The Dalí Multimedia Software Library

Wei-Tsang Ooi, Brian Smith, Sugata Mukhopadhyay, Haye Hsi Chan, Steve Weiss, Matthew Chiu[a]
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 by using high-level libraries or by writing it "from scratch" using C. We have found that the first approach produces inefficient code, while the second approach is time-consuming and produces complex code that is difficult to maintain or reuse. We therefore designed and implemented *Dalí*, a set of reusable, high-performance primitives and abstractions that are at an intermediate level of abstraction between C and conventional libraries. By decomposing common multimedia data types and operations into thin abstractions and primitives, programs written using Dalí achieve performance competitive with hand-tuned C code, but are shorter and more reusable. Furthermore, Dalí programs can employ optimizations that are difficult to exploit in C (because the code is so verbose) and impossible using conventional libraries (because the abstractions are too thick). We discuss the design of Dalí, show several example programs written using Dalí, and show that programs written in Dalí achieve performance competitive to hand-tuned C programs.

Keywords: Multimedia toolkits, image processing, video processing, MPEG, JPEG

1. INTRODUCTION

The multimedia research community has traditionally focused much of its efforts on the compression, transport, storage, and display of multimedia data. These technologies are fundamentally important for applications such as video conferencing and video-on-demand, and results from the research community have made their way into many commercial products. For example, JPEG[15] and MPEG[8] are ubiquitous standards for image and audio/video compression, and video conferencing tools such as Microsoft NetMeeting[11] and CU-SeeMe[4] are gradually becoming part of a desktop PC.

Although many research problems remain in these areas, the research community has begun to examine the systems problems that arise in multimedia data processing, such as content-based retrieval and understanding[17,19], video production[22], 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 this research. Currently, researchers have several options (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. For example, several man-years of work went into writing the Berkeley MPEG player (`mpeg_play`)[16], and we believe a similar amount of effort would be required to reproduce this software.

A more commonly used option is to modify an existing code base to add the desired functionality. For example, many researchers have "hacked up" `mpeg_play` to test their ideas. However, this approach requires understanding thousands of lines of code and usually results in complex, unmanageable systems that are difficult to debug, maintain, and reuse.

A third option is to use standard libraries, such as ooMPEG[13] or the Independent JPEG Group (IJG) software[6]. Such libraries provide a high-level API that hides many of the details of the underlying compression scheme. However, since programmers can not penetrate the "black-box" of the API, they can only exploit limited optimizations. For example, to extract a gray scale image from an MPEG frame, the programmer must convert the RGB image returned by the ooMPEG library into a gray scale image. A much more efficient strategy is to extract the gray scale image directly from the MPEG frame data, since it is stored in a YUV color space. The ooMPEG abstractions do not support this optimization. Another problem is that these libraries usually provide functions for specific multimedia format, making interoperability between the libraries difficult. For example, it is difficult to transcode an MPEG I frame into a JPEG image, although both of them are DCT coded.

These concerns have lead us to develop Dalí, a library for constructing 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

[a] Correspondence: Email : `dali@cs.cornell.edu`; WWW : `http://www.cs.cornell.edu/dali`

exposes some low level operations and data structures but provides a higher level of abstraction than C, making it possible to compactly write high-performance, processing-intensive multimedia software. Dalí's mechanisms include:

Resource control. Programmers have full control over memory utilization and I/O. With few exceptions, Dalí routines do not *implicitly* allocate memory or perform I/O – such functions are always explicitly requested by the programmer. This feature gives programmers tight control over performance-critical resources, an essential feature for writing applications with predictable performance. Dalí also gives programmers mechanisms to optimize their programs using techniques such as data copy avoidance and structuring their programs for good cache behavior.

"Thin" primitives. Dalí breaks complex functions into simple functions that can be layered. This feature promotes code reuse and allows optimizations that would otherwise be difficult to exploit. For example, to decode a JPEG image, Dalí provides three primitives: (1) a function to decode the bit stream into three `SCImages`, one for each color component (a `SCImage` is an image where every "pixel" is a structure containing DCT coefficients), (2) a function to convert each `SCImage` into a `ByteImage` (an uncompressed image whose pixels are integers in the range 0..255), and (3) a function to convert from YUV color space to RGB color space. Exposing this structure has several advantages. First, it promotes code reuse. For instance, the inverse DCT and color-space conversion functions are shared by the JPEG and MPEG routines. Second, it allows optimizations that would be difficult to exploit otherwise. For example, compressed domain processing techniques [1,17,19,20] can be implemented on `SCImages`.

Exposing Structure: Dalí provides functions to parse compressed bit streams, such as MPEG, JPEG, and GIF. These bit streams consist of a sequence of *structural elements*. For example, an MPEG-1 video bit stream consists of a *sequence header* followed by one or more *group-of-pictures* (GOPs -- Figure 6). Each GOP is a *GOP header* followed by one or more pictures. Each picture is a *picture header* followed by encoded picture data. While other libraries hide these structures from programmers, Dalí exposes them. Dalí provides five functions for each structural element: *find*, *parse*, encode, skip, and *dump*. The functions operate on data in a memory buffer (call a `BitStream`). *Find* locates that element in the `BitStream`, *parse* reads the element into an associated data structure, *encode* writes a data structure into the `BitStream`, *skip* moves the `BitStream` cursor past that structural element, and *dump* copies the element from one `BitStream to` another. These routines allow a programmer to operate on a bit stream at a high level, but to perform operations that are impossible with conventional libraries. For example, writing a routine that counts the number of I frames in an MPEG sequence is trivial: one simply finds each picture header, parses it, and increments a counter if it is an I frame (indicated by the type field in the picture header structure). Similarly, writing a program to demultiplex MPEG system streams or analyze the structure of MPEG sequence is very easy. Similar considerations hold for other formats, such as GIF and JPEG.

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. We believe that we have achieved these goals. For example, a Dalí program that decodes an MPEG-1 video into a series of RGB images is about 150 lines long, runs about 10% faster than `mpeg_play`, and can be easily modified for new environments.

The contributions of this research are two-fold. First, we believe that Dalí provides a fairly complete set of operations that will be useful to the research community for building processing intensive multimedia application. Dalí is freely available for research use from `http://www.cs.cornell.edu/dali/`. Second, this research is a case study in designing high-performance multimedia APIs. It provides a model for what APIs operating systems should provide to programmers.

The rest of this paper is organized around these two contributions. To show how Dalí is used, we describe it in Section 2 through three illustrative examples. Section 3 describes the design principles of Dalí. The implementation and its performance are briefly discussed next, and we conclude by presenting our plans for Dalí and outline related work.

2. DALÍ BY EXAMPLE

This section is intended to give the reader a feel for programs written using Dalí. We first outline the major abstractions defined by Dalí and then present three examples of programs written with Dalí that illustrate its use and power.

2.1. Abstractions

To understand Dalí, it is helpful to understand the data types Dalí provides. The basic abstractions in Dalí are:

- `ByteImage` – a 2D array of values in the range 0..255.
- `BitImage` – a 2D array of 0/1 values.

- `SCImage` – an image where each "pixel" is a structure that represents the run-length-encoded DCT blocks found in many block-based compression schemes, such as MPEG and JPEG.
- `VectorImage` – an image where each "pixel" is a structure that represents motion-vector found in MPEG or H.261.
- `AudioBuffer` – an abstraction to represent audio data (mono or stereo, 8-bit or 16-bit).
- `ImageMap` –represents a look-up table that can be applied to one `ByteImage` to produce another `ByteImage`.
- `AudioMap` – a look-up table for `AudioBuffer`.
- `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`.
- `Kernel` – 2D array of integers, used for convolution.
- `BistreamFilter` – a scatter/gather list that can be used to select a subset of a `BitStream`.

These abstractions can be used to represent common multimedia data objects. For example,

- A gray-scale image can be represented using a `ByteImage`.
- A monochrome image can be represented using a `BitImage`.
- An irregularly shaped region can be represented using a `BitImage`.
- An RGB image can be represented using three `ByteImages`, all of the same size.
- A YUV image in 4:2:0 format can be represented using three `ByteImages`. The `ByteImage` that represents the Y plane is twice the width and height of the `ByteImages` that represent the U and V planes.
- The DCT blocks in a JPEG image, an MPEG I-frame, or the error terms in an MPEG P- and B-frame can be represented using three `SCImages`, one for each of the Y, U and V planes of the image in the DCT domain.
- The motion vectors in MPEG P- and B-frame can be represented with a `VectorImage`.
- A GIF Image can be represented using three `ImageMaps`, one for each color map, and one `ByteImage` for the color-mapped pixel data.
- 8 or 16-bit PCM, μ-law or A-law audio data (mono or stereo) can be represented using an `AudioBuffer`.

Header	File Format
`PnmHdr`	NETPBM image header
`WavHdr`	WAVE audio header
`GifSeqHdr`	GIF file sequence header
`GifImgHdr`	GIF file image header
`JpegHdr`	JPEG image header
`JpegScanHdr`	JPEG scan header
`MpegAudioHdr`	MPEG-1 audio (layer 1, 2, 3) header
`MpegSeqHdr`	MPEG-1 video sequence header
`MpegGopHdr`	MPEG-1 video group-of-picture header
`MpegPicHdr`	MPEG-1 video picture header
`MpegSysHdr`	MPEG-1 system stream system header
`MpegPckHdr`	MPEG-1 system stream pack header
`MpegPktHdr`	MPEG-1 system stream packet header

Table 1. Header abstractions in Dalí

Dalí also has abstractions to store encoding-specific structures. For example, an `MpegPicHdr` stores the information parsed from a picture header in an MPEG-1 video bit stream. The header abstractions in the current implementation are listed in Table 1.

2.2. Examples

Although the set of abstractions defined in Dalí is fairly small (9 general purpose and 13 header abstractions), the set of operators that manipulate these abstractions is not. Dalí currently contains about 500 operators divided into 12 packages. The rationale for defining so many operators is discussed in section 3. For now, we simply note that it is neither practical nor productive to describe all the operators.

Instead, we present three examples that illustrate the use of the Dalí abstractions and give you a feel for programs written using Dalí. The first example shows how to use Dalí to manipulate images, the second shows how to use Dalí for MPEG decoding, and the last shows how to use a Dalí `BistreamFilter` to demultiplex an MPEG systems stream.

2.2.1. Image Primitives

The first example uses Dalí to perform a picture-in-picture operation (figure 3). Before explaining this example, we must describe the `ByteImage` abstraction in detail. A `ByteImage` consists of a *header* and a *body*. The header stores information such as width and height of the `ByteImage` and a pointer to the body. The body is a block of memory that contains the image data. A `ByteImage` can be either *physical* or *virtual*. The body of a physical `ByteImage` is contiguous in memory, whereas a virtual `ByteImage` borrows its body from part of another `ByteImage` (called its *parent*). In other words, a virtual

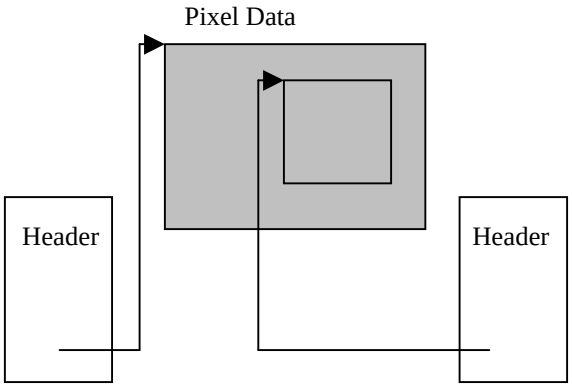


Figure 1. The `ByteImage` whose header is on the left is a physical `ByteImage`, whereas the `ByteImage` whose header is on the right is virtual.

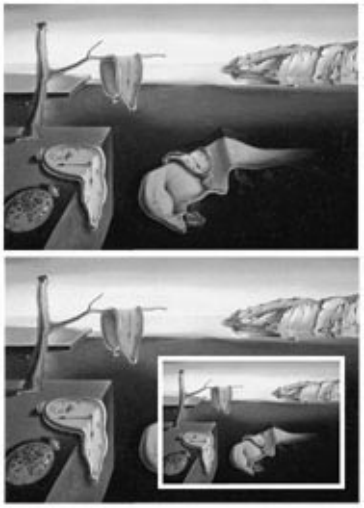


Figure 3. The input (top) and output (bottom) of the PIP operations

```
1  void PIP(image, borderWidth, margin)
2      ByteImage *image;
3      int borderWidth, margin;
4  {
5      int w  = ByteGetWidth(image);
6      int h  = ByteGetHeight(image);
7      int destW = w/2;
8      int destH = h/2;
9      int destX = w - destW - margin;
10     int destY = h - destH - margin;
11     ByteImage *dest;
12     ByteImage *temp;

13     temp = ByteNew(destW,destH);
14     ByteShrink2x2(image, temp);

15     dest = ByteClip(image,
16       destX-borderWidth, destY-borderWidth,
17       destW+2*borderWidth,destH+2*borderWidth);
18     ByteSet(dest, 255);
19     ByteFree(dest);

20     dest = ByteClip(image,
21       destX, destY, destW, destH);
22     ByteCopy(temp, dest);
23     ByteFree(dest);
24     ByteFree(temp);
25 }
```

Figure 2. PIP function written in Dalí

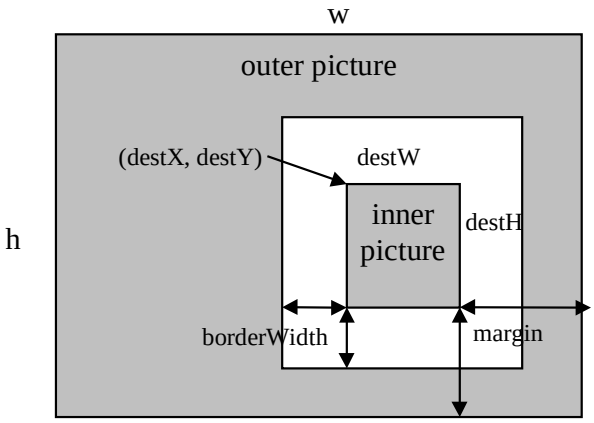


Figure 4. Variables used in PIP functions.

`ByteImage` provides a form of shared memory – changing the body of a virtual `ByteImage` implicitly changes the body of its parent (see Figure 1).

A new physical `ByteImage` is allocated using `ByteNew(w,h)`. A virtual `ByteImage` is created using `ByteClip(b,x,y,w,h)`. The rectangular area whose size is `w x h` and has its top left corner at `(x,y)` is shared between the virtual `ByteImage` and the physical `ByteImage`. The virtual/physical distinction applies to all image types in Dalí. For example, a virtual `SCImage` can be created to decode a subset of a JPEG image.

We now show how to use Dalí to create a "picture in picture" (PIP) effect on an image (figure 3). We choose this example as an example because it is simple, yet involves basic operators that illustrate the principles of Dalí.

The steps to create the PIP effect can be briefly stated as follows: given an input image, (1) shrink the image by half, (2) draw a white box slightly larger than the scaled image on the original image, and (3) paste the shrinked image into the white box..

Figure 2 shows a Dalí function that performs the PIP operation. The function takes in three arguments: `image`, the input image; `borderWidth`, the width of the border around the inner image in the output, and `margin`, the offset of the inner image from the right and bottom edge of the outer image. (See Figure 4).

Line 5 to line 6 of the function query the width and height of the input image. Line 7 to line 10 calculate the position and dimension of the inner picture. Line 13 creates a new physical `ByteImage`, `temp`, which is half the size of the original image. Line 14 shrinks the input image into `temp`. Line 15 creates a virtual `ByteImage` slightly larger than the inner picture, and line 18 sets the value of the virtual `ByteImage` to 255, achieving the effect of drawing a white box. Line 19 de-allocates this virtual image. Line 20 creates another virtual `ByteImage`, corresponding to the inner picture. Line 21 copies the scaled image into the inner picture using `ByteCopy`. Finally, line 22 and 23 free the memory allocated for the `ByteImages`.

This example shows how images are manipulated in Dalí through a series of simple, thin operations. It also illustrates several design principles of Dalí, namely (1) sharing of memory (through virtual images), (2) explicit memory control (through `ByteClip`, `ByteNew` and `ByteFree`), and (3) specialized operators (`ByteShrink2x2`). These design principles will be discussed in greater details in Section 3.

2.2.2. MPEG and BitStreams Primitives

Our next example illustrates how to process MPEG video streams using Dalí. Our example program decodes the I-frames in an MPEG video stream into a series of RGB images. Before discussing the example, we briefly review the format of MPEG video streams and the relevant Dalí abstractions and functions.

To parse an MPEG video stream, the encoded video data is first read into a `BitStream`. A `BitStream` is an abstraction for input/output operations – that is, it is a buffer. To read and write from the `BitStream`, we use a `BitParser`. A `BitParser` provides functions to read and write data to and from the `BitStream`, plus a cursor into the `BitStream`.

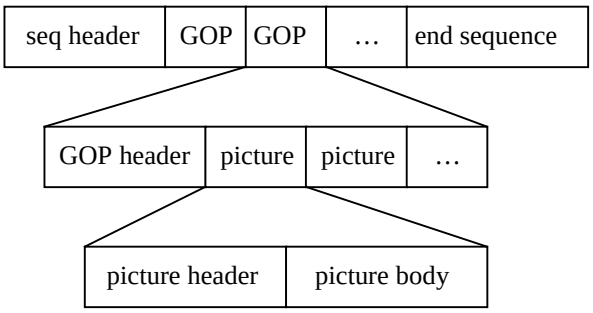


Figure 5. Format of an MPEG-1 Video Stream.

An MPEG video stream consists of a *sequence header*, followed by a sequence of *GOPs* (*group-of-pictures*), followed by an *end of sequence* marker (Figure 5). Each GOP consists of a GOP header followed by a sequence of *pictures*. Each picture consists of a *picture header*, followed by the compressed data required to reconstruct the picture. Sequence headers contain information such as the width and height of the video, the frame rate, the aspect ratio, and so on. The GOP header contains the timecode for the GOP. The picture header contains information necessary for decoding the picture, most notably the type of picture (I, P, B). Dalí provides an abstraction for each of these structural elements (see Table 1).

Dalí provides five primitives for each structural element: `find`, `skip`, `dump`, `parse`, and `encode`. `Find` positions the cursor in the `BitStream` just before the element. `Skip` advances the cursor to the end of the element. `Dump` moves the bytes corresponding to the element from input `BitStream` to the output `BitStream`, until the cursor is at the end of the header. `Parse` decodes the `BitStream` and stores the information into a header abstraction, and `encode` encodes the information from a header abstraction into a `BitStream`. Thus, the `MpegPicHdrFind` function advances the cursor to the next picture header, and `MpegSeqHdrParse` decodes the sequence header into a structure.

Given this background, we can describe the Dalí program shown in Figure 6, which decodes the I-frames in an MPEG video into RGB images. Lines 1 through 7 allocate the data structures needed for decoding. Line 8 associates `inbp` to `inbs`. The cursor of `inbp` will be pointing to the first byte of buffer in `inbs`, which is a memory-mapped version of the file. Lines 9-10 move `inbp` to the beginning of a sequence header and parse the sequence header into `seqhdr`.

We extract vital information such as width, height and the minimum data that must be present to decode a picture (`vbvsize`) from the sequence header in lines 11-13. Lines 14 through 22 allocate the `ByteImages` and `SCImages` we need for decoding the I-frames. The variables `scy`, `scu`, and `scv` store compressed (DCT domain) picture data, `y`, `u`, and `v` store the decoded picture in YUV color space, and `r`, `g`, and `b` store the decoded picture in RGB color space.

```
// filename is the name of the MPEG file to parse
1   BitStream *inbs = BitStreamMmapReadNew (filename);
2   BitParser  *inbp = BitParserNew ();
3   MpegSeqHdr *seqhdr = MpegSeqHdrNew ();
4   MpegPicHdr *pichdr = MpegPicHdrNew ();
5   int w, h, vbvsize, status;
6   ScImage *scy, *scu, *scv;
7   ByteImage *y, *u, *v, *r, *g, *b;

8   BitParserWrap(bp, bs);

9   MpegSeqHdrFind(bp);
10  MpegSeqHdrParse(bp, seqhdr);

11  w = MpegSeqHdrGetWidth(seqhdr);
12  h = MpegSeqHdrGetHeight(seqhdr);
13  vbvsize = MpegSeqHdrGetVbvSize(seqhdr);

14  r = ByteNew(w, h);
15  g = ByteNew(w, h);
16  b = ByteNew(w, h);
17  y = ByteNew(w, h);
18  u = ByteNew((w+1)/2, (h+1)/2);
19  v = ByteNew((w+1)/2, (h+1)/2);
20  scy = ScNew((w+15)/16, (h+15)/16);
21  scu = ScNew((w+31)/32, (h+31)/32);
22  scv = ScNew((w+31)/32, (h+31)/32);

23  while (1) {
24    status = MpegPicHdrFind (inbp);
25    if (status == DVM_MPEG_NOT_FOUND) break;
26    MpegPicHdrParse (inbp, pichdr);
27    if (pichdr->type == I_FRAME) {
28      MpegPicIParse (inbp,scy,scu,scv);
29      ScToByte (scy, y);
30      ScToByte (scu, u);
31      ScToByte (scv, v);
32      YuvToRgb420 (y, u, v, r, g, b);
33    }
34  }
```

Figure 6. Dalí code to decode the I-frames of an MPEG video to RGB format.

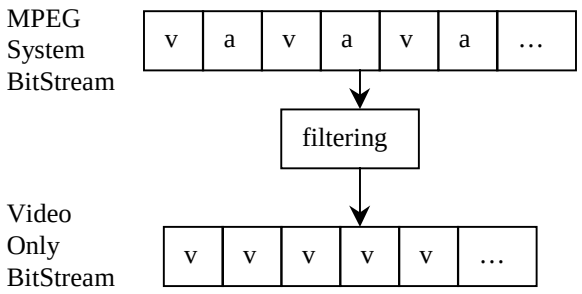


Figure 7. An MPEG system stream consists of interleaving video (v) and audio (a) packets. Filter efficiently extracts a portion of one input stream. For example, it extracts the video portion to create a video only stream.

The main loop in the decoding program (lines 23-34) starts by advancing the `BitParser` cursor to the next MPEG picture header (line 24). If the picture header is not found, we exit the loop (line 25). Otherwise, we parse the picture header (line 26) and check its type (line 27). If it is an I-frame, we parse it into three `SCImages`, (line 28), convert the `SCImages` to `ByteImages` (lines 29-31), and convert the `ByteImages` into RGB color space (line 32).

Breaking down complex decoding operations like MPEG decoding into "thin" primitives makes Dalí code highly configurable. For example, by removing lines 32 to 34, we get a program that decodes MPEG I-frame into gray scale images. By replacing line 31 to 34 with JPEG encoding primitives, we get an efficient MPEG I-frame to JPEG transcoder. Similarly, we can just as easily write a Motion-JPEG to MPEG I-frame transcoder.

2.2.3. BistreamFilters

Our final example illustrates how we can filter out a subset of a `BitStream` for processing. `BistreamFilters` were designed to simplify the processing of bit streams with interleaved data (e.g., AVI, QuickTime, or MPEG systems streams). `BistreamFilters` are similar to scatter/gather vectors – they specify an ordered subset of a larger set of data.

A common use of filtering is processing MPEG system streams, which consists of interleaved audio or video (*A/V*) streams (Figure 7). In MPEG, each A/V stream is assigned an unique id. Audio streams have ids in the range 0..31; video streams ids are in the range 32..47. The A/V streams are divided up into small (approx. 2 Kbytes) chunks, called *packets*. Each packet has a header that contains the id of the stream, the length of the packet, and other information (e.g., a timecode).

In this example, we build a `BistreamFilter` that can be used to copy the packets of the first video stream (id = 32) from a system stream stored in one `BitStream` to another. Once copied, we can use the Dalí MPEG video processing primitives on the video-only `BitStream`. The Dalí code for building this filter is shown in Figure 8.

Lines 2 through 8 allocate and initialize various structures needed by this program. The variable `offset` stores the byte offset of a packet in the bit stream, relative to the start of the stream. Line 9 advances the cursor to the beginning of the first packet header and updates `offset`. The main loop (lines 10-18) parses the packet header (line 11) and, if the packet belongs to the first video stream, its offset and length are added to `filter` (line 14). `EndOfBitstream` is a macro that checks the position of the bit stream cursor against the length of the data buffer.

Once the `BistreamFilter` is constructed, it can be saved to disk, or used as a parameter to functions such as `BitStreamFileFilter`, which reads the subset of a file specified by the filter, or `BitstreamDumpUsingFilter`., which copies the data subset specified by a filter from one `BitStream` to another.

This example illustrates how Dalí can be used to demultiplex interleaved data. The technique is easily extended to other formats, such as QuickTime, AVI, MPEG-2 and MPEG-4. Although this mechanism uses data copies, the cost of copying is offset by the performance gain when processing the filtered data. Another option (one we initially tried) is to integrate the filter mechanism directly into the bit-at-a-time parsing functions provided by the `BitParser`. Although this design avoids unnecessary data copies, we found the overhead of checking if the cursor was at a filter segment boundary on each function call too high to make this design practical. A better option would be to provide hardware support for scatter/gather vectors[3].

```
1   #define SIZE (128*1024)
2   int len, offset, start = 0;
3   MpegPktHdr *hdr = MpegPktHdrNew();
4   BitStream *bs = BitStreamNew (SIZE);
5   BitParser *bp = BitParserNew ();
6   BitStreamFilter * filter = BitStreamFilterNew();

7   BitParserAttach (bp, bs);
8   BitStreamFileRead (bs, file);

9   offset = MpegPktHdrFind (bp);
10  while (!eof(file) && !EndOfBitstream(bp)) {
11    MpegPktHdrParse (bp, hdr);
12    if (hdr->id == 32) {
13      len = hdr->len;
14      BitStreamFilterAdd(filter, offset, len);
15      start += UpdateIfUnderflow (bp,bs,file,SIZE/2);
16      offset = start + MpegPktHdrFind(bp);
17    }
18  }
```

Figure 8. Filtering an MPEG system stream by copying the first video stream to another `BitStream` for processing.

These three examples should give you a feel for Dalí programs. Interested readers should consult the Dalí web site at `http://www.cs.cornell.edu/dali` for more details and examples.

3. DESIGN PRINCIPLES OF DALÍ

One of the contributions of this research is that it provides 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. This goal put us at an unusual point in the design space. In this section, we highlight the principles that emerged during the design of Dalí.

Three themes emerge from these principles. The first theme is *predictable performance*. We designed Dalí to allow programmers to easily predict the performance of their code. We believe it is important that programmers have a simple, well-defined cost model for the functions provided by a library. Predictable performance is important for writing high-performance code because it simplifies the analysis required when making design decisions between alternative implementations of a program. It is also important for writing programs that are well-behaved in real-time environments.

The cost of functions in existing libraries can be difficult to predict. This unpredictability has several sources. Often, functions will perform hidden, expensive operations, such as I/O or memory allocation. How much these operations cost, and when they occur, is hidden behind the abstractions provided by the API. A second source of unpredictability is the APIs, which often provide abstractions that are too coarse. For instance, many video-decoding libraries provide a function to "get the next frame." But the execution time for this function can be vastly different when decoding MPEG video, depending on the frame type (I, P, or B). A third source of unpredictability is that the execution time of a function can be very non-linear, depending on the value of the parameters. For example, scaling an image down by a factor of 2 can be significantly faster than scaling an image down by a factor of 1.9 if interpolation is used.

The second, closely related theme in the design of Dalí is *resource control*. We wanted to give programmers better control over the machine's resources (at the language level, not the OS level). Predictable performance gives programmers control over their use of the CPU, but memory and I/O are very important resources in multimedia applications. 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.

The final theme is *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 database management systems perform their own I/O, we separated the Dalí I/O functions from the computation functions. Throughout the design, we tried to make Dalí extensible and pieces of it replaceable.

In summary, the design problem we faced was providing an API that was coarse enough to provide a useful level of abstraction to the programmer, yet fine enough to give the programmer tight control over their code. The following sections describe the mechanisms we use to solve this problem in detail.

3.1. 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. First, it makes the I/O method used transparent to Dalí primitives. Other libraries use integrated processing and I/O. A library that integrates file I/O with its processing is difficult to use in a network environment, since the I/O behavior of networks is different from that of files. Second, the separation of I/O also allows control of when I/O is performed. For example, we are building a multithreaded implementation of Dalí that will allow us to use a double buffering scheme to read and process data concurrently. Third, by isolating the I/O calls, the performance of the remaining functions becomes more predictable.

3.2. Sharing of Memory

Dalí provides two mechanisms for sharing memory between abstractions. These mechanisms are called *clipping* and *casting*. In clipping, one object "borrows" memory from another object of the same type. An example usage of clipping can be seen in Figure 1. Clipping functions are extremely cheap (they only allocate an image header structure), and are provided for all Dalí image and audio data types. Clipping is useful for avoiding unnecessary copying or processing of data. For example, if we only want to decode part of the gray-scale image in an MPEG I-frame, we could create a clipped `SCImage` that contains a subset of DCT blocks from the decoded I-frame and then perform the IDCT on that clipped image. The advantage of this strategy is that it avoids performing the IDCT on encoded data that we will not use.

While clipping is the sharing of memory between objects of the same type, *casting* refers to the sharing of memory between objects of different types. Casting avoids unnecessary copying of data. Casting is often used in I/O, since all I/O must be done through a `BitStream`. To avoid copying data, a section of the BitStream buffer can be shared with another object. For instance, we can read a PGM[b] image file into `BitStream`, parse the headers, and cast the remaining data into a `ByteImage`.

3.3. Explicit Memory Allocation

In Dalí, the programmer allocates and frees all non-trivial memory resources using *new* and *free* primitives (e.g., `ByteImageNew` and `ByteImageFree`). 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.

To illustrate these points, consider the `ByteCopy` function, which copies from one `ByteImage` to another. One potential problem is that the two `ByteImages` might overlap (e.g., if they share memory, via clipping). One way to implement `ByteCopy` is shown on the left side of Figure 9. This implementation allocates a temporary buffer, copies the source into the temporary buffer, copies the temporary buffers into the destination, and frees the temporary buffer. In contrast, the Dalí `ByteCopy` operation assumes that the source and destination do not overlap, so it simply copies the source into the destination. The programmer must determine if the source and destination overlap, and if so allocate a temporary `ByteImage` and two `ByteCopy` calls (Figure 9, right).

```
ByteCopy(src, dest) {
  temp = malloc ();
  memcpy src to temp;
  memcpy temp to dest;
  free (temp);
}

ByteCopy(src, dest);
```

```
ByteCopy(src, dest) {
  memcpy src to dest;
}

temp = ByteNew ();
ByteCopy(src, temp);
ByteCopy(temp, dest);
ByteFree (temp);
```

Figure 9. Left: an implementation of `ByteCopy` where memory is implicitly allocated. Right: the Dalí implementation of `ByteCopy`.

A third possible implementation is to only allocate a temporary buffer if the source and destination overlap. This implementation has the drawback that its performance would be difficult to predict. If the source and destination overlap, the function could take 2-3 times longer to complete than if they do not.

[b] A PGM file contains a header followed by raw, gray-scale image data in row-major order – the same order as Dali `ByteImages`.

3.4. 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. `ByteCopy` is one such primitive – only the special case of non-overlapping images is implemented.

Another example is image scaling (shrinking or expanding the image). 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 an image. Each primitive is highly optimized and performs a specific task. For example, `Shrink2x2` is a specialized function that shrinks the image by a factor of 2 in each dimension. It is implemented by repeatedly adding 4 pixel values together and shifting the result, an extremely fast operation. Similar implementations are provided for `Shrink4x4, Shrink2x1,` and `Shrink1x2.` In contrast, the function `ShrinkBilinear` shrinks an image by a factor between 0.5 and 2 using bilinear interpolation. Although arbitrary scaling can be achieved by composing these primitives, splitting them into specialized operations makes the performance predictable, exposes the cost more clearly to the programmer, and allows us to produce very fast implementations.

3.5. Generalization

The drawback to specialization is that it can lead to an explosion in the number of functions in the API. Sometimes, however, we can combine several primitives without sacrificing performance, which significantly reduces the number of primitives in the API. We call this principle *generalization.*

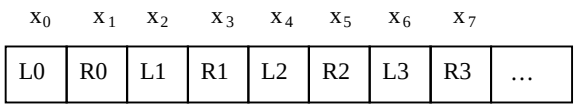
A good example of generalization is found in the primitives that process `AudioBuffers`. `AudioBuffers` store mono or stereo audio data. Stereo samples from the left and right channels are interleaved in memory (Figure 10, top).

Suppose you were implementing an operation that raises the volume on one channel (i.e., a balance control). One possible design is to provide one primitive that processes the left channel and another that processes the right channel (Figure 10a). However, we can combine the two without sacrificing performance by modifying the initialization of the looping variable (1 for right, 0 for left). This implementation is shown in Figure 10b.

In general, if specialization gives better performance, it should be used. Otherwise, generalization should be used to reduce the number of functions in the API.

3.6. Exposing Structures

Most libraries try to hide details of encoding algorithms from the programmer, providing a simple, high-level API. In contrast, Dalí exposes the structure of compressed data in two ways.




```
(a)
process_left(x)
for (i = 0; i < n; i+= 2) {
    process x[i]
}
process_right(x)
for (i = 0; i < n; i+= 2) {
    process x[i]
}

(b)
process(x, offset)
for (i = offset; i < n; i+= 2) {
    process x[i]
}
```

Figure 10. Generalization of an `AudioBuffer` primitives.

First Dalí exposes intermediate structures in the decoding process. For example, instead of decoding an MPEG frame directly into RGB format, Dalí breaks the process into three steps: bit stream decoding (including Huffman decoding and dequantization), frame reconstruction (motion compensation and IDCT), and color space conversion. For example, the `MpegPicParseP` function parses a P frame from a `BitStream` and writes the results into three `SCImages` and one `VectorImage`. A second primitive reconstructs pixel data from `SCImage` and `VectorImage` data, and a third converts between color spaces. The important point is that Dalí exposes the intermediate data structures, which allows the programmer to exploit optimizations that are normally impossible. For example, to decode gray scale data, one simply skips the frame reconstruction step on the U/V planes. Furthermore, compressed domain processing techniques can be applied on the `SCImage` or `VectorImage` structures.

Dalí also exposes the structure of the underlying bit stream. As described in the introduction and section 2.2.2, Dalí provides operations to find structural elements in compressed bit streams.

This feature allows programmers to exploit knowledge about the underlying bit stream structure for better performance. For example, a program that searches for an event in an MPEG video stream might cull the data set by examining only the I-frames initially, since they are easily (and quickly) parsed, and compressed domain techniques can be applied. This optimization can give several orders of magnitude improvement in performance in some circumstances, but since other libraries hide the structure of the MPEG bit stream from the programmer, this optimization cannot be used. In Dalí, this optimization is trivial to exploit. The programmer can use the `MpegPicHdrFind` function to find a picture header, `MpegPicHdrParse` to decode it, and, if the *type* field in the decoded header indicates the picture that follows is an I-frame, call `MpegIPicParse` to decode the picture.

4. IMPLEMENTATION

Dalí is currently implemented as a C run time library with approximately 50K lines of code. A Tcl 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-1 and AVI. The code can be downloaded from `http://www.cs.cornell.edu/dali/`

One might wonder whether the layered architecture of Dalí has any negative impact on performance. To answer this question, we compared three programs written in Dalí to similar programs widely used in the research community. These benchmarks include the Berkeley MPEG decoder, the IJG JPEG encoder, and a use of the NETPBM toolkit. Our results show that Dalí performs as well as these programs or better.

4.1. NETPBM

To compare Dalí with NETPBM, we used the following task: convert a 1600x1200 GIF image to a 320x240 gray scale image. On a Sparc 20 workstation, the command `giftopnm input.gif | ppmtopgm | pnmscale 0.2 > /dev/null` takes 2.6 seconds. The Dalí program that performs the same function takes 1.5 seconds and is about 110 lines long.

The Dalí program performs better because the implementation of NETPBM is not optimized and overhead is incurred when data is piped from one program to another. These shortcomings could be addressed by writing a single C program that combines code from `giftopnm.c`, `ppmtopgm.c`, and `pnmscale.c`, but this is a time-consuming task. In contrast, the Dalí program to perform the task can be easily optimized. For example, since Dalí exposes the color table of a GIF image (as an `ImageMap`), we can perform the RGB-to-gray conversion on the color table instead of the RGB image. This modification improved the performance by 13% and required changing four lines of code.

4.2. MPEG decoder

We compared Dalí with Berkeley MPEG decoder (`mpeg_play`). Our full function MPEG to PPM converter required about 150 lines of Dalí code. On a Sparc 20 workstation, the Dalí program ran about 10% faster than the `mpeg_play` [c] on a large variety of streams. We believe that Dalí's specialized primitives for decoding I, P, and B Frames contributes to the gain in performances.

4.3. JPEG encoding

Dalí JPEG encoding performance is comparable to the Independent JPEG Group's encoder (`cjpeg`). The IJG encoder will compress a 1600x1200 PPM image in 1.0 seconds of CPU time on a Pentium II 266 MHz WinNT workstation with 64MB of memory. The straightforward version of the equivalent Dalí encoder, which reads the whole PPM image into three `ByteImages`, converts them into the YUV color space, performs the DCT, and encodes the result, takes about 20% longer. We believe that the data copies associated with demultiplexing the RGB data in the I/O buffers into the `ByteImages` is responsible for the lesser performance of our version[d].

Not satisfied with this result, we rewrote the Dalí encoder to divide the `ByteImages` into horizontal strips using Dalí's clipping mechanism. We then perform color-space conversion, DCT, and bitstream encoding on each strip separately. This design gives superior caching performance. The improved version of JPEG encoder takes 1.0 seconds to encode the image.

---

[c] For Berkeley MPEG player, we used `mpeg_play -dither color -no_display`

[d] The Dalí casting mechanism could not be used in this case because the PPM data contains interleaved RGB data. We intend to rewrite the `ByteImage` abstraction to store and access interleaved RGB data. The mechanism will be similar to the way `AudioBuffers` store mono or stereo data in a single structure.

These experiments show that the design principles that we adopted for Dalí do not hurt performance. Rather, they allow flexible, optimized programs to be constructed with minimal effort.

5. CONCLUSION

The multimedia research community has traditionally built their software from scratch in C or by using high-level libraries. We believe that neither approach is satisfactory. We therefore developed Dalí, a software library for high-performance multimedia processing that provides lower level abstractions than most libraries, but much higher level than C. Dalí is designed for high-performance, and is based on several design principles such as explicit resource management, resource sharing and thin operations.

This paper described Dalí through examples and presented the design principles that make Dalí a high performance library. Our contribution is one of engineering, not scientific, research. We think that Dalí will prove to be a useful tool to the community because it allows efficient, processing-intensive multimedia software to be built with relatively small effort, and provides a vehicle through which research groups building this software can exchange their results.

5.1. Related Work

There are countless libraries for processing multimedia data (e.g., NETPBM[12], IJG JPEG[6], gd[5], ooMPEG[13]). Most libraries either provide a high-level API or work for a specific data format. Dalí is the first library that attempts to provide an efficient API that supports multiple multimedia data formats.

Others have proposed scripting language (VideoScheme[10], Isis[7], Rivl[21]) for processing multimedia data. Scripting languages are typically high-level, weakly typed, interpreted languages that support the composition of components and rapid prototyping. These languages provide high-level commands for manipulation of multimedia data. Isis and VideoScheme do not address performance issues in processing. Rivl addresses performance issues by using optimization techniques such as lazy evaluation and memory management. However, the optimizations that Rivl can perform are limited because Rivl combines the interpreter, optimizer and execution engine into one single system. This combined function makes it difficult to extend and debug. In fact, Dalí is initially designed to be compiler target for the Rivl compiler to address these problems.

PPE[17] is a multimedia toolkit that proposed using composable components to construct multimedia software. PPE components are lower level than Dalí. For example, PPE include components such as Huffman decoder, zigzag decoder, and IDCT decoder that can be pipelined to build a JPEG decoder. However PPE is designed to provide adaptive and configurable modules that can adapt themselves to heterogeneous environment, and not for more general multimedia processing. PPE is meant for building decoders whose component can be easily replaced. For instance, a fast, inaccurate IDCT can be used instead of a slower, more accurate IDCT when CPU power is limited. In contrast, Dalí is designed to construct more complex applications than PPE and to allow the program to exploit high-level optimizations.

5.2. Future Work

We are currently enhancing and Dalí with support for more multimedia data types such as MPEG-2, MPEG-4 and H.263. A Java binding is also under development. We plan to build a multithreaded implementation of Dalí in the near future. Dalí's clipping mechanism will allow us to easily exploit the parallelism inherent in most multimedia processing. For example, we can clip a `ByteImage` into small strips and process them in parallel using separate threads.

We are integrating Dalí into the Mash[9] toolkit. 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.

ACKNOWLEDGEMENTS

We would like to thank the Dalí team: Sugata Mukhopadhyay, Haye Hsi Chan, Tibor Janosi, Steve Weiss, Matthew Chew, Jose Machuca, Jiesang Song and Daniel Rabinovitz for implementation of Dalí. This research was supported by DARPA/ONR (contract N00014-95-1-0799), and grants from the National Science Foundation, Kodak, Intel, Xerox, and Microsoft.

REFERENCES

1. S. Acharya, B. Smith, *Compressed Domain Transcoding of MPEG*. Proceedings of the International Conference on Multimedia Computing and Systems (ICMCS) 1998, Austin, Texas. June 1998.
2. E. Amir, S. McCanne, H. Zhang, *An Application Level Video Gateway*. Proceedings of ACM Multimedia '95, San Francisco, California, November 1995.
3. J. Carter, W. Hsieh, M. Swanson, A. Davis, M. Parker, L. Schaelicke, L. Stoller, T. Tateyama, and L. Zhang, *Memory System Support for Irregular Applications*, Fourth ACM Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR '98), Carnegie Mellon University, Pittsburgh, PA, USA, May 28-30, 1998 May 1998. `http://www2.cs.utah.edu/impulse/publications.html`
4. CU-SeeMe, `http://www.wpine.com/Products/CU-SeeMe/`
5. gd GIF library, `http://www.boutell.com/gd.`
6. Independent JPEG Group software, release 6b, March 1998 `ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz`
7. Isis. *A multilevel scripting environment for responsive multimedia*. `http://isis.www.media.mit.edu/projects/isis/`
8. D. Le Gall, *MPEG: A Video Compression Standard for Multimedia Applications*, Communications of the ACM, pp. 46-58, Vol. 34, Num.4, April 1991.
9. S. McCanne et. al., *Toward a Common Infrastructure for Multimedia-Networking Middleware*. In Proc. 7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97), St. Louis, Missouri, May 1997.
10. J. Matthews, P. Gloor, F. Makedon, *VideoScheme: A Programmable Video Editing System for Automation and Media Recognition*. ACM Multimedia '93, August 1993, Anaheim, CA, pp. 419-426.
11. Microsoft NetMeeting, `http://www.microsoft.com/netmeeting`
12. NETPBM Graphics Package, March 1994, `ftp://ftp.cs.ubc.ca/ftp/archive/netpbm/netpbm-1mar1994.tar.gz`
13. ooMPEG: Object-oriented MPEG Decoder, `http://www.cs.brown.edu/software/ooMPEG/`
14. J. Ousterhout, *Tcl and the Tk Toolkit*. Addison-Wesley, Massachusetts, 1994.
15. W. Pennebaker, *JPEG Still Image Data Compression Standard*, Van Nos and Reinhold, New York, 1992.
16. K. Patel, B. Smith, L. Rowe, *Performance of a Software MPEG Video Decoder*, Proc. of the First ACM International Conference on Multimedia, pp. 75-82, Anaheim, CA, August 1-6, 1993.
17. N. Patel, I. Sethi, *Compressed Video Processing for Cut Detection.* IEEE Proceedings: Vision, Image and Signal Processing, pp. 315-323, Vol. 143, October 1996
18. E. Posnak, R. Lavender, H. Vin, An *Adaptive Framework for Developing Multimedia Software Components.* Communications of the ACM, October 1997.
19. B. Shen, I. Sethi, *Convolution-Based Edge Detection for Image/Video in Block DCT Domain*, Journal of Visual Communication and Image Representation, Vol. 7, No. 4, pp. 411-423, 1996.
20. B. Smith, L. Rowe, *Compressed Domain Processing of JPEG-encoded Images*, Real-Time Imaging, pp. 3-17,Vol. 1, Num. 2, July 1996.
21. J. Swartz, B. Smith, A *Resolution Independent Video Language*. Proceedings of ACM Multimedia Conference. San Francisco, California, 1995.
22. T. Wong et. al., *Software-only video production switcher for the Internet MBone.* Proceedings of Multimedia Computing and Networking, pp. 28-39, San Jose, California, January 1998.