AUTOMATIC GENERATION OF ENTROPY CODING PROGRAMS USING FLAVOR

Yihan Fang and Alexandros Eleftheriadis
Department of Electrical Engineering
Columbia University
New York, NY 10027, USA

Abstract - Flavor (Formal Language for Audio-Visual Object Representation) is a new programming language for media-intensive applications. It extends the typing system of C++ and Java to incorporate bitstream representation semantics. This allows describing in a single place both the in-memory representation of data as well as their bitstream-level (compressed) representation as well. In this paper, we present our approach of hybrid variable length code segmentation in program code generation for parsing entropy-coded data from the bitstream. We also show that this approach produces a very good tradeoff between space (memory required) and time (decoding speed).

1. INTRODUCTION

Multimedia applications are being developed intensively to provide richer features and more functionalities for end-users. In order to facilitate this process, it is essential to simplify and speed up the development of software that processes coded multimedia information. This includes audio and video encoders/decoders, as well as applications that manipulate such information (e.g., for editing, content creation, multimedia indexing and searching [7], [8]). Such information is invariably encoded in a highly efficient form, to minimize the cost of storage and transmission. This source coding [1] operation is almost always performed in a bitstream-oriented fashion: the data to be represented is converted to a sequence of binary values of arbitrary (and typically variable) lengths, according to a specified syntax. The syntax itself can have various degrees of sophistication and is not easily decoded using general-purpose programming languages.

Flavor is a novel programming language specifically designed to formally describe the bitstream syntax for multimedia applications. Flavor can be used to describe the bitstream of any multimedia standards or specification, including GIF, JPEG, and MPEG-1 or MPEG-2 [2][6]. Currently, Flavor is used in the MPEG-4 standardization effort for the description of the bitstream syntax [3][4][6]. Emerging multimedia representation techniques can directly use Flavor to represent bitstream syntax in their specifications. This would bring the advantage of direct access to compressed multimedia information by application developers with essentially zero programming using a software tool that processes Flavor code and generates regular C++ or Java code. Another advantage is that such software tools can automatically generate optimized code for Huffman encoding/decoding, a very common tool in media representation.

2. OVERVIEW OF FLAVOR

Flavor provides a formal way to specify how data is laid out in a serialized bitstream. It is based on a *principal of separation* between bitstream parsing operations and encoding, decoding and other operations. This separation acknowledges the fact that the same syntax can be utilized by different tools, but also that the same tool can work unchanged with a different bitstream syntax.

Flavor was designed to be an intuitive and natural extension of the typing system of object-oriented languages like C++ and Java, anticipating an audiovisual world comprised of audiovisual objects, both synthetic and natural, and combining it with well-established paradigms for software design and implementation.

Flavor has been explicitly designed to follow (as much as possible) a declarative approach to bitstream syntax specification. Developers specify how the data is laid out on the bitstream, and do not detail a step-by-step procedure that parses it.

In the following we briefly go through a few features of Flavor that differ from traditional programming languages. For a complete description of Flavor, see [5] [9] or visit the Flavor web site at `http://www.ee.columbia.edu/flavor`.

- **Parsable Variable Declarations**

Parsable variables are the core of Flavor's design; it is the proper definition of these variables that defines the bitstream syntax. Parsable variables include a parse length specification immediately after their type declaration (e.g., '`int(3) a;`').

- **Flow Control Constructs**

Flavor provides a wide range of facilities to define sophisticated bitstreams, including `if-else`, `switch`, `for`, and `while` constructs. In contrast with regular C++ or Java, these are all included in the data declaration part of the class. This is in line with the declarative nature of Flavor, where the focus is on defining the structure of the data, not operations on them.

- **Classes**

Flavor classes are the fundamental type structure, as in other object-oriented programming languages. The parsable objects defined in a class will be present in the bitstream in the same order they are declared. A class is considered parsable if it contains at least one variable that is parsable. Flavor classes are able to access external information by using *parameter types*. These act the same way as formal arguments in function or method declarations do. Flavor supports polymorphism for parsable classes by introducing the concept of *object identifier*, which is a unique parsable variable associated with each class.

- **Maps**

A wide variety of representation schemes, rely heavily on entropy coding, and in particular Huffman codes [1]. These are variable length codes (VLCs) which are uniquely decodable (no codeword is the prefix of another). Flavor provides extensive support for variable length coding through the use of *maps*, which are declarations of tables in which the 1-to-*n* correspondence between codewords and values is described. The following is a simple example of a map declaration.

```
map A(int) {
  0b0,  1,        // Note: 0b0 is a bit string of length 1
  0b10, 2,
  0b11, int(5)    // escape: read 5 more bits to get value
}
```

Figure 1: A simple map declaration.

The `map` keyword indicates the declaration of a map. A map contains a series of entries, each starting with a bit string that declares the codeword of the entry followed by the value to be assigned to this codeword. A map can assign value(s) to a fundamental type, a class type, or an array. Flavor maps also support "escape codes," which are useful when the alphabet encoded using the Huffman codewords is too large to be able to assign a codeword to every source symbol. In this case, instead of a value, one (or more) of the bit strings is associated with a parsable type specification (e.g., `int(5)`) rather than a specific value. Figure 1 defines a map named `A` with return type `int`. After the map is properly declared, we can define parsable variables that use it by indicating the name of the map where we would put the parse size expression. For example, we can define a parsable integer 'i' whose value is obtained from the bitstream according to map A simply by: `int(A) i;`

3. THE FLAVOR TRANSLATOR

We have developed a translator that evolved concurrently with the design of the language. It translates Flavor source to regular C++ or Java code. The translator reads a Flavor source file and generates a set of C++ or Java file(s) that contain declarations of all Flavor classes. The translator also generates for each class a `put()` and a `get()` method. The `get()` method is responsible for reading a bitstream and loading the class variables with their appropriate values, while the `put()` method does the reverse. For parsing operations, the only task required by the programmer is to declare an object of the class type at hand, and then call its `get()` method with an appropriate bitstream. While the same is also true for the `put()` operation, the application developer must also load all class member variables with their appropriate values before the call is made.

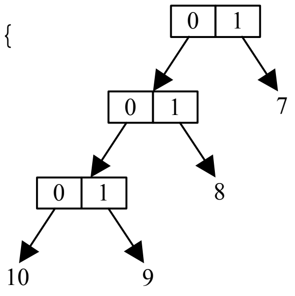
4. MAP PROCESSING: VLC PARSER GENERATION

Map processing is one of the most useful features of Flavor, as hand-coding variable length code tables is tedious and error prone. By using the Flavor translator, optimization of parsing VLCs is performed at zero cost. Maps can also be used for fixed-length code mappings just by making all codewords have the same length. As a result, one can very easily switch between fixed and variable-length mappings when designing a new representation format.

Traditional ways of parsing entropy codes normally follow one of two techniques (or a mix thereof). The first is to use a binary tree, which is traversed one bit at a time. Assume that the length of the longest codeword is *N* bits. Obviously, this approach—although conceptually simple—is extremely slow, requiring *N* stages of bitstream input and lookup. This is shown in Figure 2(b), where the variable length coding table is defined in Figure 2(a).

```
map Example(int) {
     0b1,      7,
     0b01,     8,
     0b001,    9,
     0b000,  10
}
```

(a)



(b)

Index	Length	Value
000	3	10
001	3	9
010	2	8
011	2	8
100	1	7
101	1	7
110	1	7
111	1	7

(c)

Figure 2: Two decoding schemes: (a) VLC map definition (b) single-bit binary tree (*N* stages, 1 bit per stage), (c) single lookup table (1 stage, *N* bits per stage).

On the other end of the spectrum, the fastest possible decoding approach is obtained by using a single lookup table. The size of the table is 2^N, where *N* is the length of the longest codeword. Each row of the table contains two entries: the value associated with that codeword (if any), and the length of the codeword. As some codewords may have length *M* smaller than *N*, we consider a row as corresponding to a codeword depending on if the first *M* bits of its index match the *M* bits of the codeword. As a result, a codeword is associated with 2^{N-M} entrie(s). This scenario is shown in Figure 2(c). The cost of decoding would be a single look-ahead read of the input bitstream using *N* bits and then a lookup in the table using the input value as the index to the table and finally skipping the number of bits of the matched code length.

The drawback of the single-stage approach is, of course, the big memory requirements for the lookup table. For example, MPEG-1 and MPEG-2 variable-length code tables have a maximum codeword length of 32 bits. This would require a table with 2^{32} entries, which is impractical. The majority of the entries would be duplicates, due to the unique decodability requirements of a table. For example, the presence of a single-bit codeword would mean that half of the table's entries—those starting with the same bit value—would all be associated with the same entry.

A compromise between these two extremes is a combination of smaller lookup tables, organized in a hierarchical fashion. However, it is impossible to find an optimal partitioning strategy for a map, so that the number of steps is minimized while the total memory requirements are kept within a given bound, unless we do an exhaustive search (it is easy to construct counterexamples that demonstrate that incremental optimization is not possible). One approach is to use fixed-length multi-bit lookup tables, with a fixed step length. This combines the efficiency of multi-bit lookups, and the lower memory requirements of a hierarchical lookup scheme. However, there can still be substantial penalty in terms of wasted lookup space, since the tables are not customized for each map.

We adopt a hybrid approach, that maintains the space efficiency of binary tree decoding, and most of the speed associated with lookup tables. In particular, instead of using lookup tables we use nested `switch` statements. Each time the read-ahead size is determined by the maximum of a fixed step size and the size of the next shortest code. The fixed step size is used to avoid degeneration of the algorithm into binary tree decoding. The benefit of this approach is that only complete matches require `case` statements, while all partial matches can be grouped into a single `default` statement (that, in turn, introduces another `switch` statement).

The space requirements consist of storage of the `case` values and the comparison code generated by the compiler (this code consists of just 2 instructions on typical CISC systems, e.g., a Pentium). While slightly larger than a simple binary tree decoder, this overhead still grows linearly with the number of code entries (rather than exponentially with their length). This is further facilitated by the selection of the step size. When the incremental code size is small, multiple case statements may be assigned to the same codeword, thus increasing the space requirements.

We compared the performance of various techniques, including binary tree parsing, fixed step full lookups with different step sizes and our hybrid switch statement approach. The results are shown in Table 1. The experiments are performed on an Ultra 2 Sparc Workstation. The time shown is the time of parsing a file of 100,000 variable length codes taken from two different VLC tables in the MPEG-2 video international standard (ITU-T H.262) [11].

	VLC table for `macroblock_address_increment`	VLC table for `coded_block_pattern`
Codewords	34	64
File size (K bytes)	30	60
Binary tree	0.20 sec	0.36 sec
Fixed step 2 bits	0.15 sec	0.22 sec
Fixed step 3 bits	0.13 sec	0.17 sec
Fixed step 5 bits	0.12 sec	0.13 sec
Hybrid switch	0.06 sec	0.06 sec

Table 1: Performance Comparison of binary tree, fixed step sizes and our hybrid approach using switch statement.

In terms of time, the technique is faster than a hierarchical full-lookup approach with identical step sizes. The speed is 2 times faster as using the fixed step size 5 which is used in the MPEG-2 Software Simulation Group decoder. This is because switching consumes little time comparing with fixed-step's function lookups. Furthermore, it is optimized by ordering the `case` statements in terms of the length of their codeword. As shorter lengths correspond to higher probabilities, this minimizes the average number of comparisons per codeword. Further optimization is also possible in our implementation. In particular, in our current implementation there are some redundant (performance-wise) function calls that are resulting from ease of code generation and maintainability of our translator.

When processing a map, the translator first checks that it is uniquely decodable, i.e., that no codeword is the prefix of another. It then generates a class for it and constructs two methods for each map: `get()` and `put()`. The `get()` method is responsible for decoding a map entry and returning the decoded value, while the `put()` method is responsible for the output of the corresponding codeword. Note that these two methods are specifically generated and customized for each map, instead of using a generic (and thus less efficient) parsing algorithm.

5. CONCLUDING REMARKS

Flavor can provide significant benefits in the area of media representation and multimedia application development. With appropriate translation software, and a bitstream representation written in Flavor, obtaining access to such content is as simple as cutting and pasting the Flavor source code from the specification into an ASCII file, and running the translator. With Flavor’s translation tool, customized code for each specific VLC table is automatically generated that achieves a very good tradeoff between computational and memory requirements.

6. REFERENCES

[1] T. M. Cover and J. A. Thomas, Elements of Information Theory, Wiley, 1991.

[2] Graphics Interchange Format, CompuServe Inc., 1987, 1989.

[3] B. G. Haskell, A. Puri, and A. N. Netravali, Digital Video: An Introduction to MPEG-2, Chapman and Hall, 1997.

[4] ISO/IEC JTC1/SC29/WG11 N2201, Final Text of ISO/IEC FCD 14496-1 Systems, May 1998.

[5] Eleftheriadis, “A Syntactic Description Language for MPEG-4,” Contribution ISO/IEC JTC1/SC29/WG11 M546, Dallas, November 1995.

[6] Y. Fang and A. Eleftheriadis, “A Syntactic Framework for Bitstream-Level Representation of Audio-Visual Objects,” Proc., 3rd IEEE Int’l Conf. on Image Processing, Lausanne, September 1996, pp. II.429–II.432.

[7] O. Avaro, P. Chou, A. Eleftheriadis, C. Herpel, and C. Reader, “The MPEG-4 System and Description Languages: A Way Ahead in Audio Visual Information Representation,” *Signal Processing: Image Communication*, Special Issue on MPEG-4, Vol. 9, No. 4, May 1997, pp. 385–431.

[8] S. W. Smoliar and H. Zhang, “Content-Based Video Indexing and Retrieval,” *IEEE Multimedia Magazine*, Summer 1994.

[9] J. R. Smith and S.-F. Chang, "Visually Searching the Web for Content," IEEE Multimedia Magazine, Vol. 4 No. 3, Summer 1997, pp.12-20.

[10] A. Eleftheriadis, "Flavor: A Language for Media Representation", Proceedings, ACM Multimedia '97 Conference, Seattle, WA, November 1997.

[11] ISO/IEC Recommendation ITU-T H.262, MPEG-2 Video International Standard, 1995.