THE DALÍ MULTIMEDIA SOFTWARE LIBRARY[1]

Wei-Tsang Ooi, Brian Smith, Sugata Mukhopadhyay,
Haye Hsi Chan, Steve Weiss, Matthew Chiu, Jiesang Song
Department of Computer Science,
Cornell University,
Ithaca, NY 14850

Abstract - This paper presents a new approach for constructing libraries for building processing-intensive multimedia software. Such software is currently constructed either "from scratch" or by using high-level libraries. We have found that the second approach produces inefficient code, while the first approach is time-consuming. We therefore designed and implemented *Dalí,* a set of reusable, high-performance primitives and abstractions that are at an intermediate point in this design space. By decomposing common multimedia data types and operations into thin abstractions and primitives, programs written using Dalí are shorter and more reusable then hand-tuned C code, yet achieve competitive performance. This paper describes the design and implementation of Dalí.

INTRODUCTION

Recently, the multimedia research community has begun to examine the systems problems that arise in processing intensive multimedia applications. These applications include content-based retrieval and understanding [4,5], video production [6], and transcoding for heterogeneity and bandwidth adaptation [1,2].

The lack of a high-performance toolkit that researchers can use to build processing-intensive multimedia applications is hindering the progress of this research. Currently, researchers have several options when constructing processing intensive multimedia applications, none of which is very good. They can develop code from scratch, but the complex nature of common multimedia encoding schemes (e.g., MPEG) makes this approach impractical. A second option is to modify an existing code base (e.g. `mpeg_play`) to add the desired functionality. However, this approach requires understanding thousands of lines of code and usually results in a complex system that is difficult to debug, maintain, and reuse. A third option is to use standard libraries, such as ooMPEG or the IJG JPEG library. Such libraries provide a high-level API that hide many details, making optimization difficult or impossible. Furthermore, interoperability between the libraries is problematic.

[1] This research was supported by DARPA/ONR (contract N00014-95-1-0799), and grants from the National Science Foundation, Kodak, Intel, Xerox, and Microsoft.

These concerns led us to develop Dalí, a library for building processing-intensive multimedia software. Dalí consists of a set of simple, interoperable, high-performance primitives and abstractions that can be composed to create higher level operations and data types. Dalí lies between the high-level APIs provided by common libraries and low-level C code. It exposes some low level operations and data structures but provides a higher level of abstraction than C. We have found that this design makes it possible to write compact high-performance, processing-intensive multimedia software.

The challenge of Dalí was to design a library of functions that (1) allowed us to write code whose performance was competitive with hand-tuned C code, (2) allowed us to perform almost any optimization we could think of without breaking open the abstractions, and (3) could be composed in interesting, unforeseen ways.

`ByteImage`	2D array of values in the range 0..255. A `ByteImage` can represent a gray-scale image. Three equal-size `ByteImages` can represent a RGB image, one for each color channel.
`BitImage`	2D array of values in the range 0..255. A `ByteImage` can represent a gray-scale image. Three equal-size `ByteImages` can represent a RGB image, one for each color channel.
`DCTImage`	2D array of elements, each of which is a sequence of (*index*,*value*) pairs representing the run-length-encoded DCT blocks found in block-based compression schemes, such as MPEG and JPEG
`VectorImage`	2D array of vectors, each with a horizontal and vertical component, representing motion-vectors found in MPEG or H.261.
`AudioBuffer`	1D array of 8 or 16-bit values. 8 or 16-bit PCM audio, μ-law or A-law audio data can be represented using an `AudioBuffer`. The audio can be either stored as single channel or contain both left and right channels
`ByteLUT`	Look-up table for `ByteImages`, which can be applied to a `ByteImage` to produce another `ByteImage`. A GIF Image can be represented using three `ByteLUTs`, (one for each color plane in the color map), and one `ByteImage` (the color-mapped pixel data).
`AudioLUT`	Look-up table for `AudioBuffers`, which can be applied to an `AudioBuffer` to produce another `AudioBuffer`. For example, `AudioLUT` can be used to convert a 16-bit `AudioBuffer` into a 8-bit `AudioBuffer`
`Kernel`	2D array of integers, used for convolution
`Filter`	A scatter/gather list that can be used to select a subset of a BitStream. For example, Filter can be used to select an audio stream from an MPEG-1 system stream.
`BitStream/` `BitParser`	A `BitStream` is a buffer for encoded data. A `BitParser` provides a cursor into the `BitStream` and functions for reading/writing bits from/to the `BitStream`.

Table 1. Basic Abstractions in Dali

The rest of this paper is organized as follows. To show how Dalí is used, we describe it in section 2 through an example. Section 3 describes the design principles of Dalí. The implementation and its performance is briefly discussed next, and we conclude by presenting our plans for Dalí.

EXAMPLE OF DALÍ

In this section, we present an example of Dalí program that illustrate its use and power. To understand Dalí, it is helpful to understand the data types that Dalí provides. We list the basic abstractions in Table 1. Beside these basic abstractions, Dalí also has abstractions to store encoding-specific information, such as headers (from MPEG, JPEG, AVI, GIF, PNM), Huffman table (from JPEG), etc.

The example that we present in Figure 1 shows how to use Dalí to decode the I-Frames from an MPEG-1 video stream into RGB images. It illustrates that breaking down complex decoding operations into "thin" primitives makes Dalí code highly configurable. For example, by removing lines 36-38, we get a program that decodes I-Frame into gray scale images. By replacing line 35-38 with JPEG encoding primitives, we get an efficient I-Frame to JPEG transcoder.

Due to space limitations, we cannot describe other Dalí abstractions in greater detail. Interested readers should consult the Dalí web site[2] for more details and examples.

DESIGN PRINCIPLES OF DALÍ

One of the contributions of this research is a case study in the design of high-performance software libraries for processing multimedia data. Many of the design decision we made differ from other libraries because Dalí emphasizes performance over ease of use. Three themes emerge from these design decisions :

- *predictable performance*. It is important that programmers have a simple, well-defined cost model for the functions provided by a library. Predictable performance simplifies the analysis required when making design decisions between alternative implementations. It also ensure well-behaved program in real-time environments.
- *resource control*. We wanted to give programmers better control over the machine's resources (at the language level). Dalí provides several mechanisms for giving the programmer tight control over memory allocation and I/O execution, and for reducing or eliminating unnecessary memory allocation.
- *replacability* and *extensibility*. We wanted Dalí to be usable in many applications, not just the ones we envision. For example, Dalí would be useful in building a multimedia database. Since most DBMSs perform their own I/O, we separated the Dalí I/O functions from the computation functions.

The following sections describe how we solved this problem in greater detail

---
[2] `http://www.cs.cornell.edu/dali`

I/O Separation. Few Dalí primitives perform I/O. The only ones that do are special I/O primitives that load/store `BitStream` data. All other Dalí primitives use `BitStream` as their data source. This separation has three advantages. It makes the I/O method used transparent to Dalí primitives, gives programmer control of when I/O is performed and makes the performance of the remaining functions more predictable.

```
Bitsream bs = BitStreamNew(65536);        // Allocates the data structures
BitParser  *bp = BitParserNew();          // needed for decoding.
MpegSeqHdr *seqhdr = MpegSeqHdrNew();
MpegPicHdr *pichdr = MpegPicHdrNew();
int w, h, vbvsize, done=0;

BitParserAttach (bp, bs);                 // Point cursor of bp to start of
BitStreamFileRead (bs, file, 0);          // bs and read in data from file

MpegSeqHdrFind (bp);                      // Move cursor to start of seq hdr
MpegSeqHdrParse (bp, seqhdr);             // and parse the seq hdr

w = seqhdr->width;                        // Find out the width, height and
h = seqhdr->height;                       // minimum amount of data needed
vbvsize = seqhdr->vbv_buf_size;           // to decode the video sequence.

r = ByteNew(w, h);                        // Allocate ByteImages & DCTImages
g = ByteNew(w, h);                        // needed. r, g, b store decoded
b = ByteNew(w, h);                        // pic in RGB color space, and y,
y = ByteNew(w, h);                        // u, v store the decoded pic in
u = ByteNew(w/2, h/2);                    // YUV color space.  Variables
v = ByteNew(w/2, h/2);                    // dcty, dctu, dctv store semi-
dcty  = DctNew(w/16, h/16);               // compressed (DCT domain) picture
dctu  = DctNew(w/32, h/32);               // data
dctv  = DctNew(w/32, h/32);
while (!done) {
  int marker;
  MpegAnyMarkerFind (bp);                 // Advance cursor to next marker
  marker = MpegGetCurrMarker (bp);        // and retrieve the marker.
  switch (marker) {
    case PIC_HDR_MARKER :                 // If we are at the beginning of a
      MpegPicHdrParse (bp, pichdr);       // pic hdr, we parse it and check
      if (pichdr->type == I_FRAME) {      // its type.  If the pic is an
        MpegPicIParse(bp,dcty,dctu,dctv); // I-Frame, we parse it into three
        DctToByte (dcty, y);              // DCTImages, and perform IDCT to
        DctToByte (dctu, u);              // convert them into YUV images.
        DctToByte (dctv, v);              // Then we convert the YUV images
        YuvToRgb420 (y, u, v, r, g, b);   // into RGB images.
      }
      break;

    case GOP_HDR_MARKER :                 // If we encounter a GOP header,
      MpegGopHdrSkip (bp);                // skip it since we don't care
      break;                              // about the information from the
                                          // GOP header.
    case SEQ_END_MARKER :                 // If we encounter a seuqence end
      done = 1;                           // marker, we set done to 1 to
  }                                       // terminate the while loop.
  if (!done) {
    UpdateBitstream(bp,bs,file,vbvsize);  // We top up bs with data from
  }                                       // file to make sure there are
}                                         // at least vbvsize bytes in bs
```

Figure 1. Dalí code to decode the I-frames of an MPEG video to RGB format

Explicit Memory Allocation. In Dalí, the programmer allocates and frees all non-trivial memory resources using new and free primitives (e.g., `BitStreamNew` and `BitStreamFree`). Functions never allocate temporary memory – if such memory is required to complete an operation (scratch space, for example), the programmer must allocate it and pass it to the routine as a parameter. Explicit memory allocation allows the programmer to reduce or eliminate paging, and make the performance of the application more predictable.

Sharing of Memory. Dalí provide two mechanisms for sharing memory between abstractions: *clipping* and *casting*. In clipping, one abstraction "borrows" memory from another of the same type. For example, to decode part of the gray-scale image in an MPEG I-frame, we create a clipped `DCTImage` that contains the required subset of DCT blocks from the decoded I-frame and perform IDCT on that clipped image. Casting refers to sharing of memory between objects of different types. For instance, we can read a gray image file into `BitStream`, parse the headers, and cast the remaining data into a `ByteImage`. Sharing of memory eliminates unnecessary copying and processing.

Specialization. Many Dalí primitives implement special cases of a more general operation. The special cases can be combined to achieve the same functionality of the general operation, and have a simple, fast implementation whose performance is predictable. An example is image scaling. Instead of providing one primitive that scales an image by an arbitrary factor, Dalí provides five primitives to shrink an image (`Shrink4x4`, `Shrink2x2`, `Shrink2x1`, `Shrink1x2`, and `ShrinkBilinear`) and five others to expand and image.

Exposing Structures. Dalí exposes the structure of compressed data in two ways. First, Dalí exposes intermediate structures in the decoding process. We have seen an example of this in the MPEG decoding program presented in Figure 1. Dalí also exposes the structure of the underlying bit stream, which can be exploit by programmers for better performance. For example, a program that searches for an event in a MPEG video stream might initially cull the data set by examining only the I-frames. Such optimizations are impossible using libraries that hide the encoding structure from programmers.

IMPLEMENTATION AND PERFORMANCE

Dalí is currently implemented as a C run time library with approximately 50K lines of code. A Tcl and Java binding is also available. It has been ported to Win95/NT, SunOS 4, Solaris, and Linux. The Dalí library is divided into several packages according to their functionality and data type support. Supported data type includes PNM, GIF, JPEG, WAV, MPEG and AVI. The code can be downloaded from `http://www.cs.cornell.edu/dali/`.

To evaluate the performance of Dalí, we are writing simple utilities and comparing them to similar utilities widely used in the research community. So far, our results are very encouraging. For example, our Dalí MPEG to PPM decoding program is about 150 lines long but is 10% faster than the Berkeley MPEG player on a Sparc 20[3]. We also compare the performance of a JPEG decoder, JPEG encoder and MPEG encoder written in Dalí with `djpeg`, `cjpeg` and Berkeley MPEG encoder and found that our Dalí programs run at least as fast as these highly optimized C code

CONCLUSION AND FUTURE WORK

We think that Dalí will be a useful tool to the community because it allows efficient, reusable, processing-intensive multimedia software to be built with relatively small effort, and provides a vehicle through which research groups that uses this software can exchange their results. We are currently enhancing Dalí with support for MPEG-2, MPEG-4 and H.263. We are also building a multithreaded implementation of Dalí and integrating Dalí into the Mash [3]. Mash is a toolkit for constructing multimedia applications such as vic and vat. Integrating Dalí into Mash will allow us to build many interesting applications, such as a programmable media gateway where users can upload Dalí programs that process multimedia data as it flows through the network.

REFERENCE

[1] S. Acharya and B. Smith, "Compressed Domain Transcoding of MPEG," In *Proc. of Intl. Conf. on Multimedia Comp. and Sys.*, Austin, Texas. June 98.

[2] E. Amir, S. McCanne and H. Zhang, "An Application Level Video Gateway," In *Proc. of ACM Multimedia '95*, San Francisco, California, Nov 95.

[3] S. McCanne. et. al., "Toward a Common Infrastructure for Multimedia-Networking Middleware," In *Proc. of 7th NOSSDAV*, St. Louis, Missouri, May 97.

[4] N. V. Patel and I. K. Sethi, "Compressed Video Processing for Cut Detection," *IEEE Proc: Vision, Image and Signal Processing*, Vol. 143, Oct 96.

[5] B. Shen and I. K. Sethi, "Convolution-Based Edge Detection for Image/Video in Block DCT Domain," *J. of Visual Comm. and Image Representation (In Press).*

[6] T. Wong et. al. "Software-only video production switcher for the Internet MBone," *Proc. of Multimedia Computing and Networking*. San Jose, California, 1998.

[3] We used `mpeg_play -dither color -no_display`