value and/or the 9-context neighbours values (using $Gath_1$ or $Gath_9$), possibly pick one or many of them from the 9-context values (using $G_i$), perform some arithmetic operations or lookup table to get the $(CX, D)$ pair. The OL formula for the coding operations, their signature and breakdown rule are represented below, as corresponds to the formula description in section 6.1.2:

Zero Coding:

$$ZC(b, \sigma_9, v_1^p) : (Z_4 \times Z_2^9 \times Z_2) \rightarrow (N, Z_2)$$

$$ZC := (T_{ZC} \times I) \circ (I \times (G_3 + G_5) \times (G_1 + G_7) \times (G_0 + G_2 + G_6 + G_8) \times I) \circ (I \times \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \times I)$$

Zero Coding Wrapper that takes the full input:

$$ZC_W := ZC \circ (I \times Gath_9 \times Gath_1) \circ G([b, \sigma, pos, v_1^p, pos])$$

Sign Coding:

$$SC(\sigma_9, \chi_9) : (Z_2^9 \times Z_2^9) \rightarrow (N, Z_2)$$

$$SC := (I \times \text{xor}_2) \circ (T_{SC} \times I) \circ (H \times V \times I) \circ (L_4^2 \times G_4) \circ \left(\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \end{pmatrix}\right)$$

Where

$$H, V : (Z_2^9 \times Z_2^9) \rightarrow N$$

$$H := h \circ (f \times f) \circ (G_1 \times C_{-2} \times G_1 \times G_7 \times C_{-2} \times G_7) \circ L_4^2 \circ \left(\begin{pmatrix} 1 \\ 1 \end{pmatrix}\right)$$
\[ V := h \circ (f \times f) \circ (G_3 \times C_{-2} \times G_3 \times G_5 \times C_{-2} \times G_5) \circ L_2^3 \circ \left( \begin{array}{c} 1 \\ 1 \end{array} \right) \times \left( \begin{array}{c} 1 \\ 1 \end{array} \right) \tag{7.3} \]

Also
\[ f := \text{mul}_2 \circ (I \times \text{sub}_2) \circ (I \times C_1 \times \text{mul}_2) \]
\[ h := \text{min}_2 \circ (C_1 \times \text{max}_2) \circ (C_{-1} \times \text{sum}_2) \]

Sign Coding Wrapper:
\[ SC_W := SC \circ (Gath_9 \times Gath_9) \circ G([\chi, \text{pos}, \sigma, \text{pos}]) \]

Magnitude Refinement Coding:
\[ \text{MRC}(\sigma'_1, \sigma_9, v^P_1) : (\mathbb{Z}_2 \times \mathbb{Z}_2^9 \times \mathbb{Z}_2) \rightarrow (\mathbb{N} \times \mathbb{Z}_2) \]
\[ \text{MRC} := (T_{\text{MRC}} \times I) \circ (I \times (G_0 + G_1 + G_2 + G_3 + G_5 + G_6 + G_7 + G_8) \times I) \]

Magnitude Refinement Coding Wrapper:
\[ \text{MRC}_W := \text{MRC} \circ (Gath_1 \times Gath_9 \times Gath_1) \circ G([\sigma', \text{pos}, \sigma, \text{pos}, v^P, \text{pos}]) \]

RLC is basically a few looped conditions or can be implemented as a switch, so it’s defined as an OL operator but use a template to map to the implementation.
\[ \text{RLC} : (v^P_4) \rightarrow (\mathbb{N} \times \mathbb{Z}_2)^n \]

where \( n \) could be 1 or 3

### 7.3.4 Coding Passes

In each coding pass, bits in the current bit-plane are traversed according to the scan pattern. Checks are made to determine whether and which coding operations are applied.

**Significance Propagation Pass**

\[ \text{SPP} := \prod_{i=0}^{m+n} \text{NextBit} \circ \text{SPPCode} \]
Figure 7.1: Significance Propagation Pass and Magnitude Refinement Pass Formula.
SPPCode := \langle (\text{Stat}_\sigma \circ \text{SC}_W, \text{Noop}) \circ \text{Enc} \circ \text{Stat}_\eta \circ \text{ZC}_W, \text{Noop} \rangle_{\text{CW}_{\text{SPP}}}

NextBit calculate and update the current coding bit position \( pos \) according to the scan pattern. \( \text{Stat}_{syb} \) set the status by changing the current position in status plane \( syb \) to be 1. \( \text{CW}_{\text{SPP}} \) checks whether SPP needs to be applied and \( \text{Enc} \) determines whether last decision bit is 1 or not.

\[
\text{CW}_{\text{SPP}} := \text{C}_{\text{SPP}} \circ \text{Gath}_9 \circ \text{G}([^\sigma, pos])
\]

\[
\text{C}_{\text{SPP}} (\sigma_9) : \mathbb{Z}_2^9 \rightarrow \mathbb{Z}_2
\]

\[
\text{C}_{\text{SPP}} := \text{and}_2 \circ (\text{eq} \times \text{neq}) \circ (C_0 \times I \times C_0 \times I) \circ (G_4 \times (G_0 + G_1 + G_2 + G_3 + G_5 + G_6 + G_7 + G_8)) \circ \left( \begin{array}{c} 1 \\ 1 \end{array} \right)
\]

Figure 7.1 shows the flow chart for Significance Propagation Pass and Magnitude Refinement Pass with operators instead of formula in Figure 6.3 and Figure 6.4.

**Magnitude Refinement Pass**

\[
\text{MRP} := \prod_{i=0}^{m \times n} \text{NextBit} \circ \text{MRPCode}
\]

MRPCode := \langle (\text{Stat}_{\sigma'}, \text{MRC}_W, \text{Noop}) \rangle_{\text{CW}_{\text{MRP}}}

\[
\text{CW}_{\text{MRP}} := \text{C}_{\text{MRP}} \circ (\text{Gath}_1 \times \text{Gath}_1) \circ \text{G}([^\sigma, pos, \eta, pos])
\]

\[
\text{C}_{\text{MRP}} (\sigma_1, \eta_1) : (\mathbb{Z}_2 \times \mathbb{Z}_2) \rightarrow \mathbb{Z}_2
\]

\[
\text{C}_{\text{MRP}} := \text{and}_2 \circ (\text{eq} \times \text{eq}) \circ (C_0 \times I \times C_0 \times I)
\]

**Cleanup Pass**
Figure 7.2: Cleanup Pass Formula.
\[ CUP := \prod \langle \text{CUPCode} \rangle_{\text{NextBit}} \]

\[ \text{CUPCode} := \langle \langle \text{Stat}_\sigma \circ \text{SC}_W \rangle_{\text{Enc}} \circ \langle \text{RLC}, \text{ZC}_W \rangle_{\text{CW}_{\text{CUPBranch}}, \text{Noop}} \rangle_{\text{CW}_{\text{CUP}}} \]

\[ \text{CW}_{\text{CUP}} := \text{C}_{\text{CUP}} \circ (\text{Gath}_1 \times \text{Gath}_1) \circ \mathcal{G}([v^P, pos, \sigma', pos]) \]

\[ \text{C}_{\text{CUP}}(v_1^P, \sigma_1') : (\mathbb{Z}_2 \times \mathbb{Z}) \rightarrow \mathbb{Z}_2 \]

\[ \text{C}_{\text{CUP}} := \text{and}_2 \circ (\text{eq} \times \text{eq}) \circ (C_0 \times I \times C_0 \times I) \]

\[ \text{CW}_{\text{CUPBranch}} \text{ determines whether RCL or ZC is to be applied} \]

\[ \text{CW}_{\text{CUPBranch}} := \text{C}_{\text{CUPBranch}} \circ (\text{Gath}_9 \times \text{Gath}_9 \times I) \]

\[ \circ (I \times \text{Next}_3 \times I \times I \times I) \circ \mathcal{G}([v^P, pos, v^P, pos, j]) \]

Next\(_3\) moves the coordinates ahead by 3.

\[ \text{Next}_3 := \text{NextBit} \circ \text{NextBit} \circ \text{NextBit} \]

\[ \text{C}_{\text{CUPBranch}}(v_9^P, v_9^P, j) : (\mathbb{Z}_2^9 \times \mathbb{Z}_2^9 \times \mathbb{Z}) \rightarrow \mathbb{Z}_2 \]

\[ \text{C}_{\text{CUPBranch}} := \text{and}_3 \circ (\text{eq} \times \text{eq} \times \text{eq}) \]

\[ \circ (C_0 \times \text{Gath}_{\text{ACC}} \times C_0 \times \text{Gath}_{\text{ACC}} \times C_0 \times \text{mod}) \circ (I \times I \times I \times C_4) \]

\[ \text{Gath}_{\text{ACC}} := (G_0 + G_1 + G_2 + G_3 + G_5 + G_6 + G_7 + G_8) \]

### 7.4 Optimized Implementations

The first implementation directly derived from the interpretation of the standard to guarantee the possibility of the generating Tier-1 code using Operator Language. It also verify the correctness of the interpretation and the algorithm. The initial performance is measured to be worse than JasPer implementation.

There are mainly two reasons for the poor initial performance. First, there is no optimization so far for the algorithm directly interpreted from the JPEG 2000 standard specification, thus the memory access, data layout are default and mediocre. The second reason is that on the implementation level the optimization such as copy propagation and SSA form is not fully supported, particularly for the code generated by selection and conditional iteration operators. The optimization in code level is added but not elaborated in the thesis. Algorithm level optimizations and the corresponding alternative implementations are described as follows.
7.4.1 Packed data

It is obviously seen from the general input and output signature (Table 7.1) that the sign plane, magnitude bit plane and both three state variables only needs to be two dimensional array of bits while in the first implementation, they are stored as integers. Loading and storing them definitely waste much of the memory bandwidth than it is actually needed. It is possible to pack the bit value and the three status values into one integer.

This is done by first change the input and output signature (data type) to change the data type. Then the implementation of \( G \) is changed so that the orginal return of \( G \) is appended with the binary and with some constant to extract what is needed from the packed value. This optimization trades arithmetic computation with memory access and would potentially yield better performance on microprocessors running on algorithms where the bottleneck is memory bandwidth rather than computation.

7.4.2 Improved Coding Operations by Lookup

Coding operations only use part of the input defined in table 7.1 - the entire input of the EBCOT algorithm. As most of the status are just a few bits, it’s observed that for coding operations it might be more efficient to do a direct lookup on a precomputed table with the linearized value of all the possible bits and states.

The result is that rather than having Sign Coding be computed by formulas such as (7.1), (7.2) and (7.3) with much computation then a final look up, but rather use a direct lookup table. The OL formula for Sign Coding will become (7.4).

\[
SC(\sigma_9, \chi_9) : (\mathbb{Z}_2^9 \times \mathbb{Z}_2^9) \rightarrow (\mathbb{N}, \mathbb{Z}_2)
\]

\[
SC := (I \times \text{xor}_2) \circ (T'_{SC,CX} \times T'_{SC,D} \times I) \circ (L_2^4 \times G_4) \circ \left( \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \right)
\]

(7.4)

Where

\[
T'_{SC,CX}(\sigma_9, \chi_9) : (\mathbb{Z}_2^9 \times \mathbb{Z}_2^9) \rightarrow \mathbb{N}
\]

\[
T'_{SC,D}(\sigma_9, \chi_9) : (\mathbb{Z}_2^9 \times \mathbb{Z}_2^9) \rightarrow \mathbb{Z}_2
\]
Using the packed bit and status implementation mentioned above, there are still potentially 9 accesses (for 9-context neighbours) needed for each coding operation, a bit less than the previous $9n$ access, in which $n$ being the number of bit/status that is needed for the coding operation.

However, the problem with lookup is that the index need to be extracted from different parts of the possible 9 values using bit-operators and reassembled, that would be potentially causing a lot of arithmetic operations. It’s not know which alternative would be giving a better performance (heavily depends on the arithmetic/memory access amount tradeoff and performance comparison of them). But including this alternative would allow SPIRAL’s searching ability to get the better performance algorithm.

### 7.4.3 Exchange write for read by incorporating neighbours

The previous optimization reads the possible 9-context values, extract the bits to form the index, and do the lookup in the pre-computed index tables for the context (CX) and decision (D) values. This reduces in the memory read but increase the computation for the index. One potential way of even reducing the memory load without much increasing the computation overhead is to pack the bits that is needed for each lookup together. Then there is only one instruction for doing the bit operation to extract the index and possibly another for shifting to get the index.

The problem for the potential implementation is that, as the bits needed for the look up contain the neighbour bit/status, thus, each packed value of the current encoding position need to include the corresponding bits needed of the 9 neighbours. There are two problems with that: firstly, packing the neighbour might exceed the integer boundary (32 bits), that could be solved by using use multiple consecutive integers or even the SSE register (128 bits) to hold the status data. The second problem is that there are possible updates of the status in all of the three coding passes, since the status is now included in all of the 9 neighbours, the update has to be applied on all of the 9 context values - as the previously update to the status always apply on the same bit as the one the encoding is being done. An interesting observation to the algorithm is updates definitely happen less frequently than the checking or the coding operation, which need to load data. Thus changing to updating 9 values (write) is likely to be better than loading 9 values (read). Plus, all the status value always go from 0 to 1 and never changes back, so no checking of the status is needed, update is just a binary or with some constant.

Analysis is made to all the coding passes and coding operations about which states or values are needed for each of them (Table 7.2 and 7.3). The final packed
<table>
<thead>
<tr>
<th>Coding Pass</th>
<th>Condition</th>
<th>Selection</th>
<th>Coding Operation</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPP</td>
<td>$\sum \sigma_8 &gt; 0, \sigma_1 = 0$</td>
<td>$\times$</td>
<td>ZC - SC</td>
<td>$\eta_1, \sigma_1$</td>
</tr>
<tr>
<td>MRP</td>
<td>$\sigma_1 \neq 0, \eta_1 = 0$</td>
<td>$\times$</td>
<td>MRC</td>
<td>$\sigma_1$</td>
</tr>
<tr>
<td>CUP</td>
<td>$\sigma_1 = 0, \eta_1 = 0$</td>
<td>$j, \sigma_9$</td>
<td>RLC - SC or ZC - SC</td>
<td>$\sigma'_1$</td>
</tr>
</tbody>
</table>

Table 7.2: Coding Pass Context Lookup

<table>
<thead>
<tr>
<th>Coding Operation</th>
<th>Read States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Length Coding</td>
<td>$v_4^P$</td>
</tr>
<tr>
<td>Zero Coding</td>
<td>$b, \sum H_\sigma, \sum V_\sigma, \sum D_\sigma$</td>
</tr>
<tr>
<td>Sign Coding</td>
<td>$\chi_4, \sigma_4$ bitwise</td>
</tr>
<tr>
<td>Magnitude Refinement Coding</td>
<td>$\sigma'_1, \sum \sigma_8$</td>
</tr>
</tbody>
</table>

Table 7.3: Coding Operation

state that include all the neighbour status needed for the coding operators can be find in Table 7.5, it could be fit to a 32 bit integer. The neighbour status values are represented by coordination N(orth), S(outh), E(ast) and W(est) as in Table 7.4. The constants for updating the $\sigma$ can be found in Table 7.6.

7.4.4 Use SSE for update status

Once the 9 context values are included as a packed status, according to the previous optimization, update of states has to be applied on the 9 values. These values are formed in the 3 by 3 matrix and need 9 memory reads, 9 binary or operators and 9 memory writes.

There is a potential optimization for the update of status, that is, by using the Streaming SIMD Extensions provided with Intel Processors. Starting from SSE2, Intel has introduced intrinsic that can load, write and do some basic operators (including the binary or intrinsic) in the whole 128 bit - that is, 4 consecutive 32-bit integers - in one instruction.

The difference of this optimization is that it is not on algorithm level but only on the code generation level, so it is put in a special code generator that, also

<table>
<thead>
<tr>
<th>$\sigma_{NW}$</th>
<th>$\sigma_N$</th>
<th>$\sigma_{NE}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_{W}$</td>
<td>$\sigma$</td>
<td>$\sigma_E$</td>
</tr>
<tr>
<td>$\sigma_{SW}$</td>
<td>$\sigma_S$</td>
<td>$\sigma_{SE}$</td>
</tr>
</tbody>
</table>

Table 7.4: 4, 8/9-context neighbours defined by coordination (for $\sigma$)
<table>
<thead>
<tr>
<th>Index</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>χ</td>
<td>σ</td>
<td>η</td>
<td>σ'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>σW</td>
<td>σS</td>
<td>σE</td>
<td>σN</td>
<td>σNW</td>
<td>σSW</td>
<td>σSE</td>
<td>σNE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>χW</td>
<td>χS</td>
<td>χE</td>
<td>χN</td>
<td>σW</td>
<td>σS</td>
<td>σE</td>
<td>σN</td>
</tr>
</tbody>
</table>

**Table 7.5**: Packed State and Data

| pos | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| σNW | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |
| σN  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  |    |    |    |    |    |    |    |    |
| σNE | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  |    |    |    |    |    |    |    |    |
| σW  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  |    |    |    |    |    |    |    |    |
| σ   | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |    |    |    |    |    |    |    |    |
| σSE | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  |    |    |    |    |    |    |    |    |

**Table 7.6**: 9 constants for update of σ, showing only lower 24 bits
as an alternative, could generate code with these SSE instructions.

The following code is what is generated for the update in the first three elements in a row. For each location, one address computation, one constant load, one memory load, one logical operation and one store is needed.

```c
a123 = (a118 + (-69));
a124 = Y1[a123];
s10 = ((a124)|(512));
Y1[a123] = s10;
a125 = (a118 + (-68));
a126 = Y1[a125];
s11 = ((a126)|(16388));
Y1[a125] = s11;
a127 = (a118 + (-67));
a128 = Y1[a127];
s12 = ((a128)|(1024));
Y1[a127] = s12;
```

And the following code is the corresponding SSE instruction, for the three locations, one address computation, one vector memory load, one vector constant load, one vector logical operation and one vector store is needed. Thus, in terms of instruction count, it is one third the cost of the previous implementation.

```c
__m128i c1, s15, s16;
c1 = _mm_set_epi32(512,16388,1024,0);
...
a123 = (a118 + (-69));
s15 = _mm_loadu_si128((__m128i*)&Y1[a123]);
s16 = _mm_or_si128(c1, s15);
_mm_storeu_si128((__m128i*)&Y1[a123], s16);
```

In reality, however, the vector implementation might not necessarily be faster than the scalar implementation. One of the most significant reason is that the unaligned load and store, which doesn’t restrict the address have to be 16-byte aligned but could be slower even than normal loads.

### 7.4.5 Unroll

The current implementation handle the Scan Pattern, described in Section 6.1.1, using a codelet that matches the Operator `NextBit` as below.


```plaintext
int i1, i2;
i1 = (X[0] - 1);
i2 = (X[1] - 1);
if ((((i2 % 4) < 3))) {
i2 = (i2 + 1);
} else {
i2 = ((i2 / 4)*4);
i1 = (i1 + 1);
if (((i1 >= 64))) {
i2 = (i2 + 4);
i1 = 0;
}
}
Y[0] = (i1 + 1);
Y[1] = (i2 + 1);
```

This piece of codelet is relatively short but it would have quite a few conditional jumps without much of the computations in each of the branch. This
would likely flush the pipeline of the processor constantly and deteriorate the performance.

The solution for this problem is that use a unrolling technique similar to loop unrolling. Unrolling in general reduce the cost of doing loop but increase the code size and potential register use. Moreover, unrolling in EBCOT have extra benefit and is not trivial in the algorithm level. Take Significant Propagation Pass for example.

\[
SPP := \prod_{i=0}^{m \cdot n} \text{NextBit} \circ \text{SPPCode} \hspace{1cm} (7.5)
\]

\[
SPP := \prod_{i=0}^{m \cdot n/4} \text{NextBit} \circ \text{SPPCode} \circ \text{NextRow} \circ \text{SPPCode} \circ \text{NextRow} \circ \text{SPPCode} \hspace{1cm} (7.6)
\]

\[
SPP := \prod_{i=0}^{n/4} \text{NextStrip} \circ \prod_{j=0}^{m} \text{NextCol} \circ \text{SPPCode} \circ \text{NextRow} \circ \text{SPPCode} \circ \text{NextRow} \circ \text{SPPCode} \hspace{1cm} (7.7)
\]

Formula (7.5) is the original implementation that include the codelet above. And the Formula (7.6) unroll only the 4 consecutive rows, and Formula (7.7) unrolls to the strip - that is, every 4 rows of the code-block. After that, there is still a freedom of unrolling, but that is up to the code level option and the compiler. The \text{NextRow}, \text{NextCol} and \text{NextStrip} here doesn’t contain any conditional statement thus could also make the performance potentially better.

Unrolling in the algorithm level gives the flexibility of choosing the trade off between speed and code size, but in this case also eliminate some conditional codelet that could be simplified through unrolling to a certain level. As before, this unrolling is also just provided as some choice so that the compiler can search and choose the best performing one on each platform.

### 7.4.6 Parallelization

In EBOCT, each code-block is independently encoded, therefore Tier-1 coding (EBCOT with MQ-coder) is naturally parallelized. Pthread, WinThread and OpenMP parallelization are added in the modified JasPer encoder.
Experimental Results

The final generated encoder is plugged into the reference implementation JasPer, and is tested against the original JasPer itself and the Intel IPP performance library. The benchmark platform is set to be Intel Core 2 Duo and Extreme processors with SSE Extension. Intel C++ Compiler with O3 optimization is used for all the three libraries as it provides more code to assembly optimization than GCC.

Two sample pictures of different image complexity are used for the benchmark, each has two different sizes (512 by 512 and 256 by 256).

Table 8.1 and Figure 8.1 shows the encoding time in millisecond of JasPer, IPP and generated encoder over the four benchmark pictures on a Core 2 Extreme 3 GHz using 2 threads on Windows XP 64 bit edition. Table 8.2 and Figure 8.2 shows the encoding time in millisecond of JasPer, IPP and generated encoder over the four benchmark pictures on a Core 2 Extreme 2.4 GHz using 2 threads on Windows XP. It’s seen that the generated encoder is mostly better than that of IPP by a margin of few percent.

Table 8.3 shows the runtime comparison between JasPer, IPP, and generated encoder using 1 or 2 Thread on Core 2 Extreme 3.0 GHz processor with parallelized EBCOT using OpenMP. Note that JasPer is not parallelized thus there is no 2 Thread version available.
<table>
<thead>
<tr>
<th></th>
<th>bg512</th>
<th>scene512</th>
<th>bg256</th>
<th>scene256</th>
</tr>
</thead>
<tbody>
<tr>
<td>JasPer</td>
<td>90.1</td>
<td>142.9</td>
<td>25.4</td>
<td>38.6</td>
</tr>
<tr>
<td>IPP</td>
<td>46.0</td>
<td>75.7</td>
<td>13.5</td>
<td>21.0</td>
</tr>
<tr>
<td>Generated</td>
<td>49.0</td>
<td>74.9</td>
<td>13.4</td>
<td>20.3</td>
</tr>
<tr>
<td>Gen/JasPer</td>
<td>-45.6%</td>
<td>-47.6%</td>
<td>-47.2%</td>
<td>-47.4%</td>
</tr>
<tr>
<td>Gen/IPP</td>
<td>+6.5%</td>
<td>-1.1%</td>
<td>-0.7%</td>
<td>-3.3%</td>
</tr>
</tbody>
</table>

Table 8.1: Runtime and Comparison with Jasper and IPP using WinThread on 3.0GHz Core 2 Extreme

<table>
<thead>
<tr>
<th></th>
<th>bg512</th>
<th>scene512</th>
<th>bg256</th>
<th>scene256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jasper</td>
<td>102.5</td>
<td>164.9</td>
<td>28.9</td>
<td>44.9</td>
</tr>
<tr>
<td>IPP</td>
<td>57.2</td>
<td>92.7</td>
<td>16.7</td>
<td>25.5</td>
</tr>
<tr>
<td>Generated</td>
<td>56.5</td>
<td>86.8</td>
<td>16.1</td>
<td>23.5</td>
</tr>
<tr>
<td>Gen/Jasper</td>
<td>-44.9%</td>
<td>-47.3%</td>
<td>-44.3%</td>
<td>-47.7%</td>
</tr>
<tr>
<td>Gen/IPP</td>
<td>-1.2%</td>
<td>-6.4%</td>
<td>-3.6%</td>
<td>-7.8%</td>
</tr>
</tbody>
</table>

Table 8.2: Runtime and Comparison with Jasper and IPP using WinThread on 2.4GHz Core 2 Duo

Figure 8.1: JPEG 2000 Runtime Comparison with JasPer and IPP using WinThread on 2.4GHz Core 2 Duo
Figure 8.2: JPEG 2000 Runtime Comparison with JasPer and IPP using WinThread on 3.0GHz Core 2 Extreme

Table 8.3: Runtime and Comparison with Jasper and IPP using OpenMP
The aim of the project is to create a performance library for JPEG 2000 using the method of automatic code generation and performance tuning from a higher level language. The work is based on the SPIRAL project developed by Carnegie Mellon University, the high level language used is the Operator Language, the extension of the Signal Processing Language.

Firstly, the complicated JPEG 2000 program is analysed and profiled. The emphasis of the optimization is put on the EBCOT algorithm used in the entropy coding stage by JPEG 2000.

New operations with new rewrite rules are then introduced into the Operator Language. The initial implementation of EBCOT is generated with these new operators and the correctness is verified. Code level compiler optimization is enhanced for the code generated by the operations.

The prototype implementation is directly derived from what the JPEG 2000 standard specified with straightforward interpretation. With high level algorithm knowledge, new breakdown rule and rewrite rules are added so that multiple implementations can be searched to find the best implementation on a specific platform. By combining application specific breakdown rules, rewrite rules and compile strategy, it’s possible to control the algorithm to take the form which fits the hardware architecture better and get high performance library.
The final generated encoder is plugged into the open source reference implementation JasPer and the encoding time is compared against the Intel performance library, which is most likely to be hand-tuned. The result showed that by only optimizing part of the JPEG 2000 encoding algorithm we can achieve similar or even higher performance than a Intel’s commercial performance library.

The advantages of using program generation approach to implement complex algorithm such as EBCOT can be easily seen from this thesis. First, the high level language (Operator Language in this case) separates the memory access (Gather), the arithmetic operators (Coding Operations and checks for coding operations) and the algorithm specific structure (Selects and Conditional iterative), changes of each of these does not necessarily affect the others. For example, the change of data layout only need the change of the memory access operator Gather. Secondly, algorithm level rewrite is used to add an alternative implementation and the search will find the best implementation for each platform, that could be seen from the from the Unroll optimization and update state by SSE. Besides, automatic code generation and performance tuning will be able to be applied to more complex algorithms than EBCOT while it would be quite difficult and require much effort to tune such algorithm by human.
A.1 Generated Code Snippet for MRP, block size (64,64)

Y7_0 = 1;
Y7_1 = 1;
for(int i810 = 0; i810 <= 15; i810++) {
    for(int i809 = 0; i809 <= 63; i809++) {
        int a2547, a2548, a2549, a2550, a2551, a2552, a2553
            , a2554, a2555, a2556, a2557, a2558, a2559, a2560, a2561
            , a2562, a2563, a2564, a2565, a2566, a2567, a2568, a2569
            , a2570, a2571, a2572, a2573, a2574, a2575, a2576, a2577
            , a2578, a2579, a2580, a2581, a2582, a2583, a2584, a2585
            , a2586, a2587, a2588, a2589, a2590, a2591, a2592, a2593
            , a2594, a2595, s393, s394, s395, s396, s397, s398
            , s399, s400, s401, t382, t383, t384, t385, t386
            , t387, t388, t389, t390, t391, t392, t393, t394;

        a2547 = Y7_1;
a2548 = (66*a2547);
a2549 = Y7_0;
a2550 = (a2548 + a2549);
a2551 = Y1[a2550];
a2552 = ((a2551)&(262144));
a2553 = ( !(a2552) );
a2554 = ((a2551)&(1048576));
t382 = ((a2553) && (a2554));
if (((t382 != 0))) {
    a2555 = ((a2551)&(130816));
    a2556 = ((a2555) >> (8));
    t383 = nmagctable[a2556];
    a2557 = ((a2551)&(plane\text{pos}0ne));
    t384 = ((a2557 != 0));
    spiral_jpc\_mqenc\_setcur\text{ctx}(enc, t383);
    spiral_jpc\_mqenc\_put\text{bit}(enc, t384);
    s393 = ((a2551)|(65536));
    Y1[a2550] = s393;
} else {
}

a2558 = Y7\_1;
s394 = (1 + a2558);
Y7\_1 = s394;
a2559 = Y7\_0;
a2560 = (66*a2559);
a2561 = Y7\_0;
a2562 = (a2560 + a2561);
a2563 = Y1[a2562];
a2564 = ((a2563)&(262144));
a2565 = ( ! (a2564) );
a2566 = ((a2563)&(1048576));
t385 = ((a2565) && (a2566));
if (((t385 != 0))) {
    a2567 = ((a2563)&(130816));
    a2568 = ((a2567) >> (8));
    t387 = nmagctable[a2568];
    a2569 = ((a2563)&(plane\text{pos}0ne));
    t388 = ((a2569 != 0));
    spiral_jpc\_mqenc\_setcur\text{ctx}(enc, t387);
    spiral_jpc\_mqenc\_put\text{bit}(enc, t388);
    s395 = ((a2563)|(65536));
    Y1[a2562] = s395;
} else {
}

a2570 = Y7\_1;
s396 = (1 + a2570);
Y7\_1 = s396;
a2571 = Y7\_1;
a2572 = (66*a2571);
a2573 = Y7\_0;
a2574 = (a2572 + a2573);
a2575 = Y1[a2574];
a2576 = ((a2575)&(262144));
a2577 = ( ! (a2576) );
a2578 = ((a2575)&(1048576));
t389 = ((a2577) && (a2578));
if (((t389 != 0))) {
  a2579 = ((a2575)&&(130816));
a2580 = ((a2579) >> (8));
t390 = mmagctable[a2580];
a2581 = ((a2575)&&(planeposOne));
t391 = ((a2581 != 0));
spiral_jpc_mqenc_setcurctx(enc, t390);
spiral_jpc_mqenc_putbit(enc, t391);
s397 = ((a2575)|(65536));
Y1[a2574] = s397;
} else {
  a2582 = Y7_1;
s398 = (1 + a2582);
Y7_1 = s398;
a2583 = Y7_1;
a2584 = (66*a2583);
a2585 = Y7_0;
a2586 = (a2584 + a2585);
a2587 = Y1[a2586];
a2588 = ((a2587)&&(262144));
a2589 = (!a2588);
a2590 = ((a2587)&&(1048576));
t392 = ((a2589) && (a2590));
if (((t392 != 0))) {
  a2591 = ((a2587)&&(130816));
a2592 = ((a2591) >> (8));
t393 = mmagctable[a2592];
a2593 = ((a2587)&&(planeposOne));
t394 = ((a2593 != 0));
spiral_jpc_mqenc_setcurctx(enc, t393);
spiral_jpc_mqenc_putbit(enc, t394);
s399 = ((a2587)|(65536));
Y1[a2586] = s399;
} else {
  a2594 = Y7_0;
s400 = (1 + a2594);
Y7_0 = s400;
a2595 = Y7_1;
s401 = (a2595 - 3);
Y7_1 = s401;
}

Y7_0 = 1;
Y7_1 = (4 + Y7_1);
Bibliography


