Presentation Processing Support for Adaptive Multimedia Applications*

Edward J. Posnak, **Harrick M. Vin**, and **Robert G. Lavender**

Distributed Multimedia Computing Laboratory
Department of Computer Sciences, University of Texas at Austin
Taylor Hall 2.124, Austin, Texas 78712-1188
E-mail: {ejp,vin,lavender}@cs.utexas.edu, Telephone: (512) 471-9732, Fax: (512) 471-8885

ABSTRACT

In this paper, we describe a method for implementing Presentation Processing Engine (PPE) modules that allow applications to process media objects and control the frame rate, spatial resolution, and SNR. PPE modules implement compression/decompression algorithms (e.g., JPEG, MPEG, etc.) as well as image processing functions such as rotate, scale, and dither. We have developed a library of reusable compression and image processing modules that can be composed together to implement PPEs. The PPE implementation is bound at run-time according to the application's QoS requirements, available resources, and the compression format of the media object. PPEs can be dynamically reconfigured to adjust to changes in the quality of service required. These capabilities allow the PPE to tailor the access and delivery of the objects to the computing and communication capabilities of client sites as well as adapt to changes in resource availability and user preferences. Our compositional approach to building PPEs promises to achieve (1) a significant amount of functional reuse without sacrificing the ability to make low level modifications to the PPE implementation and (2) significant performance gains by allowing image processing functions to be inserted into processing pipelines at intermediate stages of compression.

1 Introduction

Recent advances in computing and communication technologies have made it economically viable to design distributed multimedia information systems that promise to enhance users' ability to access a wealth of audio, video, and textual information over high speed networks. To realize this vision, such systems must provide methods for servicing clients over a wide range of heterogeneous computing and communication environments (ranging from hand-held devices to powerful workstations, and from the huge installed base of ethernets, token rings, telephone lines, etc. to high speed and wireless networks). To do so, multimedia systems will maintain media objects at various levels of resolution, so that they can tailor the access and delivery of the objects to the computing and communication capabilities of client sites. Additionally, they will provide mechanisms that allow applications to adapt to fluctuations in the availability of computing and communication resources as well as to changes in the quality of service (QoS) desired by users.

To effectively utilize such capabilities as well as to enable complete integration of audio, video, and other such data types into the computing environment, applications must employ presentation processing mechanisms that can: (1) dynamically change the quality of presentation (by appropriately adjusting the compression/decompression modules), and (2) process these data types using a variety of operations (e.g., scaling, clipping, etc.). Additionally, as compression technology

*This research was supported in part by IBM, Intel, the National Science Foundation (Research Initiation Award CCR-9409666), NASA, Mitsubishi Electric Research Laboratories (MERL), Sun Microsystems Inc., and the University of Texas at Austin.

evolves and encoding formats proliferate, it becomes increasingly important that presentation processing mechanisms decouple applications from compression by providing a uniform interface for invoking such operations. The design and implementation of flexible, configurable presentation processing mechanisms that achieve these objectives is the subject matter of this paper.

Experiences with implementing presentation processing support for multimedia applications has clearly identified the need for increased flexibility. The first generation of internet conferencing applications, such as nv[?] and ivs,[?] were designed under constraints to facilitate implementation and experimentation, and consequently, were tightly coupled to specific compression algorithms and chroma formats. This resulted in their lack of interoperability with each other and with other compression standards that are becoming widely used. This experience has greatly influenced the design of next generation tools such as vic,[?] which employs a more flexible, modular approach that allows a variety of codecs to be supported. Similarly, the need for flexibility is evidenced in the design of systems such as Quicktime,[1] the VuSystem[9] and the Berkeley Continuous Media Toolkit.[6] These systems allow applications to access to multimedia data, independent of its compressed format, by making the codec a configurable component of a highly modular framework.

We propose that extending configurability down to the level of compression components such as DCT, quantization, and color conversion, provides additional benefits, both in terms of QoS control and performance. Recent work on QoS Filters has shown that finer-grained control over QoS can be realized by such methods as re-quantization, filtering of frequency coefficients, adding resynchronization markers etc..[?] Whereas QoS Filters are primarily targeted at the adaptation of compressed streams to cope with heterogeneity of networks, this same mechanisms can be applied at the client site to provide user control over QoS parameters and adaptation to end-system capabilities and changing resource availability. For example, a framework that allows various DCT, and dithering modules to be changed dynamically can provide applications a with wider range of selections in the frame-rate vs SNR space.

Moreover, the ability to control modules at this level can lead to significant performance gains. Experiences with implementing transcoders and image processing libraries have shown that operating on data in semi-compressed stages, rather than full decompression/recompression vastly improves performance.[?,8] While it is possible to build such hooks into monolithic codec implementations, our approach is to build a framework that provides reusable and extensible, rather than ad-hoc, mechanisms for configuring presentation processing components.

In this paper, we describe a method for implementing *Presentation Processing Engine (PPE)* modules that decouple media processing applications from compression format and provide greater flexibility in controlling the frame rate, spatial resolution, and SNR. PPE modules implement compression/decompression algorithms (e.g., JPEG,[11] MPEG[2] etc.) and image processing functions such as translation, scaling, clipping, etc. We have developed a library of reusable compression and image processing modules that can be composed together to implement flexible and efficient PPEs. The PPE implementation is bound at run-time according to the compression format of the media object, available resources, and the application's QoS requirements. PPEs can be reconfigured during playback to adjust to changes in the quality of service required. These capabilities allow the PPE to address the requirements imposed by heterogeneous environments as well as adapt to changes in resource availability and user preferences.

Our fine-grained compositional approach to building PPEs achieves 1) Increased flexibility for changing QoS and adapting to heterogeneous environments 2) significant performance gains by allowing manipulation of media at intermediate stages of compression 3) software engineering benefits of a reusable architecture that exploits the similarities in codec design [use Lavender's words for this last sentence] (e.g. facilitates rapid prototyping of experimental codecs)

The remainder of this paper is organized as follows: Section 2 describes our application development environment and how the module library has been designed to facilitate the development of PPEs. In Section 3 we give examples that illustrate the process of composing PPEs from library modules, and demonstrate the power of our approach. Section 4 provides a performance analysis that justifies the viability of this approach and Section 5 summarizes our contributions and concludes the paper.

2 The Presentation Processing Engine

To facilitate the development of adaptive multimedia applications, a collection of modular, reusable, parameterized type templates are defined that permit the multimedia application programmer to define new presentation components through controlled type instantiation and type extension. The PPE environment provides a set of uniform interfaces for processing and controlling the presentation quality of multimedia information. The environment supports the *source-module-sink* pipeline model by permitting modules, which perform various operations on the data, to be inserted between sources (e.g., microphones, cameras, storage servers, etc.) and sinks (e.g., speakers, display devices, storage servers, etc.).[3,9,6]

Good object-oriented abstractions that enable a high degree of reconfigurability often come with an associated "abstraction cost." In systems development, performance considerations are a dominate requirement. Experienced practitioners know that object-oriented techniques often lead to abstractions that initially seem to be an elegant reflection of a problem domain, but are often grossly inefficient because of over-generalization and the associated efficiency cost of having to interact with multiple abstraction boundaries enforced for the sake of a high-degree of information hiding and excessive modularity.[?,?] The challenge for systems programmers developing networking and multimedia applications is to make appropriate abstraction choices that represent a carefully considered balance between the use of static binding and dynamic binding program components. As is well known, static binding enables a compiler to make efficient choices during code generation (e.g., inlining), while dynamic binding often results in less efficient run-time actions (e.g., dynamic method lookup and dispatch).

The current PPE framework is the result of successively refined object-oriented abstractions that have resulted from an evolutionary development experience in which an appropriate balance between statically versus dynamically bound computational components has arisen. The statically determined components are represented by a set of parameterized types that allow an efficient coupling of critical abstractions that have high performance demands and minimal reconfigurability demands. The ability to define parameterized types allows for "syntactic modularity" that facilitates composition of modular components during development, but the efficieny cost of the abstraction is minimal since it is "compiled away". The efficiency cost of those abstractions requiring dynamic reconfigurability are constructed so that the efficiency cost is appropriate to the degree of configurability required. Thus the PPE framework makes the same kind of abstraction tradeoffs that are evident in the ANSI C++ Standard Template Library (STL),[?,?] which is a model for how to preserve syntactic abstractions that balance static and dynamic binding choices while incurring minimal run-time overhead.[1]

A PPE is a collection of reusable, but tightly integrated modules that collectively can be used to implement compression/decompression algorithms (e.g., JPEG,[11] MPEG,[2] etc.) as well as image processing functions such as translation, scaling, clipping, etc. The control flow in a PPE is defined by defining connectivity between selected modules. The mechanism used to realize a control flow specification is a generalized `Attach()` method, defined by each PPE module type, that creates a data path from the module's output to another module's input.

Strong typing of module type parameters and interfaces, is used to eliminate the possibility of statically misconfigured modules. The type parameterization features of C++ are used to statically instantiate a specific module type based on its required input and output data types. Attempts to compose modules with mismatchted input and output types are detected as errors by the C++ compiler.

A module's interface can be extended through subclassing and the addition of new methods to accommodate any level of functional coupling between modules or to improve performance. Modules can also be parameterized by other modules, as an alternative to attaching to them. This feature can be used in combination with inline interface extensions to "collapse" the separate abstractions implementing a pipeline segment into an efficient implementation at run-time without sacrificing the conceptual benefits of abstraction and modularity.[2] When parameterization and inline methods are used in this fashion,

[1] STL is an efficient implementation of container abstractions (e.g., sets, vectors), iterator abstractions, and memory allocators supporting generic algorithms on containers. The STL abstraction boundaries are effectively compiled away.

[2] The advantages and disadvantges of compiler-based function inlining are often hotly debated. One should not assume that inlining, in general, results in better performance. Empirical evidence suggests that careful selection of which methods are inlined can have a positive benefit on performance, dependent on a compiler's register allocation algorithm and the features of the target

a static binding of the modules takes place, resulting in a tradeoff of configurability for performance. This optimization works well for bit stream filters and huffman decoders where performance is critical and dynamic configurability is of little value.

PPE modules are designed with a recursive structure that allows them to contain pipelines of other modules, so that any set of functions, regardless of complexity, can be composed into a reusable module. The ability to hierarchically compose modules effectively captures the functional overlap that exists at multiple levels in compression algorithms. For instance, whereas primitive level building blocks such as color conversion and huffman coding are widely used, more complex aggregate operations such as the DCT/quantize/zig-zag/run-length/huffman encode sequence are also found in a number of compression algorithms. Additionally, relatively large processing sequences can be reused in developing scalable codecs since each scalable profile is often a variation or extension of some base algorithm. The various configurations of these sequences, combined with a rule set for switching between them, make up a codec module, which is one component of the PPE module.

The implementation of the PPE is bound at run-time, and may change dynamically in response to a variety of events. When the source opens a media object on behalf of the application, it determines the compression format and set of substreams needed to meet the application's QoS requirements. This information initially determines the decoder implementation in the PPE. When the PPE is attached to a sink, it determines the target output type and may change its configuration accordingly. As the media stream is parsed, reconfigurations may occur in response to control information embedded in the stream (e.g. change in frame type from intra to predicted). When a change in QoS occurs, either due to client initiation or a change in resource availability, the PPE may need to replace internal modules (e.g. dithering, inverse DCT) with ones that produce different QoS characteristics and/or reconfigure to support a different scalable profile. Each of these changes represents a QoS tradeoff that is independent of the way the input stream was compressed. Such changes can can be captured in a state machine where each state represents a different configuration of modules corresponding to some QoS level. The state transitions are associated with a set of operations that specify how to obtain the new configuration from the existing one, i.e. by creating, attaching, detaching, and deleting modules. When the level of quality changes, a codec reconfigures itself by executing the operations corresponding to the appropriate state transition.

The mechanism used to enable dynamic configuration is the *control operation*, which is invoked by one module on another to convey actions to be carried out and any relevant data. For example, to change a codec's implementation to support a new QoS level, the PPE invokes a `SetQosLevel` control operation on the codec. The codec handles this control operation by executing reconfiguration actions corresponding to one of its state transitions. Control operations are effective at realizing the most efficient implementation of a service. For example, to support the scale operation, the PPE invokes a `Scale` control operation on the decoder, which attempts to reconfigure it modules to perform scaling in the frequency domain. If the decoder cannot support this control operation and returns a failure status, the PPE will then resort to a less efficient, spatial domain implementation. For other image processing options, the PPE may explicitly pass the module to the decoder, along with instructions for how the module should be inserted into the decoder's pipeline. Another use of control operations is to perform late bindings that optimize performance. For example, after the chrominance subsampling values have been parsed from the input stream a control operation can be used to notify the color conversion module, which can then instantiate an implementation optimized for those values. The PPE uses control operations to provide an extensible set of services to applications. Integrating new data processing operations into the PPE is simply a matter of defining new control operations and their handlers; no changes to the API are necessary. Module reconfiguration occurs transparently to the application, which performs standard data manipulation using control operations and controls QoS using abstract terms such as frame rate, spatial resolution, and SNR. By hiding its implementation details, the PPE provides a uniform interface for processing and controlling the presentation quality of multimedia information.

In summary, the salient features of this fine-grained, compositional approach to implementing PPEs are:

- The ability to configure codec components during playback provides increased flexibility for changing QoS parameters. More options in the frame rate vs. SNR tradeoff space are available than by traditional techniques that simply discard

---

architecture.?

parts of the bit stream. This provides the user greater control over presentation quality and facilitates development of applications that can adapt to heterogeneous environments.

- The ability to manipulate data at intermediate stages of compression results in significant performance improvements as compared to performing the same operations on uncompressed data.[?,8] For example, the shrink-by-2 operation can be performed roughly 5 times faster on run-length encoded images than on images that are not compressed.[7] Similarly, a JPEG/H.261 transcoder that operates on frequency coefficients can process 3 times as many frames per second than one that operates on spatial data. Given that presentation processing is the current bottleneck in communications performance, the runtime performance advantages of this approach make it a promising one for the design of future distributed multimedia applications.
- The architecture exploits the similarities in codec design to provide a framework that facilitates rapid development of new software codecs. Though minor differences in standards prohibit extensive code reuse, the framework provides much of the infrastructure required to build codecs, and the developer need only implement specific modules. We find this capability to be very useful for constructing prototype codecs that support our research in fault-tolerant storage and transmission.[4,10]

3 PPE Composition: Examples

This section describes how one would construct JPEG, MPEG and MPEG-2 decoders using the PPE framework. The first example shows how static and dynamic module composition is used to implement an efficient JPEG decoder. This example demonstrates how the capability to hierarchically and dynamically configure modules facilitates development of PPEs that can support efficient data manipulation operations and different levels of quality. The second example shows how an MPEG decoder can be implemented by reusing the architecture and implementing the MPEG-specific modules. The last example again demonstrates the reusability of the framework to build an MPEG-2 Spatial/SNR scalable decoder[5] that can reconfigure its implementation to support different scalable profiles.

3.1 JPEG Decoder

JPEG is a widely used standard for still image compression. A JPEG encoder fragments an image into 16x16 pixel blocks, called macroblocks, that are further divided into a number of 8x8 pixel blocks for different color planes such as Y, Cr, and Cb. The 8x8 blocks are separately transformed into the frequency domain using the discrete cosine transform (DCT). The resulting frequency coefficients are quantized to filter out less visually perceptible components of the signal and then scanned in zig-zag fashion to obtain an approximate ordering from lowest to highest frequency. The lowest frequency DC coefficient is difference encoded against the DC value in the previous block of the same color plane and the remaining AC coefficients are run-length encoded. Finally, the blocks are further compressed using an entropy coding algorithm such as huffman encoding.

Figure 1 illustrates how a JPEG decoder can be composed from various library modules. The input bitstream is parsed by the syntax decoder and separated into control data (e.g. frame size, chrominance subsampling, quantization tables, etc.) and actual coded image data. Control data is used to set decoder state variables and to determine module configurations. The coded image data is passed down the pipeline of modules, which reconstruct the uncoded display image. The syntax decoder is parameterized by a reusable input bitstream module that provides operations to extract bits from the stream. We are able to reuse this module in all of our codecs, despite the codec-specific differences in bytestream syntax (e.g. zero stuffing in JPEG, AAL headers in DSJ), because the differences have been factored into a filter module, which is used to parameterize the input bitstream.

The block decoder, shown in Figure 1(b), is a reusable module that reads variable length coded coefficient data from the input bitstream and performs the Huffman, run-length, difference, and zig-zag decoding operations to reconstruct an 8x8

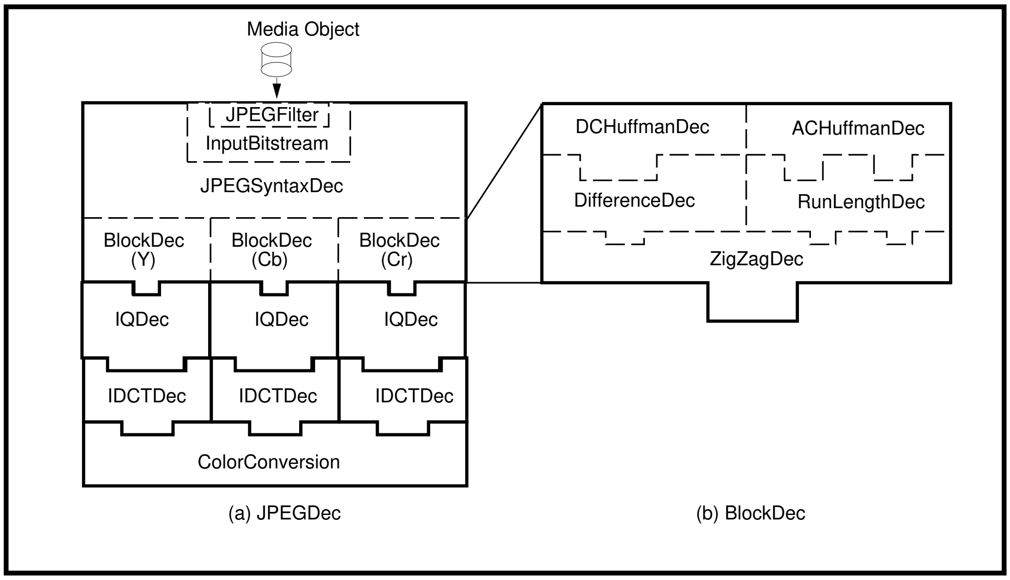

Figure 1 : Module Composition for a JPEG Decoder. (a) JPEG Decoder (b) block decoder module. Different shapes are used to indicate different module types. Modules dynamically bound via Attach are shown with a bold outline. Internal, statically bound modules are shown by a dashed outline.

block of coefficients. The actual Huffman decoders used for a given block may depend on whether the block is intra coded or predicted, whether it is a luminance or chrominance block, and whether run/level decoding is done JPEG or MPEG style. While it is possible to dynamically reconfigure a block decoder's internal modules to handle these cases, this naive approach is quite inefficient. Alternatively, we employ inlined interface extensions and module parameterization to effectively collapse the internal pipeline into a high performance, statically bound configuration. We create a number of block decoder types (e.g. intra-luminance, intra-chrominance, etc.), by parameterizing the block decoder with a pair of Huffman decoders that provide inline methods for decoding DC and AC coefficients respectively. Figure 2 shows graphically how different types of block decoders are generated using module parameterization. The templates on the left side of the figure are used to create specific modules, which are shown on the right. The figure illustrates how a JPEG luminance block decoder is composed of luminance AC and DC Huffman decoders, each of which is parameterized by an input bitstream, which itself contains a JPEG filter. It also shows how the same templates are used to build an MPEG specific block decoder. By factoring out the common code from the context-specific code, and inlining performance critical functions we can maintain a highly configurable architecture whose abstraction boundaries are enforced at compile time, but "compiled away" to produce efficient code. [3]

For the JPEG decoder we generate four Huffman decoders, which are used to parameterize two block decoders, one for luminance and one for chrominance. We then instantiate one luminance block decoder (Y) and two chrominance block decoders (Cb,Cr), each of which is attached to a dequantizer/inverse DCT pipeline. The pipelines of modules are shown having different shapes that fit together to illustrate the fact that strong typing is used to ensure that only feasible configurations are permitted. For example, the types of 8x8 blocks passed between the block decoder, inverse quantizer, inverse DCT and color conversion modules are purposely made distinct to prevent misconfigurations at compile time.

There are a number of advantages to using separate pipelines for the different color planes. Reconfiguring a video decoder to support a QoS change from color to monochrome is easily accomplished by replacing the Cb and Cr pipelines with null sink modules (and reconfiguring color conversion to optimize for this special case). Another advantage is the flexibility to apply data manipulation operations to specific planes. For example efficient brightening or darkening of an

[3] This technique has been successfully applied elsewhere to gain efficiency without losing modularity and the benefits of abstractions. The Standard Template Library? employs templates and heavy inlining in the ANSI C++ library.

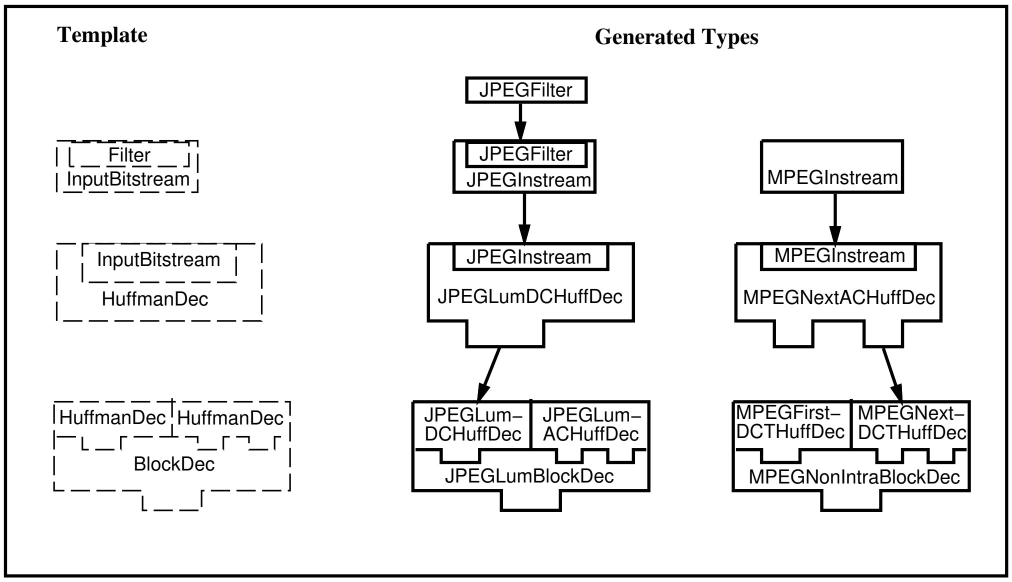

Figure 2 : Module composition using parameterization to generate a various types of input bitstreams, Huffman decoders and block decoders.

image or region can be performed by inserting a module in the luminance plane pipeline that increments the DC coefficient of the each block.[8] Other operations such as requantization and pixel subsampling, which are luminance/chrominance specific, can be implemented by inserting modules into these pipelines. Such flexibility facilitates the implementation of adaptive QoS filters and transcoders using configurable modules.

The color conversion module reconstructs the image in the target output format (e.g. colormapped, 24-bit RGB, monochrome, etc.). To support different display types, color conversion module must be able to attach to modules (i.e. window sinks) that accept different data types. When the output is a window sink, the output type is platform dependent and cannot be determined until run time. The color conversion module provides a control operation to set its output type, which can be invoked when the PPE attaches to the sink and the output type becomes known. When this control operation is handled the color conversion module binds its implementation based on the target output type. The use of control operations to perform late binding is also employed to select an optimal color conversion algorithm once the chrominance subsampling values become known.

This example has shown how static and dynamic module composition is used to implement a configurable decoder and illustrates the benefits of doing so with respect to QoS flexibility, performance and ease of implementation. Encoders, transcoders and QoS filters can be developed within this framework using a similar methodology. The next example shows how, due to structural similarity between codecs, we can reuse much of the design to construct an MPEG decoder.

3.2 MPEG Decoder

MPEG is a popular compression algorithm for digital motion video sequences. MPEG uses many of the same algorithms as JPEG to compress frames, but also exploits the temporal redundancy in video to improve compression efficiency. Intra coded (I) frames are compressed in a fashion similar to JPEG and are used as references for coding of forward predicted (P) frames and bi-directionally predicted (B) frames. Macroblocks in P frames may be intra coded or coded as differences from some macroblock in the previous I-frame, whose relative location is given by a motion vector. B frame macroblocks may be intra coded, predicted from either a past or future reference frame using motion vectors, or interpolated from both.

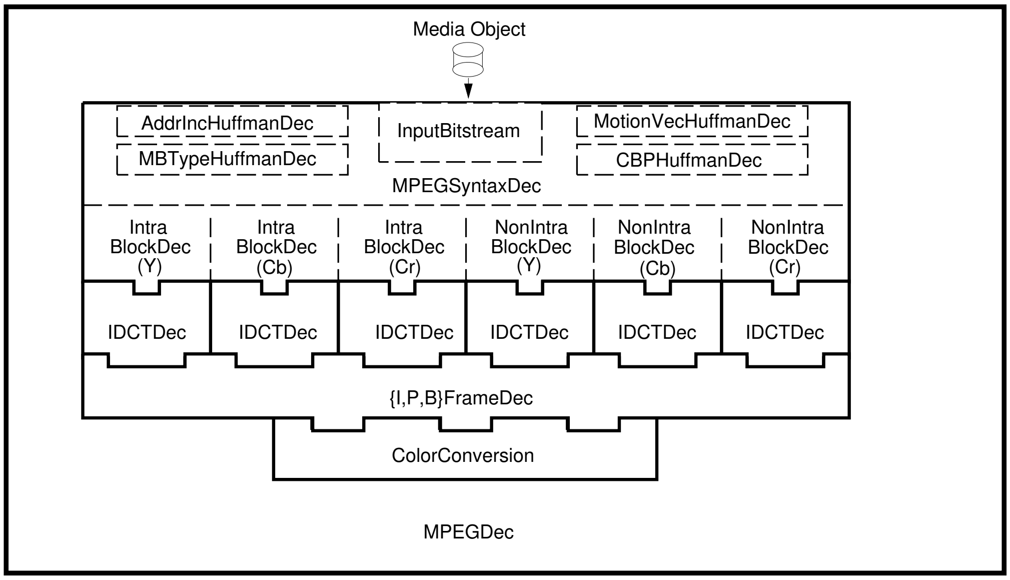

Figure 3 : Module Composition for an MPEG Decoder

Significant compression gain is achieved when P or B frame macroblocks are highly correlated with their reference frames and can be coded only as motion vectors, or skipped completely.

The architecture for an MPEG decoder, shown in Figure 3, is quite similar to that of our JPEG decoder. Again the input bitstream is parsed by the syntax decoder and separated into control data and coded image data. Huffman decoder modules for address increment, macroblock type, motion vectors and coded block pattern are used to parse encoded control data. As in the previous example, we develop high performance block decoder modules that are parameterized by DC and AC coefficient decoders. To gain further efficiency these block decoders are also parameterized with a dequantization module, so that dequantization/oddification/saturation is performed only on non-zero coefficients. This type of block decoder produces a different data type, a frequency block, than the one described in the previous example, which produces quantized frequency block. This strong typing helps prevent errors when attaching different block decoders to a dequantizer, I-DCT module, requantizer or coefficient adder as described in the next example.

The MPEG decoder pipeline includes a frame decoder that performs temporal prediction and motion compensation. There are three types of frame decoders, one each for I, P, and B frames, that are dynamically attached as each new frame type is parsed from the input stream. Each of the frame decoders reconstructs the Y, Cb, and Cr images for the color converter, using prediction and motion compensation when necessary. Since the frame type determines which of these operations are possible, as well as the semantics of skipped macroblocks and the types of optimizations that can be performed, we have found that decomposing the implementation into three separate frame decoder modules simplified and eliminated code. This is due to the fact that the frame type is known implicitly and need not be rediscovered at run time using switch statements.

As in the previous example, the color conversion module converts a reconstructed frame from YCbCr space to the target output format, and binds its implementation at run time when this information becomes known. For colormapped devices, dithering is used to remove banding effects and improve the quality of the image. However, various dithering algorithms can be employed to achieve different frame rate and SNR characteristics.[?] Control operations are provided to allow the color conversion module to dynamically change the dithering algorithm. This provides more flexible QoS control by allowing the frame rate and SNR characteristics of a video stream to be changed during playback.

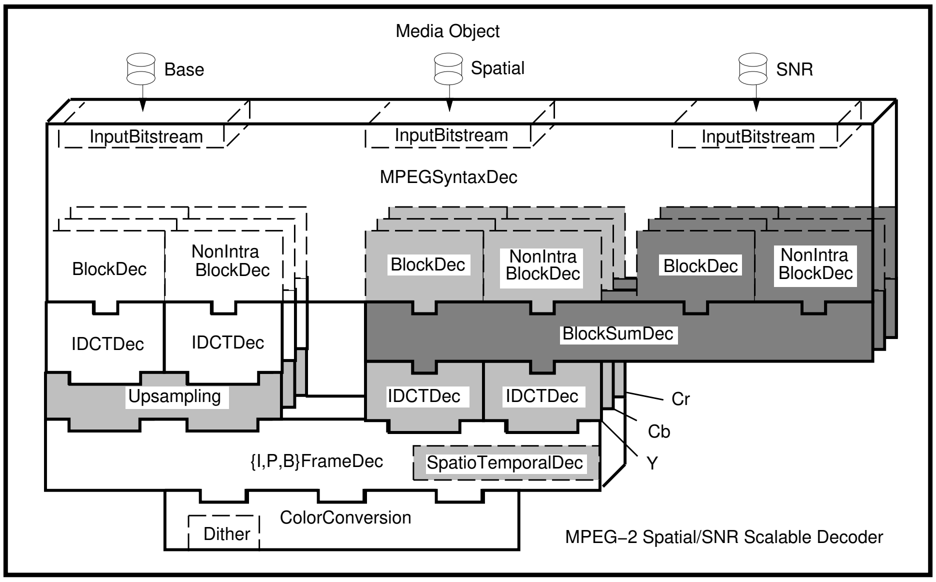

Figure 4 : Module Composition for an MPEG2 Spatial/SNR Decoder. Y, Cb, and Cr pipeline instances are shown in 3 dimensions

3.3 MPEG-2 Decoder

The next example shows how an MPEG-2 Spatial/SNR scalable decoder can be constructed by reusing and extending the architecture described in the previous examples. In this example we demonstrate how a state machine can be used to configure the codec to support different scalable profiles.

The MPEG-2 standard supports a number scalable profiles that allow a video stream to be delivered at multiple quality levels. In the spatial scalable profile, frames are encoded at two levels of spatial resolution. The base layer provides a low-spatial resolution image that can be decoded independently, or upsampled and combined with enhancement layer data to produce a higher resolution image. A macroblock in the enhancement layer may be predicted from the base layer, from a previous full resolution reference frame using motion compensation, or may be a weighted sum of both. In the SNR scalable profile the base and enhancement layers code the same images at different levels of quantization. The base layer may be decoded independently or combined with enhancement layer data to produce a higher quality image. The hybrid Spatial/SNR scalable profile allows decoding at three quality levels using base, spatial enhancement, and SNR enhancement substreams. At the lowest quality level, the base substream is decoded independently. The base and spatial enhancement streams may be combined as in spatial scalable mode to produce a medium quality image. To achieve the highest quality image, the spatial and SNR enhancement streams may be added as in SNR scalable mode and then combined with the base layer as in spatial scalable mode.

A decoder that supports all three quality levels has three associated states, which we shall call base, spatial and SNR. Figure 4 shows the configuration of a decoder in the highest quality, or SNR state [4]. Many of the modules and configurations are similar to the decoders described in the previous examples. Since the spatial scalable profile uses temporal and spatial prediction, the a module has been added to the frame decoder to perform these operations. This module combines macroblocks decoded from base and spatial substreams to produce high resolution macroblocks. The SNR substream contains fine quantization data that must be added to the spatial enhancement layer before converting transform coefficients back to the spatial domain. The BlockSumDec module is used to add blocks of quantized coefficients from the spatial and SNR substreams and output this sum to the inverse DCT module for the spatial substream. This module can be inserted

[4] For brevity of discussion, we give a high level description that excludes functions such as field reinterlacing, combining multiple chroma formats etc.

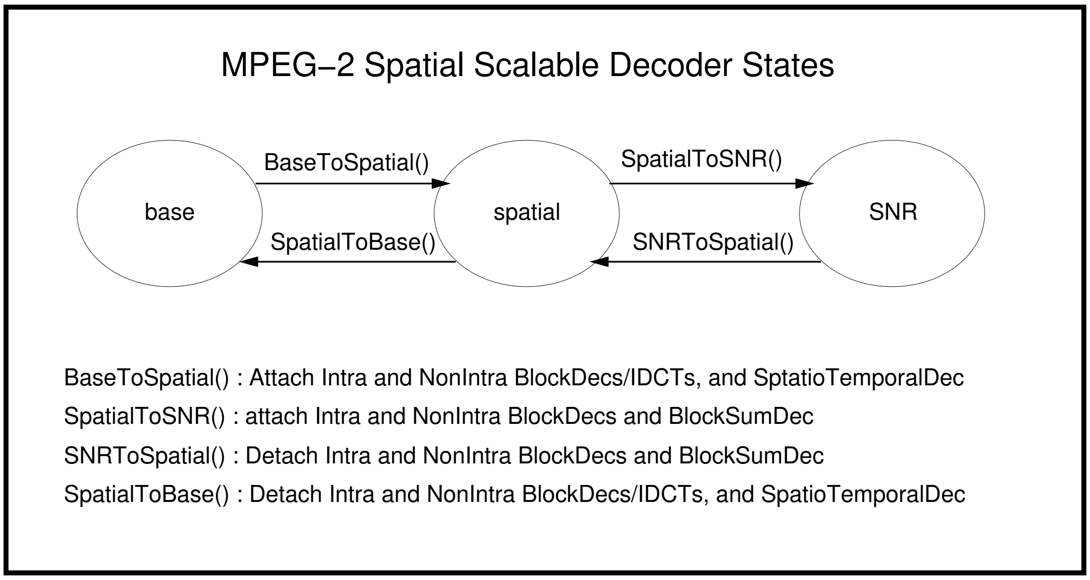

Figure 5 : State Machine for MPEG-2 Spatial/SNR Scalable Decoder

between the block decoders and the IDCT since it accepts FrequencyBlocks as input and produces them as output.

When a change in QoS requires switching between resolution levels, the protocol machine changes its configuration by adding, deleting and attaching modules. Figure 5 shows the decoder states associated with each resolution level and the configuration operations associated with each state transition. To go from high to medium resolution, the decoder transitions from SNR to spatial state by deleting the module sequence attached to the SNR substream and the block summing module. These modules are shown with dark shading in Figure 4. To switch to the base state, the module sequence attached to the spatial substream, the spatio-temporal prediction module and the upsampler are removed. These modules are shown with light shading in Figure 4. The frame decoders are replaced by the MPEG-1 frame decoders, which perform base layer temporal prediction and motion compensation.

The examples in this section have borne out the following benefits of our fine-grained modular framework for composition of PPEs.

- Increased flexibility for dynamically changing QoS parameters is made possible by (1) the capability to codec switch between scalable profiles by reconfiguring the decoder according to predefined state transitions, and (2) The ability to make frame rate vs SNR tradeoffs by replacing functionally similar modules with different algorithms such as inverse DCT and dithering.
- Improved performance is achieved by (1) allowing data manipulation at intermediate stages of decompression (e.g. by inserting a scaler in the decoding pipeline) (2) late binding of modules that can exploit knowledge about the coded bitstream to optimize their performance
- The implementation of new codecs is greatly simplified by a domain-specific framework that facilitates design reuse. Construction of JPEG, MPEG, and MPEG-2 decoders is accomplished by parameterizing the same basic architecture with compression standard specific algorithms, rather than building each decoder from scratch.

4 Performance Evaluation

In this section we evaluate the performance of the modular PPE architecture by comparing the JPEG and MPEG codec implementations described in the previous section to implementations that are available in the public domain. The fact that performance of the PPE is competitive with these implementations demonstrates the viability of the fine-grained compositional approach to building PPEs.

The JPEG decoder described in the previous section was implemented and compared to release 6 Independent JPEG Group's (IJG) JPEG implementation. In our experiments we used five different images, described in Table 1, to cover a wide range of image sizes and compression ratios. In each experiment the same image was decoded 100 times and the time to decode each frame was measured on an unloaded Sun Sparc-20 system running Solaris 2.4. We used all the fastest options for IJG, which included a fast IDCT that performed dequantization, a fast upsampling algorithm, and a color conversion algorithm optimized for the particular chrominance subampling. These same optimizations were also used in the PPE implementation. The minimum time to decode each image, by the PPE and IJG implementations is shown in Table 1.

Image	Dimensions	Bits/Pixel	PPE	IJG
jupiter	421x341	0.61	0.23	0.23
astrnaut	523x478	1.15	0.23	0.23
goldgate	640x480	1.23	0.23	0.23
lena	512x512	1.33	0.23	0.23
mirri3	551x579	1.90	0.23	0.23

These results indicate that the fine-grained modularity of the PPE does not adversely affect performance. To further evaluate the cost of modularity, as well as the overhead of dynamic configurability, we examined the PPE MPEG decoder, which dynamically configures its frame decoder with every new frame type. We implemented the PPE MPEG decoder, as described in the previous section, and compared its performance against the berkeley mpeg_play[?] implementation. Five test sequences were used, and each was decoded ten times. The highest frame rate for each decoder is shown in table 2.

Sequence	Dimensions	M:N	Bits/Pixel	PPE	mpeg_play
bicycle.mpg	352x240	4:2	0.45	30	30
flowg.mpg	352x240	1:2	1.78	30	30
jfk.mpg	192x144	1:2	0.79	30	30
tennis.mpg	352x240	1:2	0.79	30	30
zoom.mpg	256x192	1:2	0.64	30	30

These results show that the performance of a modular implementation is competetive with that of a monolithic codec optimized for performance. [Fill in analysis after experiments]

To quantitatively evaluate the framework's support for adaptive applications, we measured the variations in QoS that are achievable by reconfiguring DCT and dithering modules in our MPEG decoder. For each of the sequences used in the last experiment, we measured the frame rate and SNR achieved for each of the possible combinations of two different inverse DCT algorithms and three dithering algorithms. [5] IDCT-A is an accurate inverse DCT algorithm, used to obtain high quality reconstruction of images, whereas IDCT-B is a fast approximate algorithm which improves frame rate while sacrificing quality. Similarly, the 4x4 diffusion dither provides the highest quality at the lowest frame rate, whereas ordered dither improves the frame rate while sacrificing quality. No dither provides the highest frame rate, and lowest quality. Table 3 shows the SNR and frame rate achieved by each of the combinations for each of the video sequences.

FrameRate

[5] In all experiments the SNR calculation were made in the first run, and the frame rate not counted. SNR calculations were then turned off while measuring frame rate of subsequent runs.

Sequence	IDCT-A 4x4	IDCT-A ordered	IDCT-A none	IDCT-B 4x4	IDCT-B ordered	IDCT-B none
bicycle.mpg	30	30	30	30	30	30
flowg.mpg	30	30	30	30	30	30
jfk.mpg	30	30	30	30	30	30
tennis.mpg	30	30	30	30	30	30
zoom.mpg	30	30	30	30	30	30

SNR

Sequence	IDCT-A 4x4	IDCT-A ordered	IDCT-A none	IDCT-B 4x4	IDCT-B ordered	IDCT-B none
bicycle.mpg	40.0	40.0	40.0	40.0	40.0	40.0
flowg.mpg	40.0	40.0	40.0	40.0	40.0	40.0
jfk.mpg	40.0	40.0	40.0	40.0	40.0	40.0
tennis.mpg	40.0	40.0	40.0	40.0	40.0	40.0
zoom.mpg	40.0	40.0	40.0	40.0	40.0	40.0

These results indicate that the configurability of these modules provides a considerable amount of flexibility in trading off SNR for framerate. It has been our subjective evaluation that there are at least X distinguishable levels of quality that can be obtained from the 6 combinations of IDCT and dither modules. We plan to investiage how further flexibility can be obtained using other filtering methods such as using motion data only for certain predicted blocks. This optimization can eliminate IDCT, prediction and color conversion processing, thereby improving the frame-rate at the expense of SNR.

5 Conclusion

We have described the design of a presentation processing engine that provides flexible control over QoS parameters to allow applications to adapt to heterogeneous environments, fluctuations in resource availability, and changing client demands. We have shown, through examples, how the modular framework facilitates the development of PPEs that can dynamically bind and reconfigure their implementations to transparently provide applications with these capabilities.

The compositional approach to building PPEs has advantages in both software development and runtime performance. It allows data manipulation modules to be inserted at intermediate stages of compression, which can result in orders of magnitude speedup over functions that operate on uncompressed data.[8] The ability to reuse the PPE framework allows experimental codecs to be implemented much faster than building them from scratch. We plan to further investigate how codecs can use this flexibility to improve transmission performance and reliability by integrating transport level functions into the PPE. Given that compression technology is still evolving and presentation processing is the current bottleneck in communications performance, improvements in this area should have a significant impact on future distributed multimedia applications.

6 REFERENCES

[1] E. M. Hoffert et al. QuickTime: An Extensible Standard for Digital Multimedia. In *Proceedings of IEEE Compcon'92*, pages 15–20, 1992.

[2] D. Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):47–58, April 1991.

[3] S. Gibbs. Composite Multimedia and Active Objects. *OOPSLA '91*, pages 97–112, 1991.

[4] E. J. Posnak, S. P. Gallindo, A. P. Stephens, and H. M. Vin. Techniques for Resilient Transmission of JPEG Video

Streams. In *Proceedings of Multimedia Computing and Networking, San Jose, CA*, February 1995.

[5] A. Puri. Video Coding using the MPEG-2 Compression Standard. *Proceedings of SPIE Visual Communications and Image Processing*, 2094:1701–1713, Nov 1993.

[6] L.A. Rowe. Continuous Media Applications. Presented at Multipoint Workshop held in conjunction with ACM Multimedia '94., November 1994.

[7] B. Smith. Fast Software Processing of Motion JPEG Video. In *Proceedings of the ACM Multimedia'94*, pages 77–88, 1994.

[8] B. Smith and L. Rowe. Algorithms for Manipulating Compressed Images. *IEEE Computer Graphics and Applications*, pages 34–42, 1993.

[9] D. L. Tennenhouse and et. al. A Software-Oriented Approach to the Design of Media Processing Environments. In *International Conference on Multimedia Computing and Systems*, volume 1, pages 435–444, Boston, Massachusetts, May 1994.

[10] H. M. Vin, P. J. Shenoy, and S. Rao. Efficient Failure Recovery in Multi-Disk Multimedia Servers. In *Proceedings of the 25th International Symposium on Fault Tolerant Computing Systems*, Pasadena, CA, June 1995.

[11] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):31–44, April 1991.