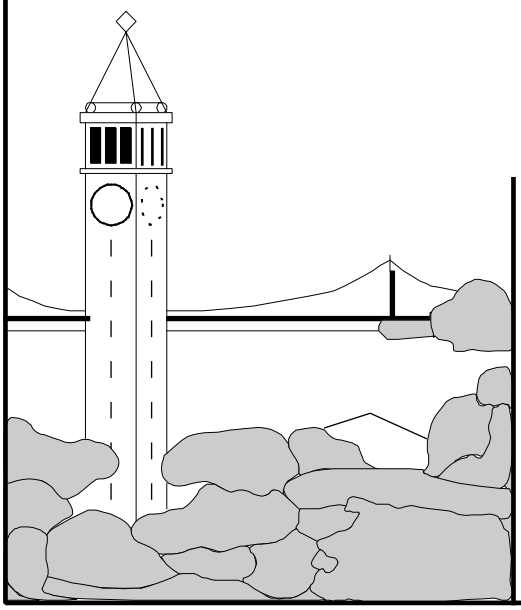
Design and Characterization of the Berkeley Multimedia Workload

Nathan T. Slingerland and Alan Jay Smith



Report No. UCB/CSD-00-1122

December 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Design and Characterization of the Berkeley Multimedia Workload

Nathan T. Slingerland and Alan Jay Smith
{slingn, smith}@cs.berkeley.edu

Computer Science Division
EECS Department
University of California at Berkeley

December 21, 2000

Abstract

The last decade has seen the integration of audio, video, and 3D graphics into existing workloads as well as the emergence of new workloads dominated by the processing of these forms of media. Unfortunately, widely accepted benchmarks which capture these new workloads in a realistic way have not emerged. The goal of this work is to present the Berkeley multimedia workload, which was developed in order to facilitate our own studies on architectural support for multimedia. Here we present a survey of existing multimedia benchmarking methods, a description of the Berkeley multimedia workload, as well as a full workload characterization including the extraction of computationally important multimedia kernels, and a detailed description of their constituent algorithms.

1 Introduction

What is multimedia and why is it important to study? We will define multimedia as proposed in [Bhas97]: an amalgamation of various data types such as audio, 2D and 3D graphics, animation, images and video within a computing system or within a user application. Put simply, a multimedia application is one which operates on data that takes form either visually or aurally. Multimedia applications are important to study because the multimedia industry is growing rapidly both in terms of consumer use and acceptance as well as economic impact. Figures 1a-1c graph current and projected trends for several key product areas in multimedia including digital video (DVD players, Figure 1a), digital imaging (digital cameras, Figure 1b), and digital audio (portable MP3 players, Figure 1c). Each of these industries relies either directly or indirectly on the ability of desktop computers to manipulate and process multimedia data. [Cont97] discusses some of the challenges involved in bringing these new workloads to desktop computers.

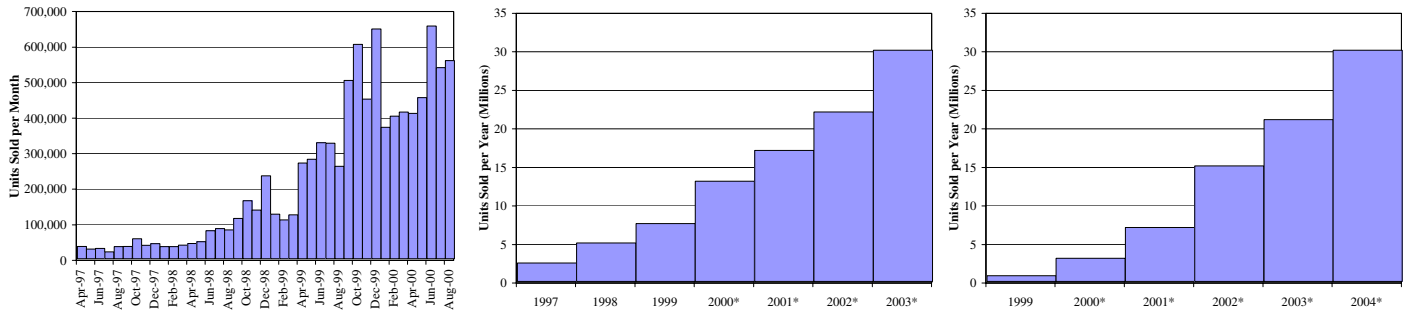
Funding for this research has been provided by the State of California under the MICRO program, and by Cisco Corporation, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Sun Microsystems, Toshiba Corporation and Veritas Software Corporation.

Benchmark performance is important for justifying the proposed architectural features to be included in a next generation processor. Many current microprocessor features (branch prediction, large caches, 32-bit and 64-bit data paths) have resulted from emphasis on performance for integer benchmarks such as SPECint92 and SPECint95 [Cont97]. Perhaps the most difficult task of any multimedia researcher is determining what applications constitute a multimedia workload. The lack of a standard, realistic multimedia workload has forced multimedia performance studies to focus either on existing digital signal processing kernels or on MPEG video playback ([Kuro98], [Lee95], [Zuck96]). Only recently have true multimedia application benchmarks begun to emerge ([Intel], [Lee97]). In order to construct a relevant and realistic workload we first survey what other researchers in multimedia have done in Section 2. We then present the Berkeley multimedia workload in Section 3, outlining the component applications and data sets, and analyzing it in Section 4.

2 Multimedia Workloads

The simplest types of benchmarks are toy benchmarks (algorithmic problems such as Towers of Hanoi, or Nine Queens), and synthetic benchmarks (small programs especially constructed for benchmarking which do not perform any useful computation in themselves, but are intended to statistically represent the average characteristics of a target workload) [Cont91]. Thus far, no studies of multimedia have employed either of these types of benchmarks. Instead, kernel benchmarks (code fragments extracted from real programs which are believed to represent a significant portion of the execution time of the original application) and sometimes application benchmarks (full applications performing a particular task with real input data sets) are used. Full application benchmarks are usually preferred to kernel benchmarks because they perform the actual task of interest in the same manner as the actual workload. Kernel benchmarks can be appropriate depending on the scope of the study as well as whether or not they are justifiably important within the workload they are intended to represent.

The importance of studying complete applications with re-



(a) DVD Player Sales

(b) Digital Camera Sales

(c) Portable MP3 Player Sales

Figure 1: **Multimedia Industry Trends [CEA], [Robe00], [Taka00]** - data denoted with (*) are projected values

alistic data sets can not be overstated. Small code sizes and undemanding data sets make coding easy and verification straightforward, but are not appropriate for performance analysis. A system's performance on a kernel which has been selected a priori without any supporting analysis is not a good indication of the actual end user's experience. Designing a useful application level benchmark consists of the selection of the applications to be included, and the data sets on which these applications will operate.

2.1 Media Libraries

Several studies have recoded DSP and multimedia algorithms (kernels) for various multimedia instruction set extensions in order to measure the performance benefits of such instructions. [Nguy99] recoded a set of DSP and multimedia algorithms in Motorola's AltiVec extension and measured the speedup over C. [Bhar98] analyzed Intel's MMX extension on a set of DSP kernels and applications taken from Intel's signal processing, recognition primitives and image processing libraries. [Rice96] quantified the performance of the VIS extension to Sun's UltraSPARC architecture on a set of image processing algorithms from Sun's XIL imaging foundation library. Sun's VIS extension was also examined in [Naka96] for a different set of imaging kernels. Unfortunately, this approach simply shifts the responsibility of kernel selection from the multimedia researcher to the library's author, who may or may not have relied upon an extensive and representative performance analysis of multimedia applications. The inclusion of a given function within a library intended for multimedia is not sufficient to conclude that the function is of *actual* performance importance, as the workload it is derived from is in most cases left a mystery.

2.2 DSP Benchmarks

Early attempts at integrating multimedia processing with desktop computers enhanced a general-purpose processor with a DSP as a media coprocessor [Lee96]. From this, it was a small jump from borrowing DSP hardware technology for desktop multimedia processing to borrowing DSP benchmarks to measure desktop media processing performance. DSP benchmarks are still being used (and in some cases created) for measuring desktop multimedia performance ([Bhar98], [Zivo94]). An example of a typical DSP specific benchmark, the BDTi benchmark, is given in Table 1 [BDTi97]. Although these kernels have been widely studied for DSP and are justifiably important in low power portable devices, it is not clear that they have the same importance in desktop multimedia applications. DSPs and general purpose processors are significantly different in their architectures and design tradeoffs presumably in order to support widely different workloads. DSPs typically do not support virtual memory and may not include data or instruction caches. DSPs have specialized addressing modes and pipelined multiply-accumulate operations; DSPs are often also designed for applications where power consumption and cost, rather than just performance, are important design guidelines.

2.3 Multimedia Benchmarks

2.3.1 MPEG

At the time of the introduction of the MPEG-1 standard, no one seriously considered anything but a dedicated hardware solution for MPEG video decoding. Even so, much of the original MPEG standard consisted of C based pseudo code. It was not long after its publication that it was found that real time MPEG playback could be done in software on high-end workstations [Rowe93]. Software based decoding is now a widespread solution to DVD viewing on personal computers (DVD video employs a subset of the MPEG-2 coding standard). The original work on software MPEG decoding also first used frames per second of SIF format (352x240 resolution) video decoding as a metric for measuring multimedia performance. Many architectural studies of multimedia have employed this metric. It is simple to understand and measure, and the underlying MPEG coding application is motivating because of the broadly attractive applications such as DVD and HDTV that it facilitates.

Software based MPEG decoding is a hard problem and not efficiently supported by traditional width (32-bit or 64-bit)

Kernel	Description	Example Applications
Real Block FIR	Finite Impulse Response filter, block real data	speech processing (e.g. G.728 speech coding)
Complex Block FIR	Finite Impulse Response filter, block complex data	modem channel equalization
Real Single Sample FIR	Finite Impulse Response filter, single sample real data	speech processing, general filtering
LMS Adaptive FIR	Least-mean-square adaptive filter, single sample real data	channel equalization, servo control, linear pred. coding
Real Single Sample IIR	Infinite Impulse Response filter, single sample real data	audio processing, general filtering
Vector Dot Product	Sum of the pointwise multiplications of two vectors	convolution, correlation, matrix multiplication
Vector Add	Pointwise addition of two vectors	graphics, mix audio signals or images, vector search
Vector Maximum	Find value and position of maximum element in vector	error control coding, block floating point algorithms
Convolutional Encoder	Convolutional forward error correction on a block of bits	North American digital cellular telephone equipment
Finite State Machine	Series of control and bit manipulation operations	Virtually all DSP applications include control operations
FFT	256-point, In place, Radix-2 fast Fourier transform	Radar, sonar, MPEG audio, spectral analysis

Table 1: **BDTi DSP Benchmark Suite [BDTi97]**

data paths, so it is fundamentally interesting. In addition, improvements to software MPEG decoding are not limited to this algorithm alone. Many other multimedia applications rely on the same algorithmic building blocks as MPEG. The H.261 and H.263 teleconferencing standards are both algorithmically similar to MPEG video coding - they are hybrid coding techniques utilizing a DCT scheme for intra-frame compression and block based motion compensation to exploit temporal redundancies for inter-frame compression. Because they are intended for video teleconferencing applications, H.261/H.263 are much more computationally symmetric algorithms than MPEG, which is an asymmetric encode once, decode many application [Arav93]. JPEG compression is algorithmically identical to the I frame format of MPEG.

Despite MPEG video's clear importance, it is not the only multimedia application worth studying. It is important not to ignore the area of broadband audio coding, as usable movie playback requires the concurrent decoding of a five channel sound track (*broadband audio* is a term used to differentiate sound signals which contains the full 20 Hz - 20 kHz range of human hearing from speech specific signals which are limited to the approximately 3.2 kHz wide band necessary for intelligible speech). Other important important industry driving media types include 3D graphics, and speech recognition and synthesis. Video games, which rely heavily on real time 3D graphics as well as other media types, drive technological change in the PC industry.

2.3.2 Intel Media Benchmark

Around the time of Intel's addition of MMX to their x86 architecture, they released the Intel Media Benchmark. Its development was spurred because an adequate industry standard multimedia benchmark did not exist to measure multimedia performance. Clearly targeted at the x86 based Wintel platform (no source code for the benchmark is publicly distributed), it is of little practical use for comparing Intel's MMX to the extensions designed by other vendors. What is interesting are the applications Intel included, as well as the weighting each section of their benchmark suite is given (Table 2). An Intel Media Benchmark score is computed by calculating the weighted geometric mean of the time of each of the component benchmarks. According to the white paper distributed on the World Wide Web, the weightings were based on feedback from leading multimedia software vendors [Intel].

Name	Description	Data Set
Video	MPEG-1 Video Decode(40%)	320x240 scaled 2x
Audio	MPEG-1 Audio Decode (20%) Audio Processing (5%)	30 sec, CD quality Sample rate conv. Special effects Mixing
Imaging	Photoshop type Filtering (5%) Box Filter - gaussian blur, emboss Image Compositing - blend Chroma Key - image overlay	Convolution Compositing Chroma Key Color space conv.
3D	Render Direct3D Scene (30%)	4 spheres, 2 lights

Table 2: **Intel Media Benchmark [Intel]**

The Intel Media Benchmark was innovative because large parts of it consist of real multimedia applications with actual multimedia data sets. As the benchmark is run, the results of the component applications are presented. For example, in the case of audio mixing, the resulting mixed audio is played through the test PC's sound card and speakers. This sets it apart from many benchmarks, in that it is possible to verify the result computed, rather than just time how long it took to execute. Distributing the benchmark only as an x86 executable is this benchmark's greatest shortcoming, limiting it to being used solely for comparisons among X86 instruction set compatible processors.

Although the Intel Media Benchmark source code is not publicly available, we did have access to it through an agreement with Intel. After examining the benchmark's contents, we chose not to modify or expand it for our work for several reasons. First, because the Microsoft Windows operating system is only available for x86 based machines (and to a limited extent on DEC Alpha machines), it would require a large effort to port the existing Windows oriented Intel Media Benchmark code to the other UNIX-based platforms that we desired to study. Most importantly, basing our work on the Intel Media Benchmark would have bound the resulting work-

load with the same closed distribution format as the original Intel Media Benchmark, precluding independent verification of our applications and data sets. We determined that a much better approach would be to utilize existing UNIX multimedia applications, as this would avoid the porting effort as well as leaving our product open source. Every hardware platform is capable of running some flavor of UNIX, but not necessarily Microsoft Windows, making a UNIX based benchmark much more attractive.

2.3.3 UCLA MediaBench

UCLA's MediaBench is a suite of media oriented applications and corresponding sample data sets designed to represent the workload of emerging multimedia and communications systems [Lee97]. The applications included in the UCLA MediaBench suite (Table 3) were selected through intuition and market driven selection on the part of its authors to represent what they considered the workload of emerging multimedia and communications systems. All of the component applications are publicly available as source code, and the complete set of applications and data sets are distributed on the World Wide Web.

The MediaBench workload represents a large portion of the types of applications that might be run on a desktop workstation or PC. The already discussed problems with DSP benchmarks and library kernels were avoided by instead using full applications and data sets. UCLA MediaBench includes image compression (JPEG, EPIC), video compression (MPEG-2), speech and audio coding (GSM, G.721, and ADPCM), speech recognition (Rasta) as well as cryptography (PGP, Pegwit). It also tests one document presentation application (ghostscript) and a popular 3D rendering API (Mesa).

What UCLA MediaBench gets right is the concept of giving application and data set selection equal importance. The benchmark is freely distributed in source code form on the world wide web along with the corresponding data sets. This open source approach facilitates confidence in the benchmark along with providing a means for critiquing the benchmark's contents. Direct cross architecture comparisons are possible for any platform capable of running UNIX or a similar operating system, although the benchmark does not specify a single numerical metric to make such a comparison. We opted not to simply use UCLA MediaBench for our study because several key multimedia applications have emerged since its inception (most notably, MP3 audio). Also, many of the data sets have aged such that they are unrealistically small compared to contemporary workloads. [Fritt99] addresses the first issue by extending UCLA MediaBench with a few other applications (H.263 video teleconferencing and MPEG-4), and notes the data set size problem. The original MediaBench is characterized in [Bish99].

3 Berkeley Multimedia Workload

We believe that the approach of MediaBench - full applications with well specified data sets - is correct. For this reason, we used UCLA MediaBench as a starting point for constructing our own multimedia workload. Open source software was used both for its portability (allowing for cross platform comparisons) and the fact that we could analyze the source code directly. The main driving force behind application selection was to strive for completeness in covering as many types of media processing as possible. We first surveyed and collected a broad sample of multimedia applications, which was distilled into a list of fundamental multimedia tasks: 1) MPEG video, 2) MPEG audio, 3) video games (including real time 3D graphics and music synthesis), 4) ray tracing (high quality static 3D rendering), 5) speech recognition and synthesis, 6) telephony (video and speech compression), and 7) electronic document viewing and rendering (e.g. PostScript and HTML documents, digital photographs).

For each task, a small subset of applications were chosen based on several factors including apparent popularity (based on personal experience and user discussions on the WWW), code quality (robustness, performance) and level of code maintenance. In most cases, there was a single application that was clearly superior to its competitors based on these metrics. It is our hope that in making our workload fully open and available that the process of peer review can serve to draw attention to any deficiencies in this workload. It is not our contention that this workload is ideal for every study of multimedia. The greatest advantage of making the Berkeley multimedia workload completely open is that anyone is free to take our workload and modify it to suit their needs.

The applications which make up the Berkeley multimedia workload are presented in Table 4, which includes image compression (DjVu, JPEG), 3D graphics (Mesa, POVray), document rendering (Ghostscript), music synthesis (Timidity), audio compression (ADPCM, LAME, mpg123), speech synthesis (Rsynth), speech compression (GSM), speech recognition (Rasta) and video game (Doom) applications. Three MPEG-2 data sets are included to cover DVD and HDTV (720P, 1080I) resolutions. A detailed description of each component application and data set can be found in Appendix B.

4 Workload Characterization

The characterization of a benchmark involves the measurement of its runtime behavior with our specific goal being to provide insight into supporting multimedia with hardware architectures. In order to provide a baseline against which to compare our characterization of the Berkeley multimedia workload, a side by side characterization in the same manner will be performed with another widely studied workload, the SPEC95 benchmark suite. All of our workload measurements were performed on the DEC (now Compaq) Alpha platform using the ATOM instrumentation tool from the Developer's Toolkit for Digital Unix (now Compaq Tru64 Unix). ATOM allows for the construction of customized instrumentation and profiling tools, as well as including a variety of standard prepackaged tools such as `gprof` and `pixie` [DEC].

The component applications for both the multimedia work-

Name	Description	Data Set
ADPCM	IMA reference ADPCM audio compression	Speech by U.S. President Clinton, 18 sec., 16-bit, 8,000 Hz
EPIC	Experimental wavelet image compression	256x256 greyscale image
G.721	CCITT reference ADPCM speech compression	Speech by U.S. President Clinton, 18 sec., 16-bit, 8,000 Hz
Ghostscript	Postscript interpreter for rendering postscript files	Color postscript drawing of a tiger
GSM	European GSM 06.10 full rate speech compression	Speech by U.S. President Clinton, 18 sec., 16-bit, 8,000 Hz
JPEG	DCT based lossy image compression	227x149 color image
Mesa	OpenGL 3D Rendering API Clone	Mipmap - texture mapping with mipmaps Osdemo - standard rendering pipeline Texgen - texture mapped version of Utah teapot
MPEG	MPEG-2 video coding	Four frames at SIF (352x240) resolution
PEGWIT	SHA1 public key encryption and authentication	SpixTools output statistics file
PGP	"Pretty Good Privacy" data encryption	SpixTools output statistics file
Rasta	Speech recognition	2.128 second SPHERE format speech file: "Laurie?...Yeah...Oh."

Table 3: **UCLA MediaBench**

Name	Description	Data Set
ADPCM	IMA ADPCM audio compression	Excerpt from Shchedrin's Carmen Suite, 28 sec., Mono, 16-bits, 44 kHz [Pope94]
DVJU	AT&T IW44 wavelet image compression	491x726 color digital photographic image [Kodak]
Doom	Classic first person shooter video game	25.8 sec. recorded game sequence (774 frames @ 30 fps)
Ghostscript	Postscript document viewing/rendering	First page of Rosenblum and Ousterhout's LFS paper (24.8 kBytes) [Rose92]
GSM	European GSM 06.10 speech compression	Speech by U.S. Vice President Gore, 24 sec., Mono, 16-bits, 8 kHz [CNN99]
JPEG	DCT based lossy image compression	491x726 color digital photographic image [Kodak]
LAME	MPEG-1 Layer III (MP3) audio encoder	Excerpt from Shchedrin's Carmen Suite, 28 sec., Stereo, 16-bits, 44 kHz [Pope94]
Mesa	OpenGL 3D rendering API clone	Animated gears, morph3d, reflect demos - 30 frames each at 1024x768
MPEG-2	MPEG-2 video encoding	16 frames (1 GOP) at DVD, HDTV 720P, HDTV 1080I resolutions
mpg123	MPEG-1 Layer III (MP3) audio decoder	Excerpt from Shchedrin's Carmen Suite, 28 sec., Stereo, 16-bits, 44 kHz [Pope94]
POVray	Persistance of Vision ray tracer	640x480 Ammonite scene by artist Robert A. Mickelsen [Mick95]
Rasta	Speech recognition	2.128 sec. SPHERE audio file: "Laurie?...Yeah...Oh."
Rsynth	Klatt speech synthesizer	181 word excerpt of U.S. Declaration of Independence (90 sec., 1,062 bytes)
Timidity	MIDI music rendering with GUS instruments	X-files theme song, MIDI file (49 sec., 13,894 bytes), Goemon patch kit [Snow96]

Table 4: **Berkeley Multimedia Workload**

load and SPEC95 were compiled with Compaq C V6.1-011 or GCC v2.8.1 on Compaq Tru64 UNIX V5.0 (Rev. 910) with the compiler flags: `-O2 -g3 -non_shared`. These parameters turn on only those optimizations which which are not architecture specific - those which eliminate redundancies in the code at the assembly level (common sub-expression elimination, constant propagation), rather than more sophisticated optimization methods (loop unrolling, procedure inlining, global scheduling) which can potentially affect the workload in an architecture specific way [Fritt99]. All of the component application codes were originally distributed with optimization levels less than or equal to this, making the codes in our study at least as well optimized as the binaries that are in wide use. All of the compiled binaries were instrumented with our tools using ATOM and run on a Compaq model DS20 workstation with dual 500 MHz Alpha 21264 processors and 1 GB of RAM. Table 5 overviews the basic characteristics of the Berkeley multimedia workload.

4.1 Operation Mix

The balance of functional resources utilized by a workload determines the optimal distribution of functional units (adders, multipliers, load/store units) as well as buffer lengths and other important architectural parameters. The number of functional units available determines how instructions are scheduled, and is limited by available die area as well as limitations due to added wire length and cycle time. Having too few of the needed functional units lengthens the execution time of a program due to the limited ability to extract parallelism, while on the other extreme, implementing too many functional units results in under utilized resources which consume additional area, reduce yield and increase wire length and cycle time [Fritt99]. Table 6 compares the relative mix of a variety of instruction types for the Berkeley multimedia workload, UCLA MediaBench+ [Fritt99] and the SPEC95 benchmark suite as well as the instruction mix for each component application both workloads. Note that this data reflects standard compilers, and does not include any specialized multimedia instructions. From this table it would appear that although there are overall similarities between the workloads,

Name	Source Lines	Instructions		Executable Size (bytes)		Loads	Stores	User Time		System Time	
		Static	Dynamic	Static	Dynamic			(sec)	(M cycles)	(sec)	(M cycles)
ADPCM Encode	300	478	64,020,339	32,768	1,507,328	4,302,782	616,116	0.116	53.0	0.037	18.5
ADPCM Decode	300	478	49,687,192	32,768	1,507,328	4,302,782	1,229,491	0.063	31.5	0.086	43.0
DJVU Encode	25,419	131,541	394,242,073	1,318,912	42,663,936	68,204,647	27,458,767	0.696	348.0	0.020	10.0
DJVU Decode	25,419	127,475	328,761,829	1,277,952	21,495,808	59,700,283	31,845,270	0.492	246.0	0.030	15.0
Doom	57,868	295,086	1,889,897,116	2,359,296	27,066,368	500,225,773	109,222,846	2.134	1,067.0	0.954	477.0
Ghostscript*	248,363	326,773	970,395,449	4,579,328	32,964,608	188,116,952	96,837,718	1.175	587.5	0.129	64.5
GSM Encode	5,473	54,228	375,971,389	385,024	1,048,576	55,009,077	14,010,892	0.468	234.0	0.003	1.5
GSM Decode	5,473	54,228	126,489,950	385,024	1,048,576	10,711,683	3,812,483	0.186	93.0	0.000	0.0
JPEG Encode	33,714	75,460	177,977,854	557,056	11,141,120	41,182,069	14,156,413	0.218	109.0	0.012	6.0
JPEG Decode	33,714	81,382	80,176,365	598,016	11,141,120	16,419,065	4,585,079	0.094	47.0	0.026	13.0
LAME*	19,704	22,078	7,989,818,554	376,832	7,274,496	1,688,230,256	720,826,607	18.361	9,180.5	0.099	49.5
Mesa Gears*	119,830	462,942	296,287,705	4,210,688	51,445,760	36,839,087	38,449,257	0.480	240.0	0.019	9.5
Mesa Morph3D*	120,401	465,167	239,456,087	4,218,880	51,642,368	28,181,931	42,865,365	0.548	274.0	0.023	11.5
Mesa Reflect*	119,883	467,373	2,752,665,912	4,251,648	61,407,232	431,196,702	221,523,544	3.524	1,762.0	0.023	11.5
MPEG-2 Enc. DVD	7,605	78,549	17,986,999,069	475,136	49,283,072	3,257,725,765	554,222,287	17.798	8,899.0	0.309	154.5
MPEG-2 Enc. 720P	7,605	78,549	47,606,551,352	475,136	127,008,768	8,581,717,942	1,563,082,541	48.013	24,006.5	0.550	275.0
MPEG-2 Enc. 1080I	7,605	78,549	111,041,463,652	475,136	284,622,848	20,148,301,625	3,349,482,784	113.605	56,802.5	0.261	130.5
MPEG-2 Dec. DVD	9,832	248,779	1,307,000,398	1,540,096	16,908,288	219,595,775	76,688,056	1.894	947.0	0.032	16.0
MPEG-2 Dec. 720P	9,832	248,779	3,992,213,571	1,540,096	42,467,328	673,343,544	243,881,680	5.749	2,874.5	0.087	43.5
MPEG-2 Dec. 1080I	9,832	248,779	8,038,214,930	1,540,096	93,847,552	1,341,912,185	464,649,094	11.988	5,994.0	0.133	66.5
mpg123*	7,790	87,313	574,034,774	729,088	3,407,872	166,675,525	45,334,678	0.733	366.5	0.003	1.5
POVray3	151,346	227,214	6,017,197,975	2,179,072	16,384,000	1,562,189,592	683,690,648	11.105	5,552.5	0.146	73.0
Rasta*	24,589	114,943	25,120,492	876,544	5,767,168	5,925,648	1,989,604	0.037	18.5	0.002	1.0
Rsynth	7,089	48,121	402,500,964	625,264	3,997,696	102,351,142	39,223,906	0.852	426.0	0.025	12.5
Timidity	40,514	81,256	4,588,632,916	892,928	26,279,936	1,340,471,112	594,047,710	2.046	1,023.0	0.160	80.0
Total	**1,099,500**	**4,105,520**	**217,315,777,907**	**35,932,784**	**993,329,152**	**40,532,832,944**	**8,943,732,836**	**242.375**	**121,182.5**	**3.169**	**1,584.5**
Arithmetic Mean	**43,980**	**164,221**	**8,692,631,116**	**1,437,311**	**39,733,166**	**1,621,313,318**	**357,749,313**	**9.695**	**4,847.3**	**0.127**	**63.4**

Table 5: **Berkeley Multimedia Workload Characteristics** - times in seconds (sec) and millions of CPU cycles (cycles). Compaq DS20 (dual 500 MHz Alpha 21264, Tru64 Unix v5.0 Rev. 910), *user time* (time spent processing in user space), and *system time* (time spent processing in system space on behalf of an application) Compaq DS20 (dual 500 MHz Alpha 21264, Tru64 Unix v5.0 Rev. 910). All applications compiled with GCC v2.8.1 except (*) compiled with DEC C v5.6-075.

SPEC95 clearly has a greater emphasis on floating point operations. Both multimedia workloads emphasize shift/logical operations to a greater degree than the SPEC benchmarks. Individual applications exhibit even more distinctive differences (see Figure 2 - full instruction mix counts are listed in Appendix A).

4.2 Data Width

The motivation behind SIMD multimedia extensions for general purpose microprocessors is to bridge the mismatch between wide data paths (32 or 64 bits) and narrow multimedia data types. In order to examine the effectiveness of the SIMD approach for multimedia relative to a more traditional workload, we measured the *data widths* of instruction operands and results for all of the instructions dynamically executed in the SPEC95 (integer and floating point application suites) and Berkeley multimedia (audio, speech, document, video and 3D domains) workloads. Data width was determined by counting the number of *leading zeros* in the absolute value of an operand or result value and then subtracting that count from the *operation width*. The correct calculation and interpretation of both of these quantities was inferred from the specific instruction involved. No attempt was made to distinguish operations on pointers from operations on program data. Only arithmetic instructions were measured (bitwise logical operations were excluded).

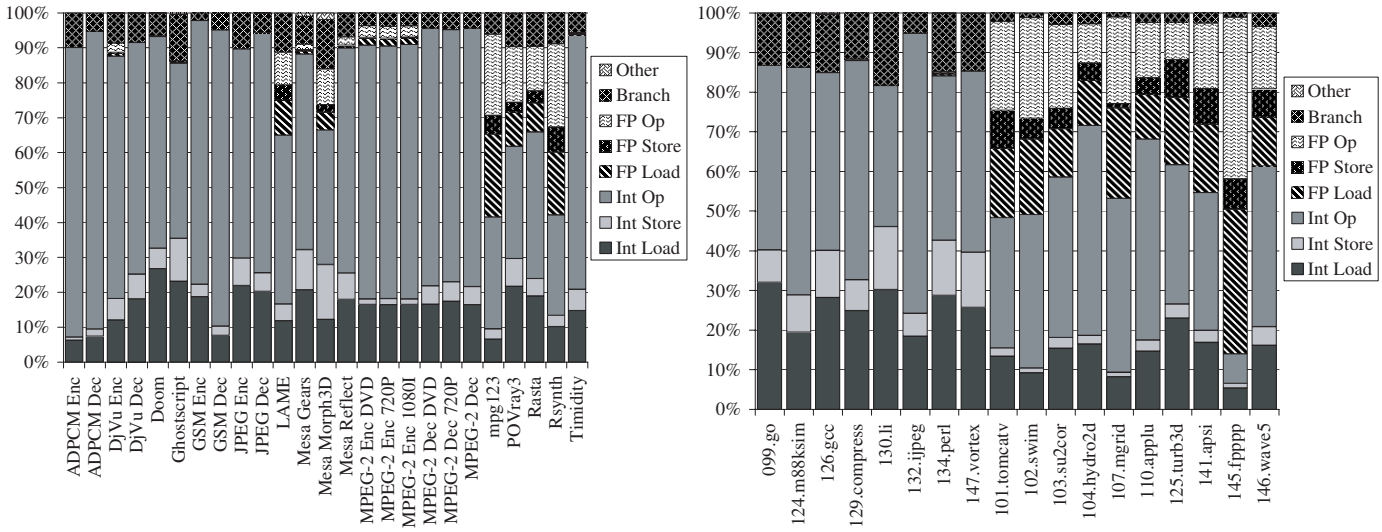
Figure 3 graphs the cumulative distribution of the data width metric for all of the dynamic instructions executed in each workload. The results were categorized according to the type of application. In the case of SPEC95, these were the integer and floating point application suites, while the Berkeley multimedia workload was divided into audio, speech, document, video and 3D graphics application domains. This data should be taken as a rough approximation due to the way in which the number of required bits was computed (discussed above). Even so, the resulting curves are quite telling. Compared to the SPEC95 workload, the curves for multimedia application domains have much more pronounced "knees", beyond which the curves flatten out until sufficient bits for pointer arithmetic (around 32-bits) are reached. Video applications utilize around 12-bits of precision, speech and audio around 16-bits, while the remaining domains of document and 3D graphics applications do not demonstrate such clear boundaries.

4.3 Potential for Extracting ILP

A *basic block* is a sequence of instructions uninterrupted by a branch or jump instruction, which must therefore execute sequentially. The average basic block size is of interest in determining the amount of instruction level parallelism available, with the average size of a basic block defining the maximum amount of local parallelism which might be extracted through superscalar techniques such as register renaming and out of order execution. (As noted below, additional parallelism may be obtained by speculating on the branch outcome.) Although typically the amount of instruction level parallelism (ILP) actually extracted is 25%-35% of this maximum, the larger the basic block, the greater the gain [Fritt99]. The ATOM `pixie`

Integer											
	Load	**Store**	**CBranch**	**CMove**	**Add/Sub**	**Mult**	**Logic**	**Shift**	**Cmp**	**Call/Ret**	**Branch**
Multimedia	16.51%	2.44%	3.94%	5.16%	37.36%	0.46%	7.03%	15.91%	3.69%	0.39%	0.41%
SPEC95	17.41%	4.72%	4.18%	0.21%	32.14%	1.01%	3.87%	2.18%	1.19%	0.51%	0.79%
Mediabench+ [Fritt99]	15.54%	6.39%	15.44%	5.85%	25.62%	1.48%	5.85%	10.03%	0.74%	2.36%	0.52%
Floating Point											
	Load	**Store**	**CBranch**	**CMove**	**Add/Sub**	**Mult**	**Div**	**Sqrt**	**Cmp**	**Convert**	**Other**
Multimedia	2.36%	0.51%	0.11%	0.00%	1.51%	1.35%	0.01%	0.00%	0.14%	0.63%	0.07%
SPEC95	13.51%	4.66%	0.13%	0.02%	7.13%	5.70%	0.19%	0.00%	0.23%	0.12%	0.09%
Mediabench+ [Fritt99]	2.75%	1.03%	0.29%	0.29%	0.98%	0.98%	0.10%	0.00%	0.00%	0.64%	0.98%

Table 6: **Overall Dynamic Instruction Mix -** Multimedia and SPEC95 data are from the DEC Alpha architecture, while Mediabench+ [Fritt99] is from the IMPACT compiler.



(a) Berkeley Multimedia Workload

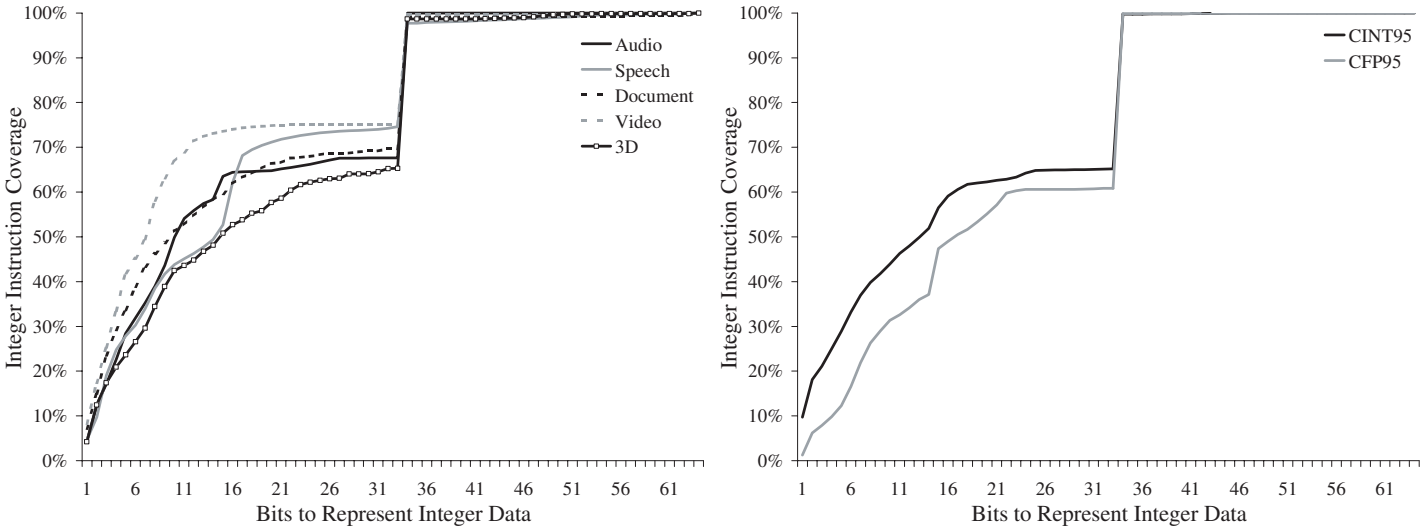
(b) SPEC95

Figure 2: **Detailed Instruction Mix Comparison**

tool reports the average executed basic block size, which for the Berkeley multimedia workload was measured to be 14 DEC Alpha instructions, while for SPEC CINT95 it was 7 instructions and for SPEC CFP95 it was 80 instructions.

In order to extract instruction level parallelism beyond the basic block it is necessary to employ speculative execution, whereby an operation residing on the expected path of control flow (beyond a branch) executes as soon as its source operands become available, rather than waiting for the branch operation to complete. Many modern microprocessors have very long pipelines in order to achieve high clock speeds (for example, the AMD Athlon has a 14-stage pipeline). On such machines, a mispredicted branch can stall the processor for a considerable number of cycles while the pipeline is drained. It is for these reasons that branch prediction is extremely critical to performance. Table 7 compares the average dynamic branch characteristics of the multimedia and SPEC95 workloads to the results presented in [Lee84] and [Perl93] for other workloads.

For most branches, there are long sequences of either taken or not taken decisions; it is less common to see alternation [Lee84]. A sequence of identical branch behavior is termed a *run*, so the sequence "NNTTTTNNTTT" consists of the run lengths 2, 4, 2, 3 where the token "T" indicates a taken branch and "N" represents a not-taken branch. Figure 4 shows the distribution of run lengths for conditional branches in both the SPEC95 and Berkeley multimedia workloads. We can see that the multimedia workload exhibits somewhat more predictable branch behavior than SPEC95 in that the proportion of shorter run lengths is greater for SPEC95. In addition, we see spikes at the significant lengths of 7 and 15, which correspond well to the loop lengths in video and imaging algorithms (e.g. 8x8 MPEG subblock, and 16x16 MPEG macroblock). This would seem to be in disagreement with [Bish99] and their study of the original UCLA MediaBench workload, for which branch prediction accuracy was found to be poor due to unpredictable data dependent branching. Figure 5 graphs the probability of a given run length for the multimedia and SPEC95 workloads as well as the results obtained in [Lee84] for their own workload.



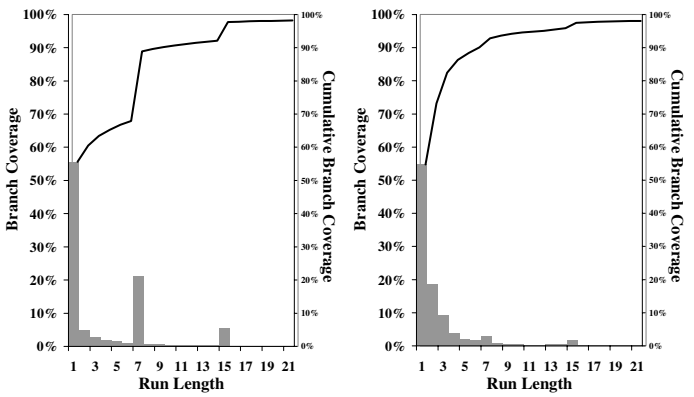
(a) Berkeley Multimedia Workload

(b) SPEC95

Figure 3: **Bits to Represent Integer Data** - number of bits computed by subtracting the number of leading zeros in the absolute value of an operand or result from the register width (64-bits) based on dynamic instruction counts

Workload (Arch.)	*dynamic*	*cond*	*cond taken*	*taken*
Multimedia (Alpha)	0.071	0.879	0.727	0.762
SPEC95 (Alpha)	0.070	0.894	0.697	0.744
Comp (MIPS) [Perl93]	0.178	-	0.645	0.728
Text (MIPS) [Perl93]	0.201	-	0.634	0.678
FP (MIPS) [Perl93]	0.102	-	0.595	0.699
Sparc (Sparc) [Perl93]	0.219	-	0.442	0.593
VAX (VAX) [Perl93]	0.272	-	0.611	0.708
68k (68k) [Perl93]	0.231	-	0.566	0.695
CPL (IBM) [Lee84]	0.317	-	-	0.640
Bus (IBM) [Lee84]	0.189	-	-	0.657
Sci (IBM) [Lee84]	0.105	-	-	0.704
Sup (IBM) [Lee84]	0.376	-	-	0.540
PDP11 (DEC) [Lee84]	0.388	-	-	0.738
CDC (6400) [Lee84]	0.079	-	-	0.778

Table 7: **Dynamic Branch Characteristics** - ***dynamic*** reports the fraction of total dynamic instructions which were branch instructions, ***cond*** the fraction of branch instructions which were conditional branches, ***cond taken*** the fraction of dynamic branches which were taken, and ***taken*** the fraction of all dynamic branches (both conditional and unconditional) which were taken. Measurements not reported by the original study are denoted with a "-".



(a) Berkeley Multimedia

(b) SPEC95

Figure 4: **Branch Run Lengths** - cumulative branch coverage indicates the fraction of dynamic branches

4.4 Locality

The locality of memory references is an important program property because it has a fundamental impact on memory hierarchy (e.g. cache) design and system performance [John99]. In order to characterize the memory locality of the Berkeley multimedia workload, we briefly examine how miss ratio varies with cache capacity. Figure 6 shows how the miss ratio of a unified cache is affected by cache capacity for the Berkeley multimedia workload in comparison to several other standard workloads (this figure has been extracted from our detailed study of the cache behavior of the Berkeley multimedia workload in [Sling00b]). In contrast to other published speculations on the nature of multimedia application cache behavior, our multimedia workload actually exhibits lower miss ratios than other widely studied workloads. A comparison of average miss ratios can be found in the Appendix.

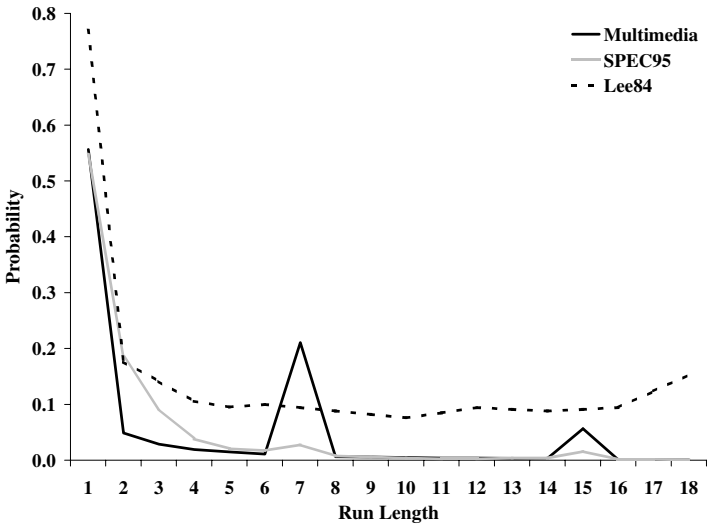


Figure 5: **Branch Run Length Probability** - distribution of the number of times that a conditional branch has the same result for the Berkeley multimedia workload, SPEC95 and [Lee84].

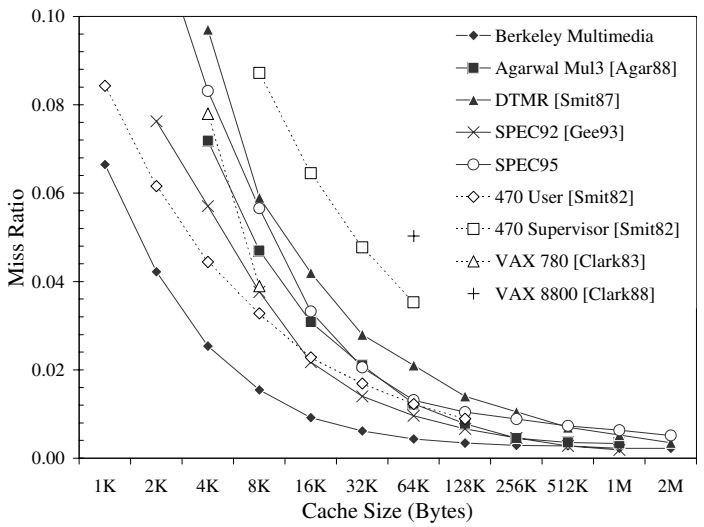


Figure 6: **Cache Comparison** - for unified cache, 32-byte line size

4.5 Multimedia Kernels

It is usually neither time nor resource efficient to hand optimize every line of code in an application. Instead, optimization should focus on execution hot spots or computational kernels, thereby limiting the recoding effort to those portions of the code which have the most potential to improve overall performance. Every application is able to benefit from this strategy by differing degrees. For most applications, there are typically relatively few source lines responsible for a large portion of execution time. In order to see how this varies between applications, we measured the amount of CPU time spent in each line of source code for the Berkeley multimedia workload. The 100 most expensive lines of each application were then ranked from highest to lowest in contribution, and graphed in Figure 7 from bottom to to. Each source line is represented as a rectangle of a height proportional to its cost in CPU time - all of the rectangles for a particular application are stacked vertically. From this graph it is possible to see how quickly a large percentage of CPU time is covered - if relatively few lines need to be accelerated, the application is a good candidate for optimization. Many small contributions indicate that the load is distributed more evenly over a large section of the source code.

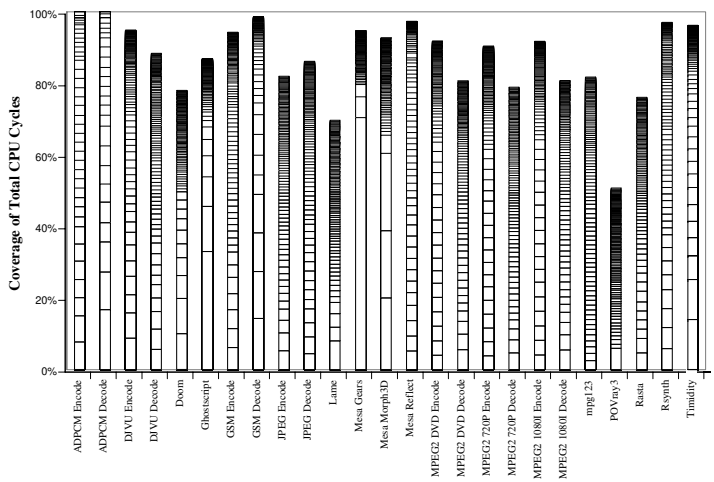


Figure 7: **Source Line CPU Time Coverage Comparison** - each column is the stacked contribution to total execution time of the 100 most expensive (in CPU time) source lines.

Kernels were established based on dynamic line references, CPU cycle coverage, and dynamic instruction coverage. Only cycles are listed in Tables 8, 9, 10, 11 and 12. Appendix C discusses the nature of the underlying algorithms for each kernel in detail.

In order to optimize a computational kernel so that it is as fast as possible, but still correct, it is first necessary to understand the basis of the algorithm behind it. In Appendix C we give an overview of the algorithms which dominated processing time in the Berkeley multimedia workload suite. The C source code for each kernel, as originally extracted from the Berkeley multimedia workload applications is listed. Note that these kernels do not necessarily correspond to a single procedure within the source application. Instead, we have listed semantically different tasks as the kernels rather than source procedures. This is due to the fact that the programmers of each application divided the algorithmic tasks into an arbitrary number of procedures or functions. More experimental applications, not concerned with speed, tend to be split into very small granularity tasks for ease of understanding and debugging. Applications that are more highly optimized tend to use very large procedures to reduce the amount of overhead.

Application	Kernel Name	Src Lines	Static Inst.	% Static Inst	% Static Inst ($\sum$)	CPU Cycles	% CPU Cycles	% CPU Cycles ($\sum$)
ADPCM Enc.	ADPCM Coder	47	108	22.6%	22.6%	68,267,663	99.9%	99.9%
ADPCM Dec.	ADPCM Decoder	31	88	18.6%	18.6%	58,309,984	99.9%	99.9%
LAME	Calc. Quant. Noise	73	598	2.7%	2.7%	1,399,988,405	15.3%	15.3%
	Quantize	55	312	1.4%	4.1%	1,404,207,237	15.3%	30.6%
	FFT	208	981	4.4%	8.5%	1,328,445,691	14.5%	45.1%
	Max Value	8	39	0.2%	8.7%	1,100,539,652	12.0%	57.1%
	Count Encoding Bits	27	76	0.3%	9.0%	670,803,428	7.3%	64.4%
	Psychoacoustic Model	301	1922	8.7%	17.7%	549,265,950	6.0%	70.4%
mpg123	Synthesis Filtering	67	348	0.4%	0.4%	224,787,420	39.6%	39.6%
	DCT64	105	371	0.4%	0.8%	128,055,600	22.6%	62.2%
	Dequantize	517	2040	2.3%	3.1%	91,188,589	16.1%	78.3%
	Parse Bitstream	150	633	0.7%	3.8%	52,379,258	9.2%	87.5%
	DCT36	41	440	0.5%	4.3%	39,868,400	7.0%	94.5%
Timidity	Resample	174	776	1.0%	1.0%	1,067,436,081	58.0%	58.0%
	Mix	143	829	1.0%	2.0%	62,7646318	35.7%	93.7%
	Convert Sample Format	6	25	0.0%	2.0%	66,395,866	3.6%	97.3%

Table 8: **Audio Kernels** - all DEC Alpha instructions are 32-bits (4 bytes) wide

Application	Kernel Name	Src Lines	Static Inst	% Static Inst	% Static Inst ($\sum$)	CPU Cycles	% CPU Cycles	% CPU Cycles ($\sum$)
GSM Enc.	Calc. LTP Parameter	61	834	1.5%	1.5%	196,327,472	51.4%	51.4%
	Short Term Analysis Filter	15	79	0.1%	1.6%	77,040,936	20.2%	71.6%
	Autocorrelation	42	715	1.3%	2.9%	25,174,153	6.6%	78.2%
	Weighting Filter	16	113	0.2%	3.1%	21,071,400	5.5%	83.7%
	Sample Preprocessing	27	185	0.3%	3.4%	19,909,272	5.2%	88.9%
GSM Dec.	Short Term Synthesis Filter	15	114	0.2%	0.2%	107,067,072	72.7%	72.7%
	Long Term Synthesis Filter	11	134	0.2%	0.4%	15,921,696	10.8%	81.3%
	Sample Postprocessing	7	60	0.1%	0.5%	8,206,884	5.6%	86.9%
Rasta	FFT	133	743	0.6%	0.6%	3,602,895	13.9%	13.9%
	Estimate Noise	196	1348	1.2%	1.8%	2,515,541	9.7%	23.6%
	Critical Band Search	28	167	0.1%	1.9%	1,898,095	8.3%	31.9%
	Fill Frame	48	274	0.2%	2.1%	855,424	3.3%	35.2%
Rsynth	Parwave	61	266	0.6%	0.6%	155,877,707	38.7%	38.7%
	Resonator	4	13	0.0%	0.6%	111,920,640	27.8%	66.5%
	Natural Source	9	26	0.1%	0.7%	39,171,784	9.7%	76.2%

Table 9: **Speech Kernels** - all DEC Alpha instructions are 32-bits (4 bytes) wide

5 Conclusions

The Berkeley multimedia workload is a freely available, open source desktop multimedia workload which can be used as a framework for studies of multimedia applications. Source code and the full results of our analysis are available on the World Wide Web at *http://www.cs.berkeley.edu/~slingn/research/.*

Although it incorporates several improvements from its predecessor, the UCLA MediaBench, in the breadth of applications as well as the size of component data sets, it has aged even in the time that we have used it for our studies of multimedia cache behavior ([Sling00b]) and SIMD instruction set performance ([Sling00d]). Because multimedia workloads are relatively immature, they are constantly evolving. For this reason, it is important that multimedia workload analysis be an ongoing process. As an example, the Mesa 3D applications contained within the Berkeley multimedia workload have become increasingly inadequate to represent the massive textures and intricate 3D models that are common in contemporary video games and other virtual environments. New multimedia standards such as MPEG-4 and JPEG 2000 [Hask98] will also need to be reflected in future benchmarking workloads, once these standards are widely adopted (see [Bove97] and [Chia99] for discussions of where multimedia is headed). Future studies should extend the Berkeley multimedia workload in those directions which actual workloads are evolving.

References

[Arav93] Rangarajan Aravind, Glenn L. Cash, Donald L. Duttweiler,

Application	Kernel Name	Src Lines	Static Inst	% Static Inst	% Static Inst ($\sum$)	CPU Cycles	% CPU Cycles	%CPU Cycles ($\sum$)
DJVU Enc.	Encode Buckets	96	629	0.5%	0.5%	164,440,848	41.0%	41.0%
	Forward Filter	38	259	0.2%	0.7%	88,712,217	22.1%	63.1%
	Init	55	522	0.4%	1.1%	48,890,524	12.2%	75.3%
	Create	30	189	0.1%	1.2%	37,103,223	9.3%	84.6%
	Read Liftblock	10	77	0.1%	1.3%	24,486,720	6.1%	90.7%
DJVU Dec.	Backward Filter	38	259	0.2%	0.2%	88,705,134	29.1%	29.1%
	Decode Buckets	101	573	0.4%	0.6%	57,724,634	18.9%	48.0%
	Image	43	256	0.2%	0.8%	32,621,373	10.7%	58.7%
	YCC→RGB Color Space	12	40	0.0%	0.8%	14,258,640	4.7%	63.4%
	Save PPM	20	190	0.1%	0.9%	10,350,651	3.4%	66.8%
Ghostscript	printf()	?	5076	1.6%	1.6%	622,512,658	57.6%	57.6%
	memmove()	?	1328	0.4%	2.0%	99,358,404	9.2%	66.8%
	PPGM Print Row	35	237	0.1%	2.1%	38,827,800	3.6%	70.4%
JPEG Enc.	Huffman Coding	159	523	0.7%	0.7%	120,477,099	63.5%	63.5%
	Forward DCT	111	426	0.6%	1.3%	35,254,796	18.6%	82.1%
	RGB→YCC Color Space	22	78	0.1%	1.4%	18,557,286	9.8%	91.9%
JPEG Dec.	Huffman Decoding	143	678	0.8%	0.8%	27,062,307	30.8%	30.8%
	Inverse DCT	119	571	0.7%	1.5%	24,570,128	28.0%	58.8%
	YCC→RGB Color Space	25	93	0.1%	1.6%	19,278,930	22.0%	80.8%

Table 10: **Document Kernels** - all DEC Alpha instructions are 32-bits (4 bytes) wide

Application	Kernel Name	Src Lines	Static Inst	% Static Inst	% Static Inst ($\sum$)	CPU Cycles	% CPU Cycles	% CPU Cycles ($\sum$)
MPEG-2 Enc.	Block Match	52	294	0.4%	0.4%	10,256,217,376	59.8%	59.8%
(DVD)	Forward DCT	14	116	0.1%	0.5%	2,107,425,600	12.3%	72.1%
	Read PPM	54	402	0.5%	1.0%	497,865,976	2.9%	75.0%
	Horizontal Sub Sample	35	244	0.3%	1.3%	437,053,920	2.6%	77.6%
	Vertical Sub Sample	76	478	0.6%	1.9%	365,081,184	2.1%	79.7%
	Quantize	15	67	0.1%	2.0%	348,478,200	2.0%	81.7%
	Inverse DCT	35	334	0.4%	2.4%	263,395,304	1.5%	83.2%
MPEG-2 Dec.	Inverse DCT	75	649	0.3%	0.3%	435,642,224	29.7%	29.7%
(DVD)	Add Block	46	191	0.1%	0.4%	200,847,816	13.7%	43.4%
	Form Prediction	57	302	0.1%	0.5%	188,829,262	12.9%	56.3%
	Dither	88	620	0.2%	0.7%	177,692,384	12.1%	68.4%
	Parse Bitstream	30	146	0.1%	0.8%	80,839,586	5.5%	73.9%

Table 11: **Video Kernels** - all DEC Alpha instructions are 32-bits (4 bytes) wide

Hsueh-Ming Hang, Barry G. Haskell, Atul Puri, "Image and Video Coding Standards," *AT&T Technical Journal*, Vol. 72, No. 1, January/February 1995, pp. 67-88

[BDTi97] Garrick Blalock, "The BDTImark: A Measure of DSP Execution Speed," 1997 White Paper, *http://www.bdti.com/articles/wtpaper.htm*, retrieved April 24, 2000

[Bhar98] R. Bhargava, L. K. John, B. L. Evans, R. Radhakrishnan, "Evaluating MMX Technology Using DSP and Multimedia Applications," *Proceedings of the IEEE International Symposium on Microarchitecture*, Dallas, Texas, November 30-December 2, 1998, pp. 37-45

[Bhas97] Vasudev Bhaskaran, Konstantinos Konstantinides, and Balas Natarajan, "Multimedia architectures: from desktop systems to portable appliances," *Proc. Multimedia Hardware Architectures, SPIE Vol. 3021*, San Jose, California, February 2-14, 1997, pp. 14-25

[Bish99] Benjamin Bishop, Thomas P. Keilliher, Mary Jane Irwin, "A Detailed Analysis of MediaBench," *Proceedings of the 1999 IEEE Workshop on Signal Processing Systems (SiPS 99)*, Taipei, Taiwan, 20-22 Oct. 1999, pp. 448-455

[Bove97] V. Michael Bove, Jr., "The Impact of New Multimedia Representations on Hardware and Software Systems," *Proc. SPIE Multimedia Hardware Architectures*, Vol. 3021, 1997

[CEA] Consumer Electronics Association, "CEA DVD Player Sales," *http://www.thedigitalbits.com/articles/ cemadvdsales.html*, retrieved September 9, 2000

[Chia99] Leonardo Chiariglione, "MPEG: Achievements and Future Projects," *Proc. of IEEE Multimedia Systems*, July 7-11, 1999, Florence, Italy, pp. 133-138

[CNN99] Cable News Network, "Wolf Blitzer Interview with Vice President Al Gore on CNN's Late Edition," *http://www.cnn.com/ALLPOLITICS/stories/ 1999/03/09/president.2000/transscript.gore/*, retrieved April 24, 2000

[Cont91] Thomas M. Conte, Wen-mei W. Hwu, "Benchmark Characterization," *IEEE Computer*, Vol. 24, No. 1, January 1991, pp. 48-56

[Cont97] Thomas M. Conte, Pradeep K. Dubey, Matthew D. Jennings, Ruby B. Lee, Alex Peleg, Salliah Rathnam, Mike Schlansker, Peter Song, Andrew Wolfe, "Challenges to Combining General-Purpose and Multimedia Processors," *IEEE Computer*, Vol. 30, No. 12, December 1997, pp. 33-37

[DEC] Digital Equipment Corporation, "ATOM Reference Manual," *http://www.partner.digital.com/www-swdev/files/DECOSF1/Docs/Other/ATOM/ref.ps*

Application	Kernel Name	Src Lines	Static Inst	% Static Inst	% Static Inst ($\sum$)	CPU Cycles	% CPU Cycles	% CPU Cycles ($\sum$)
Doom	Render Column	15	85	0.0%	0.0%	579997996	37.7%	37.7%
	Render Span	16	88	0.0%	0.0%	289970312	18.9%	56.6%
	Render Segment	80	408	0.1%	0.1%	164834403	10.7%	67.3%
Mesa	Rasterize	6	1248	0.3%	0.3%	238578833	74.5%	74.5%
(Gears)	memset()	?	288	0.1%	0.4%	61945608	19.3%	93.8%
	Transform/Normalize	51	354	0.1%	0.5%	2159640	0.7%	94.5%
	Project/Clip Test	95	447	0.1%	0.6%	2524089	0.8%	95.3%
	Lighting	48	284	0.1%	0.7%	1861654	0.6%	95.9%
POVray3	Synthesize Texture	101	903	0.4%	0.4%	1252061796	19.4%	19.4%
	Bounding Box	100	352	0.2%	0.6%	681955646	10.5%	29.9%
	Vista Buffer	38	204	0.1%	0.7%	493067923	7.6%	37.5%
	Lighting	40	219	0.1%	0.8%	353206790	5.5%	43.0%
	memmove()	?	1328	0.6%	1.4%	941081191	14.5%	57.5%

Table 12: **3D Graphics Kernels** - all DEC Alpha instructions are 32-bits (4 bytes) wide

[Fritt99] Jason Fritts, Wayne Wolf, Bede Liu, "Understanding Multimedia Application Characteristics for Designing Programmable Media Processors," *Proceedings of SPIE Photonics West, Media Processors '99, SPIE Proceedings*, Vol. 3655, San Jose, California, January 28-29, 1999, pp. 2-13

[Hask98] Barry G. Haskell, Paul G. Howard, Yann A. LeCun, Atul Puri, Joern Ostermann, M. Reha Civanlar, Lawrence Rabiner, Leon Bottou, Patrick Haffner, "Image and Video Coding - Emerging Standards and Beyond," *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 8, No. 7, November 1998, pp. 814-837

[Intel] Intel Corporation, "Intel Media Benchmark: Quantifying Multimedia, Communications, Visualization, and 3D Geometry Performance," *http://www.intel.com/procs/perf/icomp/imbwhite/*, retrieved April 24, 2000

[John99] Lizy Kurian John, Purnima Vasudevan, Jyotsna Sabarinathan, "Workload Characterization: Motivation, Goals and Methodology," *Workload Characterization: Methodology and Case Studies*, IEEE Computer Society, edited by L. K. John and A. M. G. Maynard, 1999

[Kodak] "Kodak Digital Image Offering," *http://www.kodak.com/ digitalImages/samples/imageIntro.shtml*, retrieved April 24, 2000

[Kuro98] Ichiro Kuroda, Takao Nishitani, "Multimedia Processors," *Proceedings of the IEEE*, Vol. 86 No. 6, June 1998, pp. 1203-1221

[Lee84] Johnny K. F. Lee, Alan Jay Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, Vol. 17, No. 1, January 1984, pp. 6-22

[Lee95] Ruby B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, Vol. 15, No. 2, April 1995, pp. 22-32

[Lee96] Ruby B. Lee, Michael D. Smith, "Media Processing: A New Design Target," *IEEE Micro*, Vol. No., August 1996, pp. 6-9

[Lee97] Chunho Lee, Miodrag Potkonjak, William H. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proceedings of the 30th Annual Symposium on Microarchitecture*, Research Triangle Park, North Carolina, December 1-3, 1997, pp. 330-335

[Mick95] Robert A. Mickelsen, "Dessert Ammonites POVray3 Dataset," *http://www.povray.org/ people/ram/datasets/ammndata.zip*, *POVzine2*, March/April 1995, retrieved April 24, 2000

[Naka96] Jill Nakashima, Ken Tallman, "The VIS Advantage: Benchmark Results Chart VIS Performance," White Paper, October 1996, *http://www.sun.com/microelectronics/vis/*, retrieved April 24, 2000

[Nguy99] Huy Nguyen, Lizy Kurian John, "Exploiting SIMD Parallelism in DSP and Multimedia Algorithms Using the AltiVec Technology," *Proceedings of the 1999 Interational Conference on Supercomputing*, Rhodes, Greece, June 20-25, 1999, pp. 11-20

[Perl93] Chris Perleberg, Alan Jay Smith, "Branch Target Buffer Design and Optimization," *IEEE Transactions on Computers*, Vol. 42, 1993, pp. 396-412

[Pope94] Pope Music, "Carmen Ballet for Strings and Percussion - First Intermezzo, by Rodion Konstantinovich Shchedrin (1932-Present), played by the State Symphony Orchestra 'Young Russia", conducted by Mark Gorenstein," *PM2002-2*, Copyright 1994 by Pope Music

[Rice96] Daniel S. Rice, "High-Performance Image Processing Using Special-Purpose CPU Instructions: The UltraSPARC Visual Instruction Set," University of California at Berkeley, Master's Report, March 19, 1996

[Robe00] Bill Roberts, "Digital Cameras," *Electronic Business*, March 2000, *http://www.eb-mag.com/eb-mag/Issues/2000/200003/index.asp*, retrieved September 9, 2000

[Rose92] M. Rosenblum, J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, Vol. 10, No. 1, February, 1992, pp. 26-52

[Rowe93] L.A. Rowe, K. Patel and B.C. Smith, "Performance of a Software MPEG Video Decoder," *Proceedings of the 1st ACM International Conference on Multimedia*, Anaheim, California, August 2-6, 1993, pp. 75-82

[Sling00b] Nathan T. Slingerland, Alan Jay Smith, "Cache Performance for Multimedia Applications," *University of California at Berkeley Technical Report CSD-00-1123*, December 2000

[Sling00c] Nathan T. Slingerland, Alan Jay Smith, "Multimedia Instruction Sets for General Purpose Microprocessors: A Survey," *University of California at Berkeley Technical Report CSD-00-1124*, December 2000

[Sling00d] Nathan T. Slingerland, Alan Jay Smith, "Measuring the Performance of Multimedia Instruction Sets," *University of California at Berkeley Technical Report CSD-00-1125*, December 2000

[Snow96] Mark Snow, "X-files Theme Song MIDI file," *http://w3.one.net/~kklasmei/DamnGood/X_files4.mid*, August 29, 1996, retrieved April 24, 2000

[Taka00] Dean Takahashi, "Good Vibrations," *Electronic Business*, June 2000, *http://www.eb-mag.com/eb-mag/Issues/2000/200006/index.asp*, retrieved September 9, 2000

[Zivo94] V. Zivojnovic, H. Schraut, M. Willems, R. Schoenen, "DSPs, GPPs, and Multimedia Applications - An Evaluation Using DSPstone," Proceedings of (ICSPAT '95), Boston, Massachusetts, October 1995, *http://www.ert.rwth-aachen.de/Projekte/Tools/PAPERS /cad_Zivojnovic95icspatb.ps.gz*, retrieved April 24, 2000

[Zuck96] Daniel F. Zucker, Michael J. Flynn, and Ruby B. Lee, "A Comparison of Hardware Prefetching Techniques for Multimedia Benchmarks," *Proceedings of the 3rd IEEE International Conference on Multimedia Computing and Systems*, Hiroshima, Japan, June 17-23, 1996, pp. 236-244

1 Appendix A

1.1 Additional Industry Trends

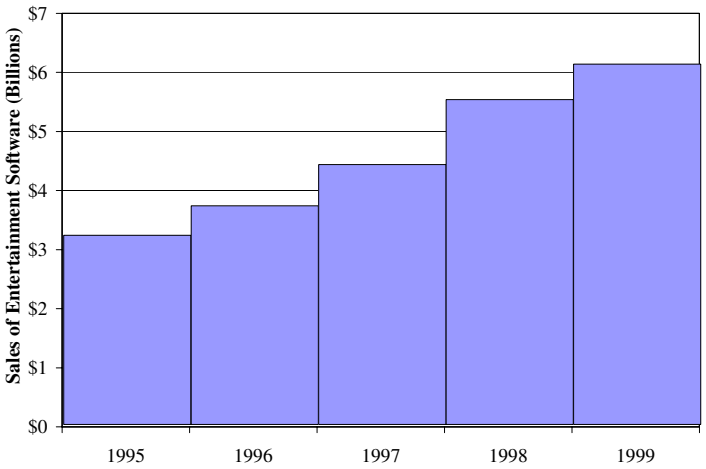


Figure 1: **Video Game Software**

1.2 UCLA MediaBench - Critique

Mesa is a clone of SGI's OpenGL 3D API which is designed for rendering 3D images in real-time on the screen. MediaBench's Mesa applications are somewhat simplistic compared to how the Mesa (or OpenGL) libraries are typically used, as they only generate static images. The Mesa and OpenGL APIs focus is on computationally lightweight rendering intended for interactive, animated output, and so find application in 3D visualization and 3D video games where interactivity is important. Most people interested in rendering static images would instead use a ray tracing application which would be capable of generating higher quality images, of course at much greater computational cost.

The MediaBench MPEG-2 coding application operates on four frames at SIF (352x240) resolution. DVD is by far the most important application of MPEG-2 thus far. However, with the exception of a few low-budget titles, most DVD media is encoded at 720x480. More importantly, because only four frames are encoded in MediaBench, the full behavior of the MPEG-2 encoder and decoder is not characterized. At least a full group of pictures (typically 8-16 frames) is required in order to capture the correct ratio of I, B and P frame types.

EPIC, as noted by its author, is experimental code, with no real effort made to keep the code efficient or fast. In addition it is limited to monochrome images with dimensions that are even powers of two.

JPEG encoding and decoding works with a 227x149 sized image of a rose (a small image with unusual dimensions). Higher resolution images are becoming more common due to digital photography in which cameras typically capture JPEG images at resolutions of at least 640x480 [Rama98].

Incorporating ADPCM and G.721 in the same benchmark suite is redundant, as each implements a variation of adaptive differential pulse code modulation. The CCITT G.721 reference code is coded in floating point, while the IMA ADPCM code uses fixed-point integer arithmetic, although they are considered to have comparable audio quality.

Including both the PGP and Pegwit data encryption applications is also redundant. It would make sense to only include one such application in the suite. PGP would be the logical application to keep as it has far more widespread application than the relatively unknown Pegwit. Even given this, it is not clear how encryption fits the definition of a multimedia application.

1.3 Multimedia Instruction Mix

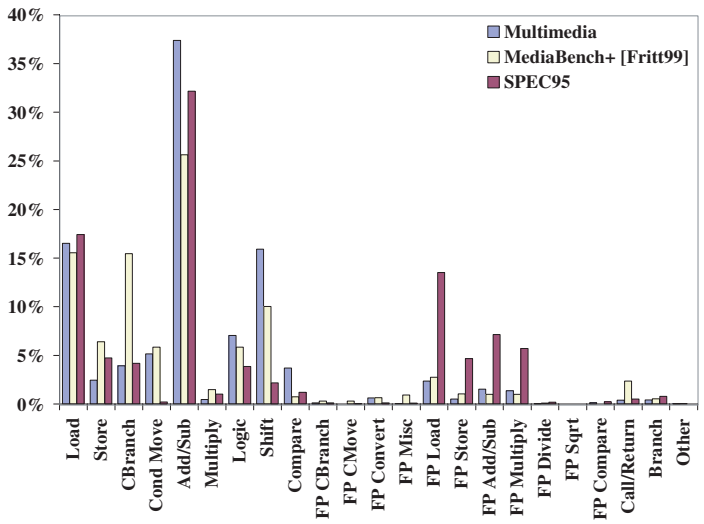


Figure 3: **Instruction Mix Comparison**

The detailed dynamic instruction mix breakdown for each component application of the Berkeley multimedia workload can be found in Tables 1, 2, and 3. The same information for SPEC95 is presented in Tables 4, 5, and 6.

2 Appendix B - Applications

2.1 ADPCM

Adaptive pulse code modulation (ADPCM) takes advantage of the continuous nature of audio signals. Neighboring samples are typically very close in magnitude. Therefore, rather than encoding each audio sample independently as is done in pulse code modulation (PCM) or raw digital audio, the preceding sample is used as a predictor for the current sample, with only the difference being quantized and subsequently encoded. [Pan93] The quantizer adapts to the rate of change of the audio waveform being compressed. A block diagram of ADPCM coding is shown in Figure 11.

Many different implementations of the ADPCM algorithm exist. The Berkeley multimedia workload includes the freely available Interactive Multimedia Association (IMA) codec.

	Total	*ADPCM Encode*	*ADPCMC Decode*	*DJVU Encode*	*DJVU Decode*	*Doom*	*Ghostscript*	*GSM Encode*	*GSM Decode*
Load Byte	6,778,280	0	0	18,370	1,110,224	1,142	5,648,543	1	1
Load Word	0	0	0	0	0	0	0	0	0
Load Long	228,569,938	2,455,961	3,682,713	18,169,163	13,077,300	123,617,409	62,724,290	4,843,102	7,634,766
Load Quad	602,032,274	1,850,037	623,285	30,173,547	40,779,567	285,229,832	176,650,287	66,725,719	3,606,089
Store Byte	8,765,064	0	0	19,266	1,110,228	111,806	7,522,543	1,221	1,218
Store Word	3,784,458	0	0	148	8	71,909	3,711,174	1,219	0
Store Long	84,841,453	1,229	1,227,981	11,817,826	6,488,672	18,447,727	43,409,507	3,448,511	3,421,624
Store Quad	184,517,792	614,622	1,246	12,722,245	13,735,778	71,531,192	75,840,453	10,072,256	462,900
Cond Branch	274,588,131	6,138,696	2,457,213	31,923,427	21,518,650	89,629,454	115,354,014	7,566,677	6,891,957
Cond Move	87,707,764	8,588,491	9,814,016	10,168,642	7,289,807	21,953,713	20,350,475	9,542,620	7,718,788
Add/Sub Long	271,787,383	9,713,090	8,590,946	39,856,714	34,805,838	132,388,734	41,588,802	4,843,259	3,242,829
Add/Sub Quad	727,606,475	7,988,694	7,375,318	60,129,890	45,205,778	302,327,509	206,076,272	98,503,014	29,479,377
Multiply Long	3,483,045	0	0	3,717	398,786	2,010,261	993,570	76,711	2,437
Multiply Quad	26,112,640	0	0	66	35	1,923,442	3,045,834	21,143,263	3,376,297
Logic	580,467,353	13,760,955	12,287,159	82,855,123	55,911,711	186,830,035	159,859,898	68,962,472	30,378,018
Shift	503,710,372	7,986,162	7,983,708	61,741,563	42,277,854	240,749,591	68,968,089	74,003,405	39,100,492
Compare	128,484,307	8,588,491	3,680,256	23,205,150	14,652,977	36,263,419	31,225,252	10,868,762	11,545,826
FP Cond Branch	369,635	0	0	257	48	277,025	92,303	2	2
FP Cond Move	249	0	0	0	0	0	249	0	0
FP Convert	4,793,117	0	0	4,277,677	48	415,533	99,857	2	2
FP Misc	9,878	0	0	1	0	0	9,877	0	0
FP Load Single	632,395	0	0	187	1	415,524	216,683	0	0
FP Store Single	45,776	0	0	3	1	0	45,772	0	0
FP Add/Sub Single	2,145,208	0	0	2,138,796	0	0	6,412	0	0
FP Multiply Single	3,220,312	0	0	3,208,194	0	0	12,118	0	0
FP Divide Single	1,077	0	0	0	0	0	1,077	0	0
FP Sqrt Single	0	0	0	0	0	0	0	0	0
FP Compare Single	0	0	0	0	0	0	0	0	0
FP Load Double	3,742,578	0	0	3,208,326	67	415,537	118,645	3	3
FP Store Double	7,257,405	0	0	1,069,410	16	278,463	5,909,516	0	0
FP Add/Sub Double	1,137,857	0	0	1,069,398	0	0	68,459	0	0
FP Multiply Double	258,818	0	0	40	19	138,512	120,246	1	1
FP Divide Double	147,726	0	0	0	0	138,508	9,218	0	0
FP Sqrt Double	0	0	0	0	0	0	0	0	0
FP Compare Double	341,241	0	0	45	29	277,021	64,145	1	1
Call / Return	35,362,106	3,694	3,694	694,314	2,767,115	11,163,007	20,530,276	200,006	130,732
Branch	43,923,076	615,831	615,831	2,365,197	3,572,166	10,672,945	25,201,200	879,906	248,610
Other	6,179,370	0	0	197,754	492,037	592,001	4,893,800	3,778	2,557
TOTAL	**3,832,804,553**	**68,305,953**	**58,343,366**	**401,034,456**	**305,194,760**	**1,537,871,251**	**1,080,368,856**	**381,685,911**	**147,244,527**

Table 1: **Berkeley Multimedia Workload Instruction Mix** - Part 1 of 3

	JPEG Encode	*JPEG Decode*	*LAME*	*Mesa Gears*	*Mesa Morph3D*	*Mesa Reflect*	*MPEG-2 Encode DVD*	*MPEG-2 Encode 720P*	*MPEG-2 Encode 1080I*
Load Byte	0	15	0	159	159	162	122,165	306,034	664,346
Load Word	0	0	0	0	0	0	0	0	0
Load Long	15,456,331	6,248,862	459,878,385	21,420,844	8,880,796	139,251,544	194,914,155	544,343,003	1,185,685,540
Load Quad	25,884,799	11,496,870	596,936,807	44,726,472	20,256,382	356,383,122	2,625,563,425	6,898,662,249	16,305,754,765
Store Byte	1,634	772	0	213	31	469,362	123,360	307,343	665,390
Store Word	860	742	0	20	20	191	8,657	24,873	66,680
Store Long	6,319,031	1,118,200	178,110,997	18,195,886	27,752,573	69,577,276	76,057,310	218,381,584	454,170,322
Store Quad	8,474,206	3,555,461	246,104,361	18,472,289	9,686,634	140,438,735	192,284,043	568,493,171	1,155,058,459
Cond Branch	15,106,646	4,680,797	729,201,213	24,205,977	30,349,047	175,610,619	591,653,260	1,619,341,331	3,633,131,334
Cond Move	3,675,691	111,929	102,951,015	383,487	1,417,693	6,932,320	1,057,999,331	2,729,061,499	6,591,019,353
Add/Sub Long	15,260,411	6,704,149	923,467,907	26,052,645	31,957,466	214,777,548	3,372,050,729	8,761,537,176	20,987,857,118
Add/Sub Quad	47,610,800	24,481,292	1,919,286,567	28,672,641	41,197,029	426,805,595	3,452,872,410	9,122,615,725	21,383,188,351
Multiply Long	72,511	317,228	34,649,358	173,063	549,197	67,448,117	69,067,294	187,416,894	419,775,140
Multiply Quad	87,041	59	5,433	2,258	2,186	2,047	3,225,646	7,898,780	23,808,528
Logic	17,141,706	12,291,636	409,973,330	53,315,410	8,181,138	311,058,130	1,014,579,305	2,770,825,984	6,196,522,815
Shift	20,495,821	13,887,427	195,885,030	50,690,180	6,870,900	660,969,527	2,845,393,313	7,464,996,665	17,696,147,219
Compare	8,592,463	2,287,275	716,176,409	19,475,327	1,575,842	91,377,573	601,297,661	1,628,584,283	3,677,923,284
FP Cond Branch	10	10	146,364,112	523,637	2,337,432	1,017,957	599	574	582
FP Cond Move	0	0	272,767	0	0	0	64,800	172,800	391,680
FP Convert	10	10	35,295,991	570,896	3,632,567	19,501,744	124,654,521	332,410,521	751,250,841
FP Misc	0	0	3,200,104	47,418	234,562	136,824	8,338,432	22,234,407	50,396,975
FP Load Single	3	1	221,027,815	2,529,198	10,694,972	6,134,444	25,109,096	66,944,891	151,730,177
FP Store Single	0	0	219,890,932	1,199,333	4,765,064	14,836,491	55	55	55
FP Add/Sub Single	0	0	117,891,963	1,165,539	6,219,499	20,595,157	0	0	0
FP Multiply Single	0	0	55,158,946	1,536,887	9,101,388	15,504,440	0	0	0
FP Divide Single	0	0	107	82,767	300,343	6,298,739	0	0	0
FP Sqrt Single	0	0	0	0	0	0	0	0	0
FP Compare Single	0	0	0	0	0	0	0	0	0
FP Load Double	17	15	666,137,380	316,513	1,410,613	796,176	304,651,610	812,395,578	1,839,703,114
FP Store Double	3	7	172,085,563	77,989	246,179	276,695	41,824,510	111,520,480	252,033,775
FP Add/Sub Double	0	0	211,622,317	420,617	748,692	833,540	238,100,487	634,928,421	1,435,971,645
FP Multiply Double	5	5	283,179,814	657,442	1,209,000	1,291,749	193,841,936	516,905,904	1,168,945,600
FP Divide Double	0	0	6,412,767	39,844	74,780	63,310	43,791	115,791	261,711
FP Sqrt Double	0	0	0	0	0	0	0	0	0
FP Compare Double	5	5	123,259,938	628,315	2,893,093	1,334,390	8,381,249	22,349,224	50,657,712
Call / Return	1,204,598	118,436	228,692,817	240,020	781,907	2,894,763	37,412,334	105,899,033	225,832,078
Branch	4,252,300	340,679	119,897,378	614,638	1,147,484	9,952,889	44,581,752	130,471,099	267,119,979
Other	18,291	31,437	0	3,145,932	4,460,302	3,210,474	71,726	183,629	409,699
TOTAL	**189,655,193**	**87,673,319**	**9,123,017,523**	**319,583,856**	**238,934,970**	**2,765,781,650**	**17,124,288,962**	**45,279,329,001**	**105,906,144,267**

Table 2: **Berkeley Multimedia Workload Instruction Mix** - Part 2 of 3

	MPEG-2 Decode DVD	*MPEG-2 Decode 720P*	*MPEG-2 Decode 1080I*	*mpg123*	*POVray3*	*Rasta*	*Rsynth*	*Timidity*
Load Byte	35	41	41	19	20,097	40,755	97	0
Load Word	0	0	0	0	0	0	0	0
Load Long	63,079,368	202,288,004	374,745,728	13,125,436	208,667,745	1,238,950	1,854,468	80,330,481
Load Quad	179,230,670	571,901,170	1,094,954,975	24,035,008	1,168,422,770	3,574,014	36,597,256	190,893,124
Store Byte	15	19	19	677	414	8,386	7,110	0
Store Word	11	7	7	1,626	159	20,853	11	0
Store Long	40,077,380	122,519,230	237,738,683	10,330,487	49,671,406	449,601	1,585,238	66,351,065
Store Quad	36,027,525	124,744,760	222,620,103	6,269,897	457,320,333	794,400	10,596,980	46,014,900
Cond Branch	55,647,468	172,647,314	341,081,043	24,583,711	414,865,553	2,128,693	18,485,990	99,413,559
Cond Move	19,019,746	54,255,467	114,571,768	1,937,309	8,483,484	274,493	672,444	7,499,293
Add/Sub Long	167,347,931	537,902,984	1,021,455,786	23,614,547	73,121,209	1,827,995	3,497,338	125,613,355
Add/Sub Quad	269,748,022	810,825,802	1,671,789,426	80,113,829	993,210,785	4,914,902	68,975,084	475,088,661
Multiply Long	1,728,774	8,269,976	9,036,080	2,140	14,186	28,847	79,398	90,551,054
Multiply Quad	61,801	163,533	368,151	272	9,349	12,930	11,584	464
Logic	257,107,844	726,921,975	1,558,167,551	41,523,867	600,640,884	1,664,309	6,558,450	115,248,825
Shift	304,799,082	896,164,794	1,891,265,922	30,665,422	217,473,019	822,593	11,030,248	427,123,484
Compare	54,915,683	167,099,953	335,191,806	2,021,424	149,077,006	1,223,935	17,792,131	96,668,577
FP Cond Branch	22	25	25	4,907,522	73,396,753	36,086	323,668	5,667
FP Cond Move	0	0	0	0	144,920	0	0	0
FP Convert	24	27	27	2,505,992	38,627,179	156,160	5,743,677	2,726,238
FP Misc	0	0	0	237,267	18,090,629	55,508	9,090,274	95,850
FP Load Single	1	1	1	174,526	113,434,481	1,482,896	59,164,116	28,840
FP Store Single	0	0	0	0	16,842,015	758,711	24,531,599	28,840
FP Add/Sub Single	0	0	0	0	3,488,004	791,053	29,327,417	0
FP Multiply Single	0	0	0	0	2,501,900	603,908	33,642,190	0
FP Divide Single	0	0	0	1	18	12,893	76,887	0
FP Sqrt Single	0	0	0	0	0	0	0	0
FP Compare Single	0	0	0	0	0	0	0	0
FP Load Double	34	38	38	133,283,472	505,378,130	645,861	8,769,336	4,545,582
FP Store Double	126	127	127	30,350,707	163,485,923	135,300	2,503,074	1,469,625
FP Add/Sub Double	0	0	0	66,730,827	387,030,229	758,369	5,082,577	414,510
FP Multiply Double	11	12	12	56,501,913	471,006,459	842,775	6,860,928	3,244,682
FP Divide Double	0	0	0	2,027	16,238,995	9,459	8,974	293,974
FP Sqrt Double	0	0	0	0	0	0	0	0
FP Compare Double	13	15	15	4,907,970	73,169,225	36,685	323,651	5,667
Call / Return	6,972,866	28,356,792	40,089,593	3,526,246	82,756,960	203,227	14,098,018	2,034,885
Branch	9,265,477	39,502,927	52,499,179	4,866,779	110,117,016	273,435	14,421,800	3,852,961
Other	1,543	2,841	3,532	8,613	15,581,145	12,676	7,308	0
TOTAL	**1,465,031,472**	**4,463,567,834**	**8,965,579,638**	**566,229,533**	**6,432,288,380**	**25,840,658**	**391,719,321**	**1,839,544,163**

Table 3: **Berkeley Multimedia Workload Instruction Mix** - Part 3 of 3

	Total	*099.go*	*124.m88ksim*	*126.gcc*	*129.compress*	*130.li*
Load Byte	203,609	4,368	2,828	191,876	158	4,379
Load Word	1,277,742	0	0	1,277,742	0	0
Load Long	20,286,152,750	6,311,668,533	10,606,481,834	93,448,183	2,227,539,341	1,047,014,859
Load Quad	39,377,367,969	3,910,491,909	5,352,162,900	352,592,763	12,119,747,714	17,642,372,683
Store Byte	140,775	4,875	2,626	128,066	454	4,754
Store Word	76,928	1,284	1,175	71,372	788	2,309
Store Long	8,802,366,731	1,834,586,505	5,419,868,490	31,381,127	797,025,009	719,505,600
Store Quad	16,032,718,297	766,064,611	2,335,438,878	157,229,445	3,649,016,338	9,124,969,025
Cond Branch	27,176,517,027	3,671,161,958	9,775,921,461	201,257,598	4,821,119,923	8,707,056,087
Cond Move	1,659,393,881	210,976,189	629,773,595	11,262,020	796,889,938	10,492,139
Add/Sub Long	14,182,186,682	1,570,606,154	8,184,968,577	57,356,939	3,139,682,863	1,229,572,149
Add/Sub Quad	42,891,794,268	8,910,951,772	13,811,128,016	243,056,322	10,743,903,825	9,182,754,333
Multiply Long	36,269,264	11,378,150	24,282,243	598,978	127	9,766
Multiply Quad	10,701,568	216,590	9,982,386	488,715	420	13,457
Logic	31,091,936,453	2,639,514,258	12,955,858,568	235,079,954	7,025,496,706	8,235,986,967
Shift	14,845,120,527	5,397,662	4,785,358,102	95,967,619	7,961,280,532	1,997,116,612
Compare	11,934,177,897	1,488,018,106	6,825,931,243	67,197,600	2,181,244,044	1,371,786,904
FP Cond Branch	161,423,661	4	44	191	161,423,393	29
FP Cond Move	0	0	0	0	0	0
FP Convert	14,092,818	4	111	76,163	14,016,511	29
FP Misc	14,000,070	0	59	12	13,999,999	0
FP Load Single	14,041,880	0	173	25,324	14,016,383	0
FP Store Single	10	0	10	0	0	0
FP Add/Sub Single	3	0	3	0	0	0
FP Multiply Single	29	0	29	0	0	0
FP Divide Single	29	0	29	0	0	0
FP Sqrt Single	0	0	0	0	0	0
FP Compare Single	0	0	0	0	0	0
FP Load Double	134,913,182	24	166	124,747	126,032,572	8,755,673
FP Store Double	29,788,745	7,380	2,407	99,162	33,410	29,646,386
FP Add/Sub Double	16,313	0	57	0	16,256	0
FP Multiply Double	25,462	2	50	25,399	0	11
FP Divide Double	14,041,707	0	0	25,324	14,016,383	0
FP Sqrt Double	0	0	0	0	0	0
FP Compare Double	161,423,561	2	32	116	161,423,393	18
Call / Return	4,655,213,074	362,997,717	1,184,773,511	27,086,286	1,148,733,180	1,931,622,380
Branch	6,265,383,589	546,379,574	1,471,193,979	32,864,807	1,612,155,092	2,602,790,137
Other	5,924,683	4,890	620,231	5,207,521	698	91,343
TOTAL	**239,798,691,184**	**32,240,432,521**	**83,373,753,813**	**1,614,121,371**	**58,728,815,450**	**63,841,568,029**

Table 4: **SPEC95 Instruction Mix** - Part 1 of 3

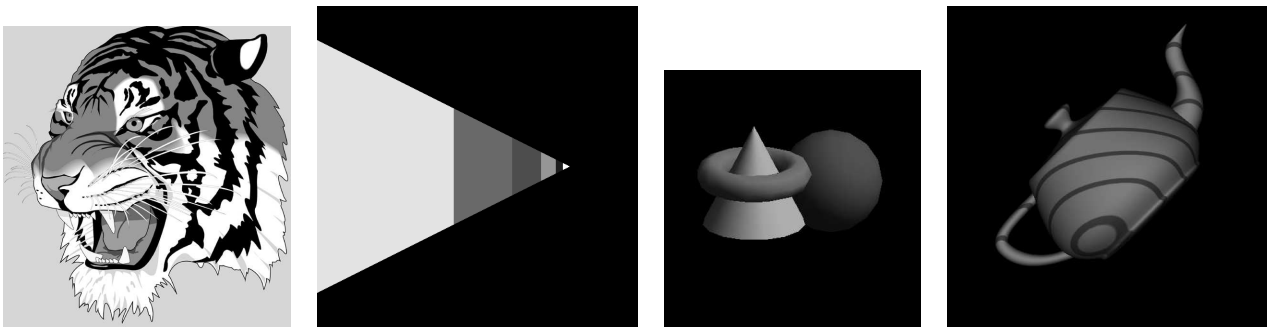
	132.ijpeg	*134.perl*	*147.vortex*	*101.tomcatv*	*102.swim*	*103.su2cor*
Load Byte	6,726	21,543,126	4,297,168	157	35	367
Load Word	0	0	0	0	0	0
Load Long	1,565,354,126	1,523,480,913	9,846,588,005	3,973,533,592	2,569,385,580	5,361,953,769
Load Quad	5,516,119,963	6,866,928,594	11,899,050,852	3,390,856,209	2,354,565,240	2,939,948,803
Store Byte	13,236	22,271,097	4,323,643	8,331	116	1,186
Store Word	9,155	672,623	405,338	921	14	79
Store Long	830,722,525	663,477,405	5,577,098,386	1,010,734,940	642,788,573	1,080,380,301
Store Quad	1,397,814,264	3,396,074,969	6,210,697,290	124,579,571	1,589,856	398,120,773
Cond Branch	1,770,835,295	3,157,116,200	9,321,380,743	1,154,389,215	643,857,110	1,292,906,173
Cond Move	48,269,242	102,025,125	210,133,987	11,356,832	2,106,485	166,804,151
Add/Sub Long	2,554,547,554	699,609,508	4,689,488,512	1,043,875,007	642,065,146	1,245,749,364
Add/Sub Quad	13,077,449,721	5,104,099,020	15,408,800,218	15,708,814,609	19,421,858,511	18,748,637,165
Multiply Long	351,356,557	5,914,301	13,811,033	0	0	3,230
Multiply Quad	11,263,378	3,207,871	15,996,312	6,681	56	4,225,520
Logic	3,580,085,183	3,569,245,181	13,778,986,525	154,114,602	8,638,288	699,549,220
Shift	6,503,660,101	1,456,713,345	2,816,736,335	1,111,793,336	644,505,894	677,164,377
Compare	1,011,698,625	1,182,300,248	1,722,940,998	75,927,103	1,062,711	299,308,720
FP Cond Branch	3,146	23,039,104	146	790	20	61,280,454
FP Cond Move	0	0	0	0	0	0
FP Convert	1,738	19,027,970	146	525,354	2,105,374	346,981,596
FP Misc	768	2	0	0	263,169	145,002,333
FP Load Single	2,560	13,173,538	4,754	587,522,250	10,204,542,186	361,723,911
FP Store Single	0	0	5,180	0	2,773,276,741	148,353
FP Add/Sub Single	0	0	0	0	8,702,401,329	75,516
FP Multiply Single	512	0	0	0	4,671,675,403	14,552
FP Divide Single	0	0	0	0	212,864,982	1,250,372
FP Sqrt Single	0	0	0	0	0	0
FP Compare Single	0	0	0	0	0	0
FP Load Double	4,819	129,071,358	5,010	9,005,902,290	11,046,996	6,203,571,211
FP Store Double	18,244	120,552,685	38,592	5,287,084,381	1,051,681	2,778,566,267
FP Add/Sub Double	768	6,235,284	0	7,048,350,750	8,413,200	4,827,211,367
FP Multiply Double	901	54	52	4,697,880,622	13,671,469	5,906,345,931
FP Divide Double	0	0	0	196,366,064	0	100,617,691
FP Sqrt Double	0	0	0	0	0	0
FP Compare Double	2,245	29,977,299	94	391,683,020	15	120,027,522
Call / Return	67,727,733	715,117,073	1,523,955,054	21,370,628	1,062,698	129,237,936
Branch	206,288,294	1,107,885,168	3,019,716,938	28,243,403	1,059,317	216,419,808
Other	3,576,457	46,533,817	99,895,700	1,066,052	284	1,050,112
TOTAL	**38,496,833,836**	**29,985,292,878**	**86,164,357,011**	**55,025,986,710**	**53,535,858,479**	**54,114,278,130**

Table 5: **SPEC95 Instruction Mix** - Part 2 of 3

	104.hydro2d	*107.mgrid*	*110.applu*	*125.turb3d*	*141.apsi*	*145.fpppp*	*146.wave5*
Load Byte	4,157	375	44	96	5,403	27	122
Load Word	0	0	0	0	0	0	0
Load Long	20,422,601,297	9,909,306,138	13,164,578,033	45,265,475,252	6,687,046,340	3,737,127,098	7,668,511,714
Load Quad	2,664,630,718	1,136,198,272	4,906,543,916	13,352,891,027	3,864,996,064	3,674,746,400	2,495,339,094
Store Byte	18,079	37,418	131	1,487	27,294	968	493
Store Word	1,696	6,228	5	233	342	63	43
Store Long	2,926,027,638	1,445,311,793	3,466,105,004	8,934,734,053	1,720,692,135	1,229,291,090	2,709,811,885
Store Quad	95,931,247	2,859,203	529,542	103,436,885	179,899,739	317,188,548	257,670,399
Cond Branch	2,791,232,435	1,385,067,908	2,946,987,277	5,591,261,753	1,473,449,941	1,126,558,430	1,777,687,458
Cond Move	1,613,621	254,933	3,889	405,054,417	5,516,540	323,989,925	201,255,644
Add/Sub Long	2,964,101,782	1,452,684,866	2,869,729,046	30,357,340,627	1,688,713,690	1,894,295,474	3,291,438,064
Add/Sub Quad	68,596,845,654	50,524,421,726	55,870,557,277	53,742,028,986	18,427,556,740	6,161,852,745	20,241,131,571
Multiply Long	1,159,373	59,285	1,532	2,717,559,457	14,263,062	90,651,609	117,067,830
Multiply Quad	7,992	6,903,320,675	2,681,788,028	242,148,541	1,094,325,054	1,675,548	520,802,970
Logic	275,065,731	66,488,497	993,037,761	936,627,757	348,448,700	826,584,723	644,719,192
Shift	2,537,885,000	2,087,886	20,009	800,784,891	107,528,539	353,167,670	285,031,678
Compare	48,584,632	890,948	12,633	401,982,952	125,243,370	620,528,093	167,387,451
FP Cond Branch	1,042,929,088	13,107,212	310	107	26,137,590	247,906,994	293,764,111
FP Cond Move	308,442,383	0	0	0	8,922,817	0	0
FP Convert	64,956	62	475,781	891,798,671	4,886,790	59,813,625	394,961,170
FP Misc	539,389,323	50	201,633,296	3,092,035	120,564,037	86,667,312	83,396,724
FP Load Single	912,496,696	374,480,753	177,052,679	792,784,256	531,861,074	176,463,236	450,039,819
FP Store Single	2	0	0	79,266,816	4	6,305,141	94,303,282
FP Add/Sub Single	0	0	0	0	4	0	0
FP Multiply Single	0	0	0	0	2	0	0
FP Divide Single	0	0	0	0	960	0	0
FP Sqrt Single	0	0	0	0	0	0	0
FP Compare Single	0	0	0	0	0	0	0
FP Load Double	15,008,138,219	30,271,519,291	13,679,188,626	42,459,228,221	10,269,709,511	50,017,915,008	7,327,169,203
FP Store Double	6,070,114,604	1,430,365,883	5,109,939,722	24,338,685,332	5,644,230,496	10,458,923,396	4,369,526,602
FP Add/Sub Double	5,073,203,796	25,430,438,750	6,229,066,291	12,232,270,251	5,378,072,693	25,974,102,873	4,170,216,429
FP Multiply Double	5,170,580,502	3,896,536,275	10,478,388,222	10,855,230,405	4,031,382,822	29,469,179,864	4,847,783,466
FP Divide Double	954,158,944	50	336,862,508	51,298,633	759,216,186	140,267,492	94,001,945
FP Sqrt Double	0	0	0	0	0	0	0
FP Compare Double	1,879,478,848	13,107,306	337	294,999	87,682,406	310,221,661	346,106,823
Call / Return	41,710,561	376,097	46,390	124,721,520	52,040,437	154,115,390	77,083,084
Branch	42,720,219	577,236	43,354	387,460,478	100,711,227	136,261,484	81,671,749
Other	210,488	76,452	1,279	1,770	48,273	1,206	785
TOTAL	**140,369,349,681**	**134,259,581,568**	**123,112,592,922**	**255,067,461,908**	**62,753,180,282**	**137,595,803,093**	**63,007,880,800**

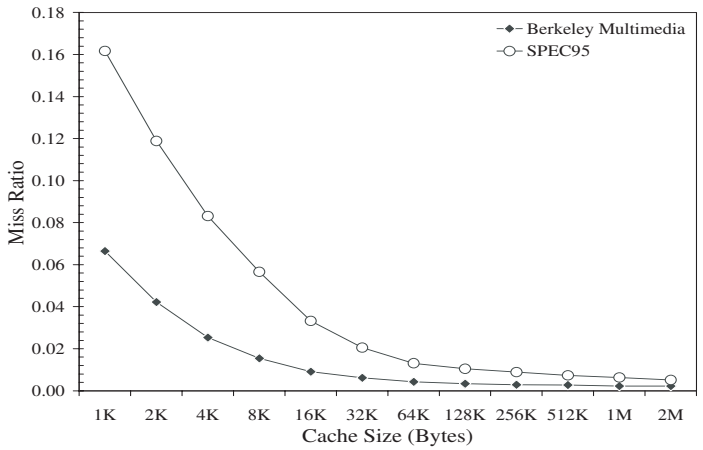
Table 6: **SPEC95 Instruction Mix** - Part 3 of 3

(a) EPIC (b) JPEG (c) MPEG-2



(d) Ghostscript (e) Mesa Mipmap (f) Mesa Osdemo (g) Mesa Texgen

Figure 2: **UCLA MediaBench Visual Data Sets**



(a) Unified Cache

Figure 4: **Average Unified Cache Miss Ratios** - 32 byte lines, 2-way associative

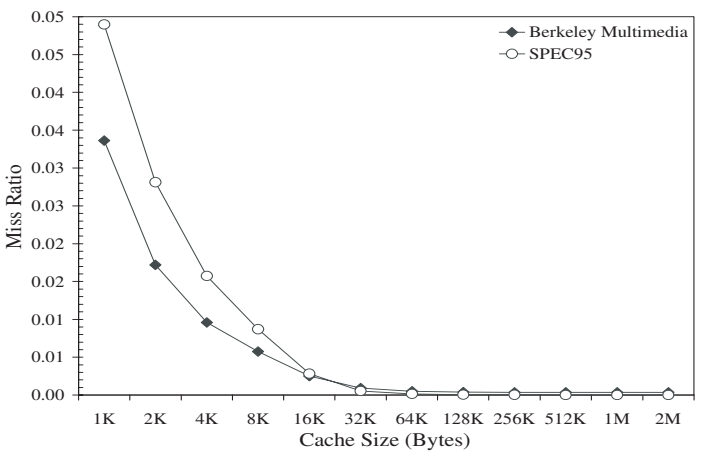


Figure 5: **Average Instruction Cache Miss Ratios** - 32 byte lines, 2-way associative

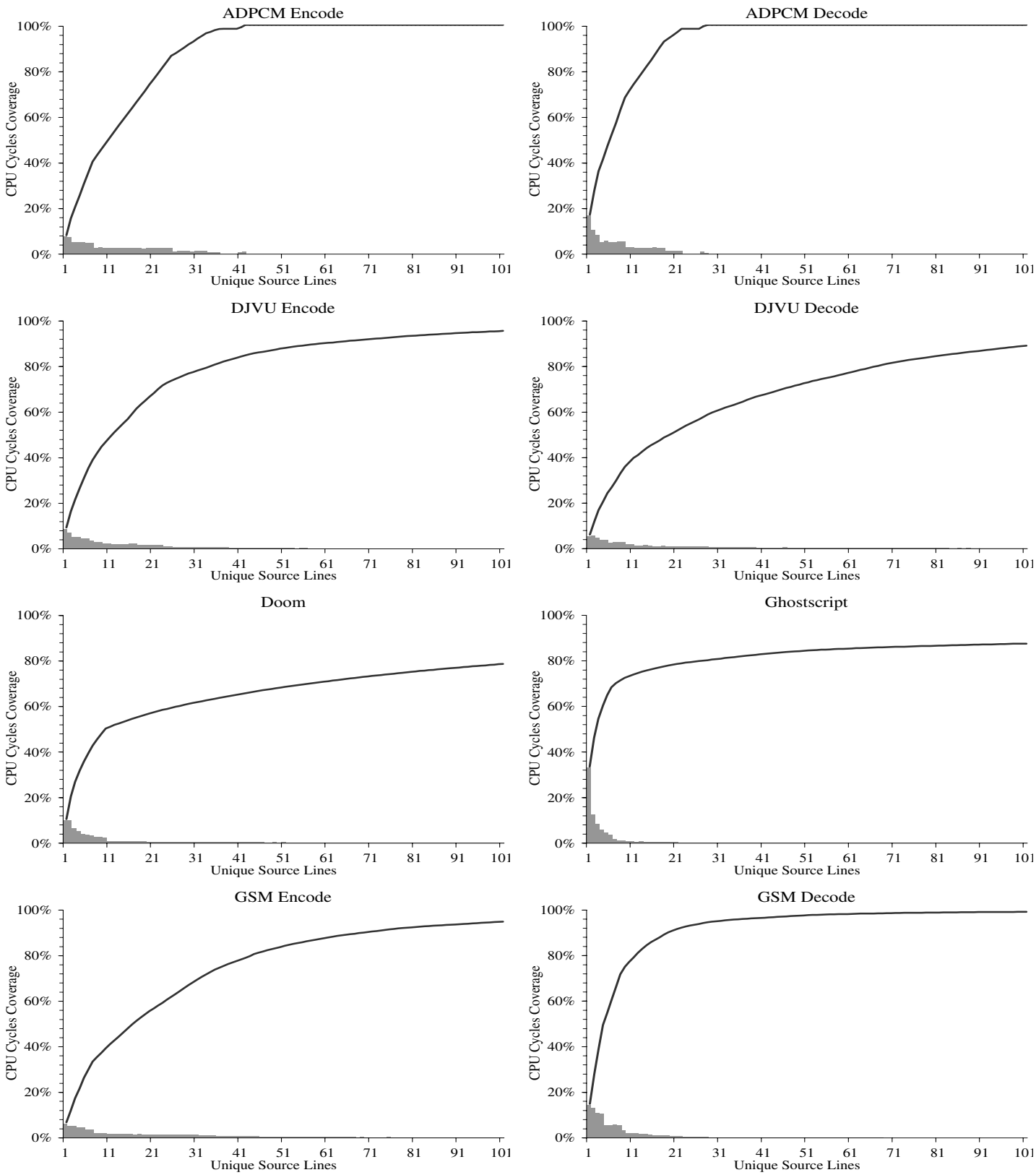


Figure 7: **Per-Kernel Source Line CPU Time Coverage** - Part 1 of 4

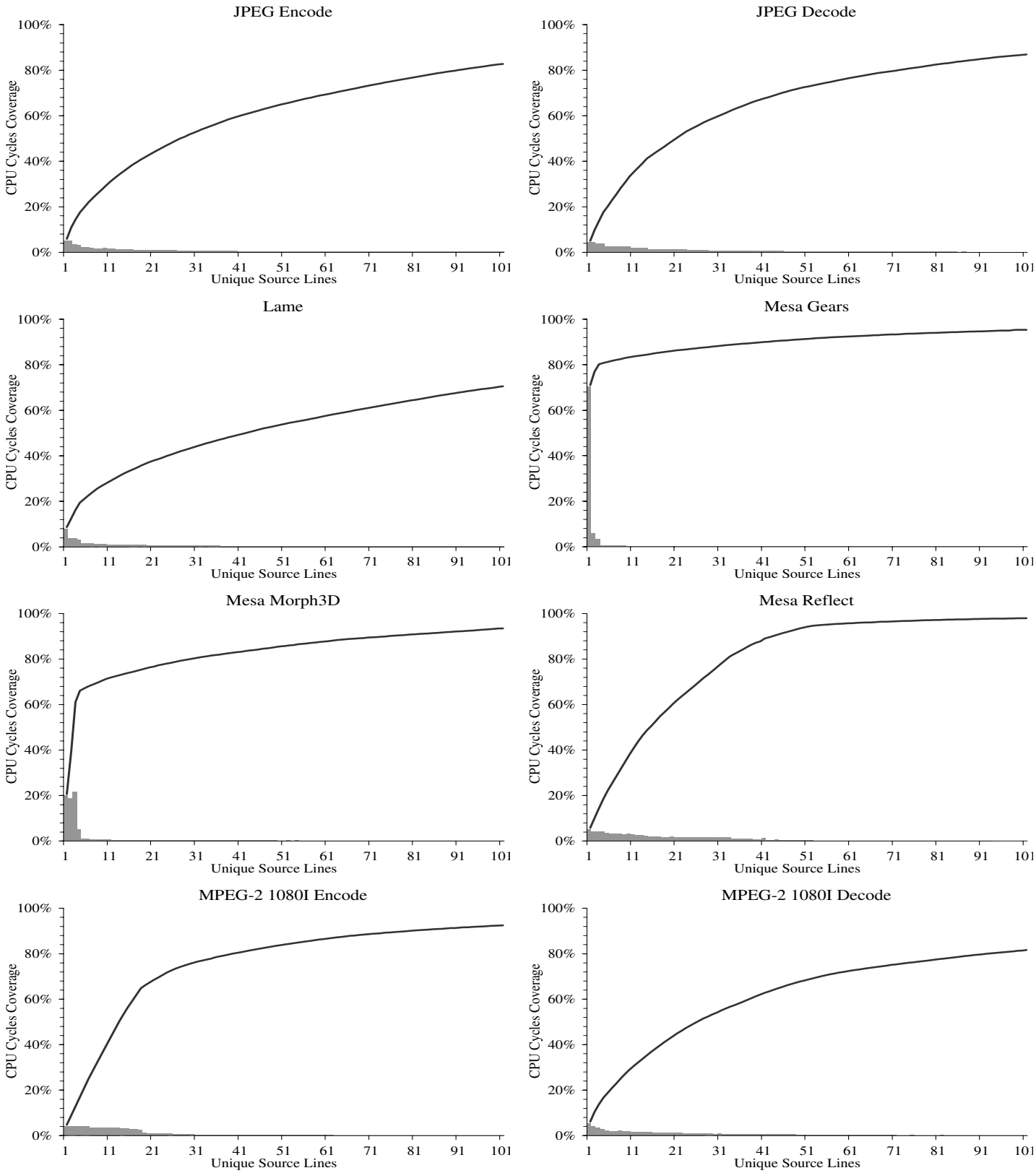


Figure 8: **Per-Kernel Source Line CPU Time Coverage** - Part 2 of 4

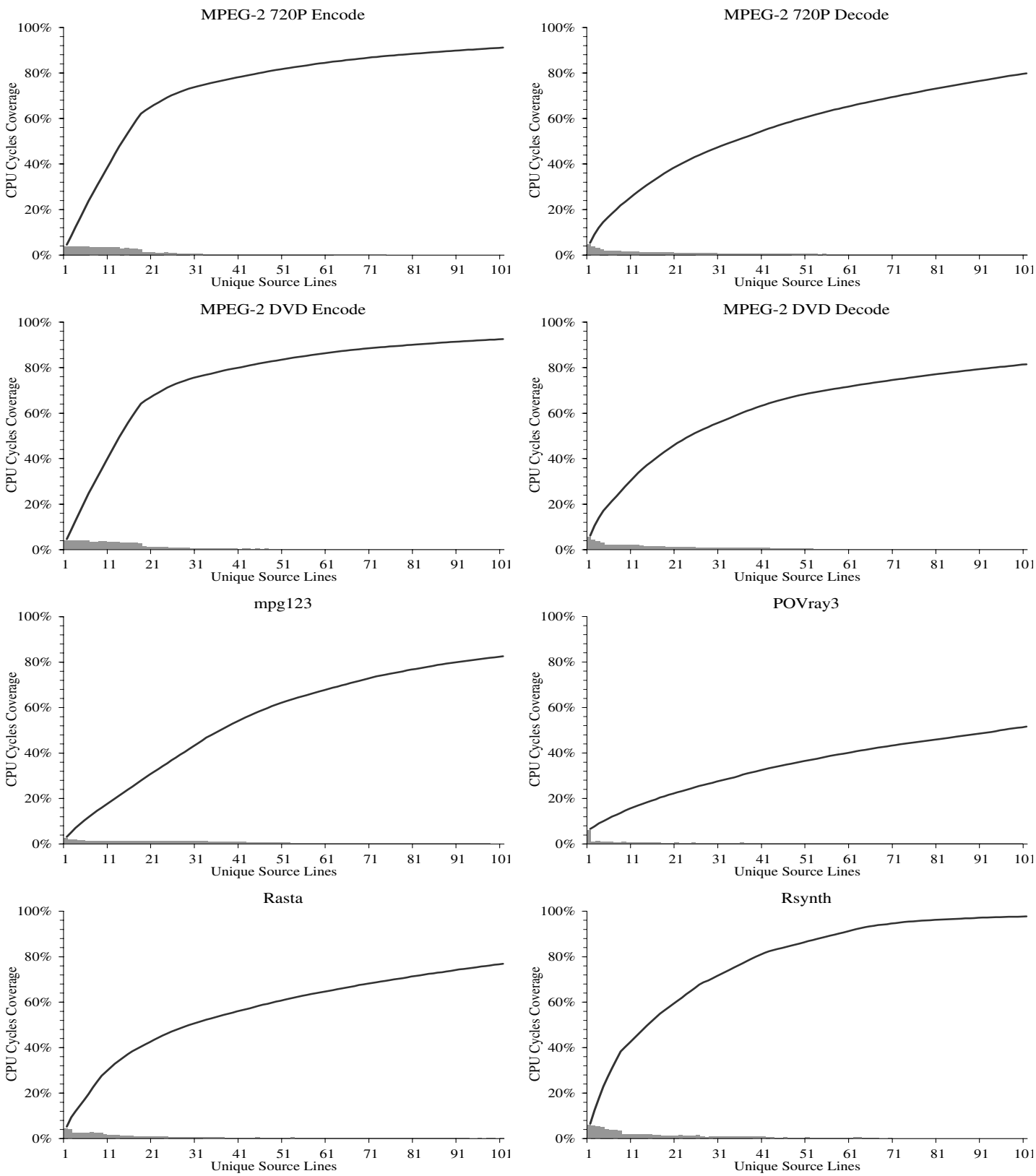


Figure 9: **Per-Kernel Source Line CPU Time Coverage** - Part 3 of 4

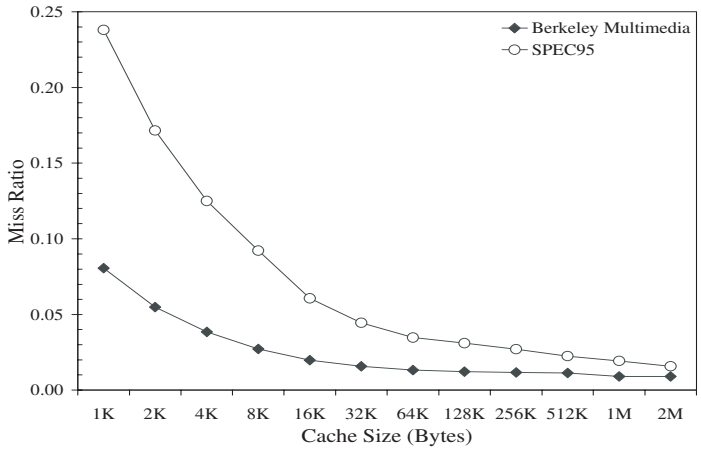


Figure 6: **Average Data Cache Miss Ratios** - 32 byte lines, 2-way associative

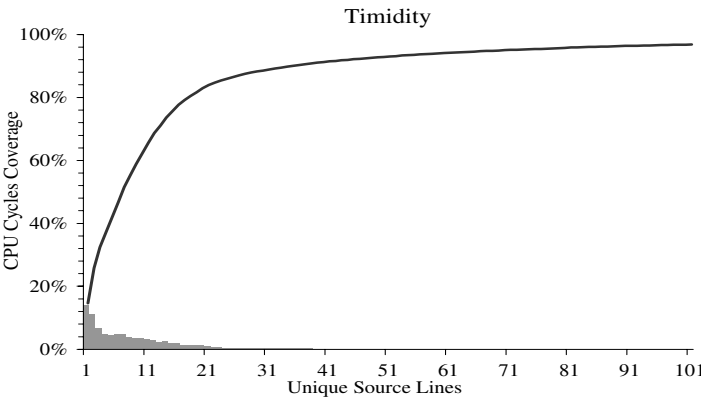
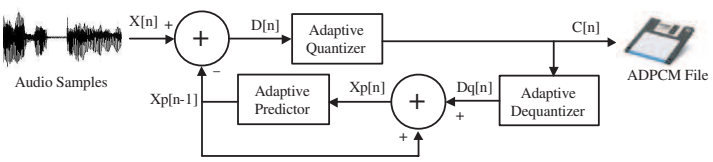
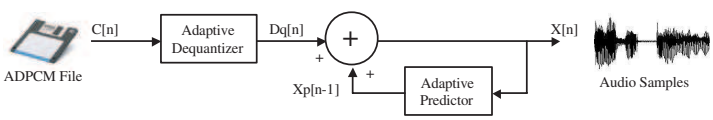


Figure 10: **Per-Kernel Source Line CPU Time Coverage** - Part 4 of 4



(a) Encoder



(b) Decoder

Figure 11: **ADPCM Codec Block Diagram**

This particular implementation is characterized by low computational overhead (real time decompression of stereo, 44.1 kHz sampling rate audio on a 20 MHz '386 class PC), and typically achieves a compression ratio of 4:1. Because ADPCM coding is a general purpose, lossy audio compression method, it has been applied to booth speech and broadband audio (e.g. music). Microsoft's WAV audio file format, which is the de facto standard for audio files on their Windows platform, utilizes a type of ADPCM compression almost identical to the IMA implementation. The European CCITT consortium has released several ADPCM based speech compression codes intended for various target bit rates: G.721 (32 Kbps), G.723 (24 Kbps).

The ADPCM data set is a mono, 28 second excerpt from Radion Konstantinovich Shchedrin's *Carmen Ballet for Strings and Percussion*, sampled at 44.1 kHz [Pope94]. This particular section of music was chosen for its large dynamic variations.

2.2 DjVu

DjVu is a document compression application from AT&T which works by splitting a raw document image into a background image, a foreground image and a mask. The mask is a bi-level image which determines whether a particular encoded pixel is a part of the background or foreground. Background and foreground images are then encoded with their IW44 continuous tone (natural image) wavelet based coding algorithm, while the mask is coded utilizing their JB2 bi-level image encoding method. [Haff98]

DjVu is distributed in such a way that the foreground/background separation algorithm is closed source, while the IW44 and JB2 algorithms are open source. The IW44 wavelet image compression utility was used as the DjVu test application. Our primary motivation in including a wavelet compression algorithm was to reflect the future algorithmic directions that image compression seems very likely to take. The JPEG 2000 standard, which is the impending successor to the JPEG standard (ISO 10918) will be based around a discrete wavelet transform, rather than the discrete cosine transform (DCT) of its predecessor. The motivating reason for this change is the fact that image quality is much higher at lower bit rates with a wavelet based transform. Wavelet compression treats an image as a continuous stream of data, where as DCT based compression standards partition an image into discrete 8x8 pixel blocks. This introduces arbitrary block boundaries that violate the continuity of the image, thus severe block artifacts are visible at low bit rates (high compression ratios) [Adel87]. A JPEG2000 codec was not included because the specification was not finalized nor in widespread use at the time of this work.

The DjVu data set is a 491x726 color digital photographic image from the Kodak's sample digital photo archive [Kodak], and is shown in Figure 12.

Figure 12: **DjVu Data Set**

2.3 Doom

Doom is a popular commercial 3D shoot 'em up video game, utilizing graphics techniques including texture mapping, non-orthogonal walls, light diminishing, light sourcing, variable height walls and ceilings, environment animation and morphing [Doom]. Although many of these features are no longer the state of the art in 3D gaming (Doom was released on December 10, 1993), our motivation in including Doom was due to its commercial origins as well as the fact that it's source code was released to the public domain; because of this latter fact, it still has a large user base of loyal game players. The source code was released to the public domain in December of 1997 by its creators, ID Software (*http://www.idsoftware.com*). Gaming applications are extremely important in driving multimedia performance in home computers. Until very recently, when the Internet and high speed network connections made streaming audio and video commonplace, gaming was the primary instance of multimedia: video, audio and 3D graphics combined within a single application.

The Doom data set consists of a pre-recorded game sequence of 774 frames (25.8 seconds at 30 fps), a frame of which is shown in Figure 13. Each frame has a resolution of 320x200 pixels.

Figure 13: **Doom Data Set**

2.4 Ghostscript

PostScript was first introduced by Adobe Systems in 1985 as a page description language for providing a device independent way in which to describe images and documents. A PostScript page description can be rendered on a printer, display, or other output device by presenting it to the PostScript interpreter controlling that device. As the *interpreter* executes commands to paint characters, graphical shapes, and sampled images, it converts the high-level PostScript description into the low-level raster data format for that particular device. The capabilities of the PostScript language are embedded in the framework of a general purpose programming language, which can be completely described in terms of ASCII characters and white space. This has facilitated the sharing of PostScript files among a wide variety of machines and operating systems, as well as creating a de facto standard for the archival and distribution of printed documents. [Adobe]

Ghostscript is an interpreter for a PostScript program. It takes as input a PostScript file, which is a script describing a series of graphics commands to render the encoded graphics and text. The output is a bitmap which can then be sent to a display or printer. Thus, it can be instrumental in either printing a PostScript file to a non-PostScript printer (a printer without an embedded version of the PostScript language) or for previewing a document on screen before printing.

Technical papers and other documents are the most common type of content encoded in PostScript files when used for archival or distribution, and are generally dominated by black and white text and figures. The data set for Ghostscript consists of the first page of Rosenblum and Ousterhout's "The Design and Implementation of a Log-Structured File System," and is shown in Figure 14 [Rose92].

The Design and Implementation of a Log-Structured File System

Mendel Rosenblum and John K. Ousterhout

Electrical Engineering and Computer Sciences, Computer Science Division
University of California
Berkeley, CA 94720
mendel@sprite.berkeley.edu, ouster@sprite.berkeley.edu

Abstract

This paper presents a new technique for disk storage management called a *log-structured le system*. A log-structured le system writes all modications to disk sequentially in a log-like structure, thereby speeding up both le writing and crash recovery. The log is the only structure on disk; it contains indexing information so that les can be read back from the log efciently. In order to maintain large free areas on disk for fast writing, we divide the log into *segments* and use a *segment cleaner* to compress the live information from heavily fragmented segments. We present a series of simulations that demonstrate the efciency of a simple cleaning policy based on cost and benet. We have implemented a prototype log-structured le system called Sprite LFS; it outperforms current Unix le systems by an order of magnitude for small-le writes while matching or exceeding Unix performance for reads and large writes. Even when the overhead for cleaning is included, Sprite LFS can use 70% of the disk bandwidth for writing, whereas Unix le systems typically can use only 5-10%.

1. Introduction

Over the last decade CPU speeds have increased dramatically while disk access times have only improved slowly. This trend is likely to continue in the future and it will cause more and more applications to become disk-bound. To lessen the impact of this problem, we have devised a new disk storage management technique called a *log-structured le system*, which uses disks an order of magnitude more efciently than current le systems.

Log-structured le systems are based on the assumption that les are cached in main memory and that increasing memory sizes will make the caches more and more effective at satisfying read requests[1]. As a result, disk trafc will become dominated by writes. A log-structured le system writes all new information to disk in a sequential structure called the *log*. This approach increases write performance dramatically by eliminating almost all seeks. The sequential nature of the log also permits much faster crash recovery: current Unix le systems typically must scan the entire disk to restore consistency after a crash, but a log-structured le system need only examine the most recent portion of the log.

The notion of logging is not new, and a number of recent le systems have incorporated a log as an auxiliary structure to speed up writes and crash recovery[2, 3]. However, these other systems use the log only for temporary storage; the permanent home for information is in a traditional random-access storage structure on disk. In contrast, a log-structured le system stores data permanently in the log: there is no other structure on disk. The log contains indexing information so that les can be read back with efciency comparable to current le systems.

For a log-structured le system to operate efciently, it must ensure that there are always large extents of free space available for writing new data. This is the most difcult challenge in the design of a log-structured le system. In this paper we present a solution based on large extents called *segments*, where a *segment cleaner* process continually regenerates empty segments by compressing the live data from heavily fragmented segments. We used a simulator to explore different cleaning policies and discovered a simple but effective algorithm based on cost and benet: it segregates older, more slowly changing data from young rapidly-changing data and treats them differently during cleaning.

We have constructed a prototype log-structured le system called Sprite LFS, which is now in production use as part of the Sprite network operating system[4]. Benchmark programs demonstrate that the raw writing speed of Sprite LFS is more than an order of magnitude greater than that of Unix for small les. Even for other workloads, such

The work described here was supported in part by the National Science Foundation under grant CCR-8900029, and in part by the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591.

This paper will appear in the *Proceedings of the 13th ACM Symposium on Operating Systems Principles* and the February 1992 *ACM Transactions on Computer Systems*.

Figure 14: **Ghostscript Data Set**

2.5 GSM

The Global System for Mobile telecommunication (GSM) protocol is currently employed in Europe for digital cellular telephony. The GSM 06.10 RPE-LTP part of the standard provides for full rate speech coding and compression of approximately 10 minutes of speech into 1 MB. The GSM compressor achieves such high compression ratios by modeling the human speech system with two filters and an initial excitation. In human speech, the glottis produces a vibration at a specific frequency which is then shaped through reflection and absorption when it passes through the vocal tract and nasal cavity. During GSM speech compression, the input speech samples are grouped into frames of 160 signed 13-bit linear-PCM values sampled at an 8 kHz sampling rate. The short term analysis section of the algorithm calculates the short term residual which will excite the short term synthesis stage of the decoder. This models the vocal and nasal tract. Long term filtering, which represents a much smaller portion of CPU time, models the glottis and deals with pitch. [Dege94]

The data set for GSM consists of an excerpt of 24 seconds of U.S. Vice President Al Gore recorded in 16-bit precision mono at an 8 kHz sampling rate: "I've traveled to every part of this country during the last six years. During my service in the United States Congress I took the initiative, in creating the Internet. I took the initiative in moving forward a whole range of initiatives that have proven to be important to our country's economic growth: environmental protection, improvements in our educational system." The source of the sound clip is from an interview with CNN's Wolf Blitzer on March 9, 1999 [CNN99].

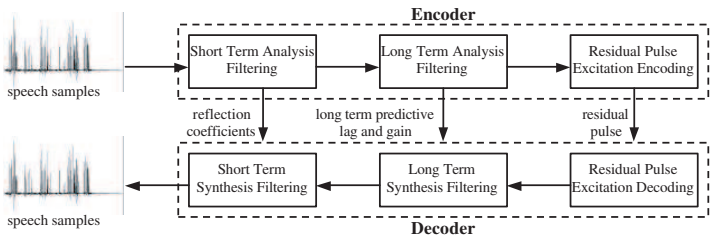
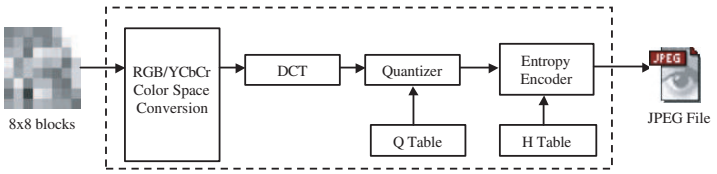


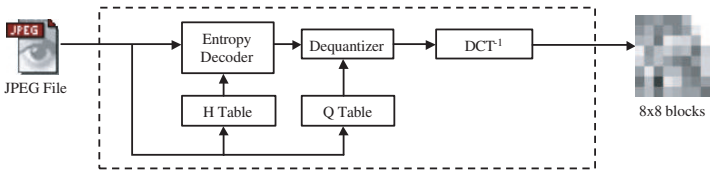
Figure 15: **GSM Speech Compression Algorithm**

2.6 JPEG

The Joint Photo Experts Group (JPEG) established the JPEG image compression standard, which was adopted as international standard ISO 10918 in 1993. JPEG image compression is a widely used method for reducing the size of natural (continuous tone) greyscale and color images; it is the de facto image compression standard both on the World Wide Web, as well as for digital cameras [Rama98]. In JPEG compression, the source image is partitioned into 8x8 pixel blocks, and each block is transformed into the frequency domain using the forward *discrete cosine transform* (DCT). The lossy part of JPEG compression stems from its psycho-visual model which recognizes that in most images the higher frequency coefficients are small and the eye is less sensitive to them, so most of the spatial frequencies in a typical 8x8 block have near zero values, and need not be encoded, or can be coded with fewer bits. This is done through the *quantization* stage which reduces the amplitude of the values that contribute little to the perceived quality of the image. Run length coding and Huffman entropy coding are then used to compress the remaining coefficients. [Wall91]



(a) Encoder



(b) Decoder

Figure 16: **JPEG Image Coding Block Diagram**

The JPEG data set is identical to that used with DjVu - a 491x726 color digital photographic image from the Kodak's sample digital photo archive [Kodak] - and is shown again in Figure 17.

Figure 17: **JPEG Data Set**

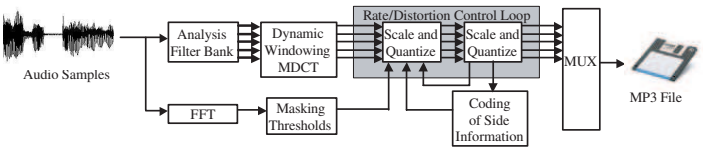
2.7 LAME and mpg123

More than any other organization, the Motion Picture Experts Group (MPEG) has had the largest hand in shaping digital multimedia compression formats. The MPEG audio

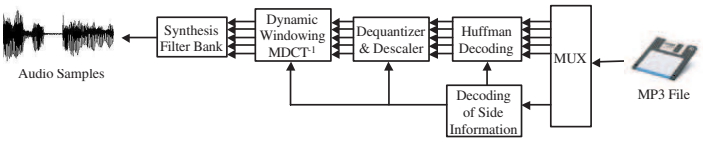
compression standard (ISO/IEC 11172-3) allows for *perceptually* lossless compression. During the design of the standard, a group of expert listeners were unable to distinguish between compressed and uncompressed audio clips with statistical significance for compression down to 256 Kbps of audio sampled at 16-bit, stereo, 48 kHz sampling rate. This represents a 6:1 compression ratio for a series of audio clips that were known to be difficult to compress. [Pan93]

Although the actual coding is lossy, the algorithm exploits perceptual weaknesses of human hearing which makes it perceived to be lossless. The presence of a strong audio signal masks a spectral range of neighboring frequencies. In addition, the ear has varying tonal acuity depending on the frequency range of the signal, thus it is possible to divide up the auditory spectrum into varying width *critical bands*, devoting a given number of bits to each band through quantization. The complete standard consists of three layers, each of which models human hearing with increasing sophistication. Based on the psycho-acoustic model (layer) involved, a given audio bitstream is encoded to minimize the quantization noise inherent in lossy compression. Layers I and II are used infrequently as their approach is not as refined as that of Layer III, and computational complexity is increasingly less of a concern, as the power of desktop computers has increased dramatically since the introduction of the MPEG standard. A block diagram of Layer III audio coding and decoding is shown in Figure 18.

The MPEG audio data set for encoding by LAME and decoding by mpg123 is a 28 second excerpt from Shchedrin's *Carmen Suite*, sampled at 44.1 kHz [Pope94]. This particular section of music was chosen for its large dynamic variations.



(a) Encoder



(b) Decoder

Figure 18: **MPEG Audio Layer III Encoder/Decoder**

2.8 Mesa

OpenGL was originally developed by SGI to be a portable 3D graphics application programmer's interface (API) for interactive applications. This differentiates it from other types of 3D rendering, such as ray tracing, in which the quality of the rendered image takes precedence over interactivity (a reasonable number of frames rendered per second). Interactivity is very important for 3D because the ability to rotate, scale or otherwise transform or move through a virtual 3D object or scene is at the heart of applications ranging from video games to medical imaging.

Mesa is an API level clone of OpenGL, which is compatible with applications written for it. In both OpenGL and Mesa, graphics primitives such as points, line segments, etc. are drawn into a frame buffer according to a selectable mode. OpenGL commands are issued via. procedure calls and accumulated in a display list for processing [Segal94]. 3D graphics rendering transforms 3D models into 2D images by simulating the physics of light propagation from lighting sources, through the objects, and eventually to the eyes. The graphics primitives in a 3D model file are organized as a tree [Chiu97]. Rendering starts with a traversal through this tree of primitives to extract the appropriate information for display such as the specific drawing primitive (line or triangle), lighting models, textures, etc. The geometry stage then transforms 3D coordinates of the model to the 2D coordinates of the screen. Finally, the rasterization stage converts transformed primitives into pixels values and stores them to the frame buffer for display. [Yang98] These three steps happen every time a scene is rendered (30+ times a second for fluid animation). A typical geometry processing pipeline is shown as part of Figure 19.

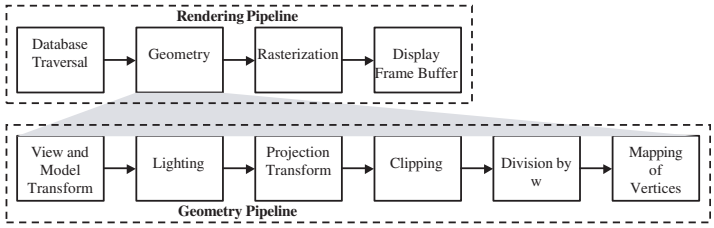


Figure 19: **3D Rendering Pipeline**

The rasterization stage is by far the most computationally intensive stage of 3D rendering. For platforms which need to support sophisticated interactive 3D applications, accelerator cards which handle the rasterization stage in hardware are the norm, with the CPU handling geometry and data base traversal. Accelerator cards are typically quite inexpensive (<$100 USD), and are sold as combination 2D/3D accelerator cards for commodity PCs.

Based on the past adoption of previously new workloads, we can predict what will happen with 3D rendering. The traditional progression for supporting a new computationally intensive workload is:

1. in software on a platform unable to provide satisfactory performance

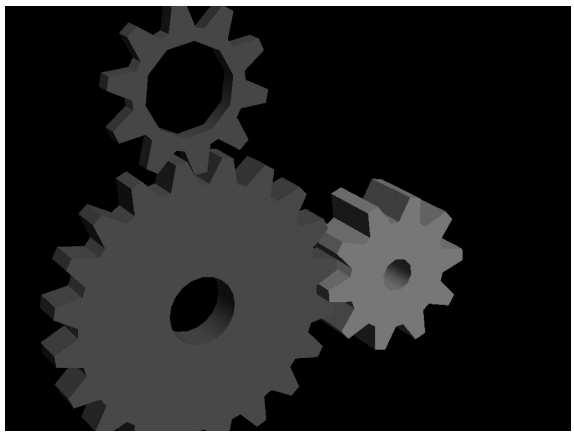
2. the addition of special purpose hardware (expansion cards) for accelerating the application

3. a folding of the special purpose hardware's functionality

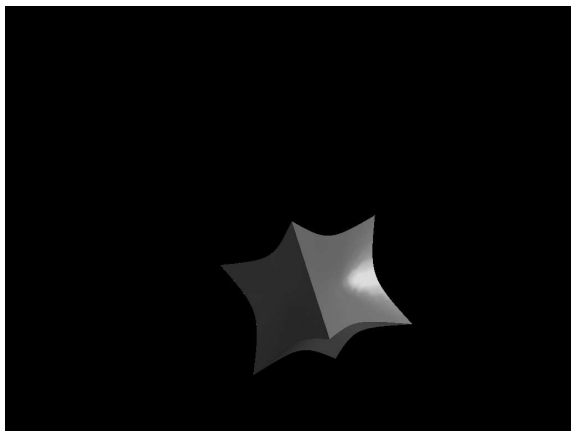
into the main CPU when enough silicon becomes available to support it

In the same way, we expect that eventually 3D workloads will follow suit, with all stages of 3D rendering handled solely by the CPU. However, the current trend is moving in the reverse direction - recent (currently high end) accelerator cards for PCs have begun incorporating the geometry processing stage as well. Rather than saturating the CPU with geometry computations, it is attractive to have it be responsible for synthesizing more detailed and realistic motions of on-screen objects [Kuni99]. This indicates that current 3D rendering capabilities are so far from meeting performance demands that there is still considerable expansion to be made into to the second stage of workload adoption (specialized hardware support). As we will see in our introduction of the POVray3 raytracing application, real time rendering of highly detailed and realistic scenes is far from being tractable on current architectures.

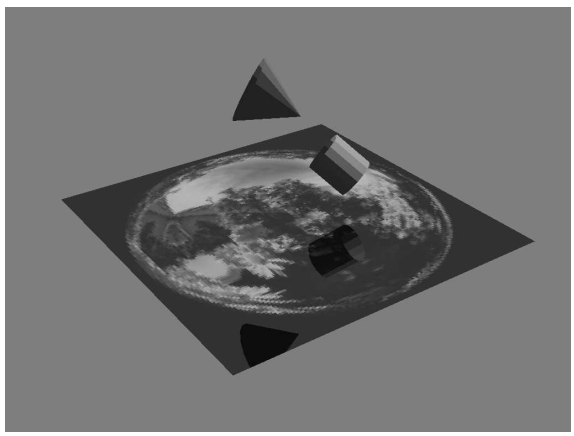
For the Berkeley multimedia workload, we focus on three relatively simple animated demonstration applications. A frame from each demo is displayed in Figure 20. In the workload, each application renders 30 frames at 1024 x 768 resolution (24-bit color). The Gears demonstration is a simple rendering application in which three colored gears rotate around their axes. Morph3D is similar to many 3D screen saver applications in that a geometrical shape is continuously rotated and morphed into other shapes while moving in a varying trajectory on the screen. The Reflect application utilizes simple texture mapping, reflection and transparency with a rotating plane under cone and cylinder shaped objects.



(a) Gears



(b) Morph 3D



(c) Reflect

Figure 20: **Mesa Data Sets**

2.9 MPEG-2 Video

The ISO Motion Pictures Experts Group (MPEG) was formed in May of 1988, bringing together the technical expertise of people from all industries interested in digital audio and video applications. The MPEG-1 standard or ISO/IEC 11172 (introduced in 1990) was the result of this collaboration, specifying a platform independent, digital solution for storing moving pictures and audio on a CD with video quality comparable to that of a VHS cassette (1.5 Mbps). MPEG video compression is a hybrid of algorithms, allowing it to achieve extremely high compression ratios. *I frames* or *intra-coded frames* utilize the discrete cosine transform (DCT) followed by quantization for intra-frame compression to remove spatial redundancy within a single frame. This is almost identical to JPEG image compression. *P frames* or *predicted frames* use a reference frame from a past I or P frame to construct a new frame. *B frames* or *bidirectionally predicted frames* reference both past and future I or P frames. These inter-coded frames exploit temporal redundancy between frames to reduce the number of bits necessary to encode a frame.

An MPEG video bitstream or sequence consists of a hierarchy of structures, as depicted in Figure 21. Each level of the hierarchy consists of a start code or header which is a unique 32-bit patterns marking the divisions between different sections of the bitstream.

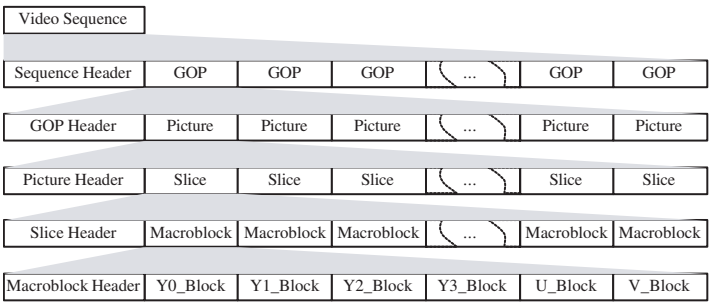


Figure 21: **MPEG Video Stream Hierarchy**

The next level of the hierarchy beneath the video bitstream is the *group of pictures* or GOP. A GOP must contain at least one I frame and any number of P and B frames. The total length of a GOP typically ranges anywhere between 8 and 20 pictures in length. The actual sequence of frame types to use is not part of the MPEG specification, and must be decided by the end user or encoding software. To visualize how this works, consider the example eight frame GOP shown in Figure 22.

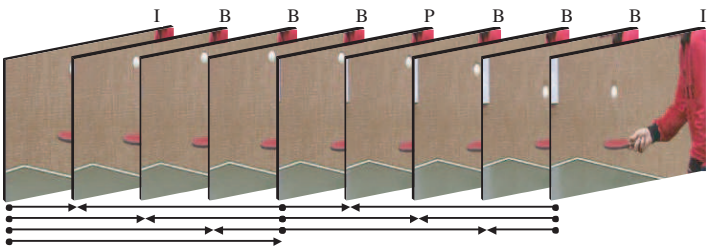


Figure 22: **Example MPEG Video Group of Pictures**

Each GOP is subdivided into slices, which in turn are subdivided into *macroblocks*. A macroblock is a 16 x 16 array of pixels. Each I frame contains only I macroblocks - macroblocks coded in the intra-frame, DCT based compression method. A P frame may contain both forward predicted (P) and I macroblocks, while a B frame may contain bidirectionally predicted (B) macroblocks as well. Because MPEG utilizes the YC_bC_r color space and color space sub sampling, every macroblock is subdivided into six 8x8 sub-blocks of data, consisting of four luminance (Y) sub-blocks and two chrominance (one C_b, one C_r) sub-blocks, as shown in Figure 21.

If measured by its influence on the digital coding of video and audio on desktop computers alone, MPEG-1 would be considered a great success. However, it was with the advent of MPEG-2 (November, 1994) that the MPEG group mapped the future of digital television. MPEG-2 (ISO/IEC 13818) extended and generalized its predecessor by supporting several levels of bit rates and encoding features. Both digital versatile disc (DVD) and high definition television (HDTV) employ the MPEG-2 standard.

2.9.1 Digital Versatile Disc (DVD)

Digital Versatile Disc or DVD is the technological successor to CDs as an optical storage media. Physically, a DVD resembles a CD in that it is a 12 cm optical disc. However, its capacity is much greater, as is shown in Table 7 [Kabl96]. And, unlike a CD, a DVD can be double sided, allowing for a maximum storage capacity per disc of 17.08 GB.

CD	Single Layer DVD	Dual Layer DVD
680 MB	4.70 GB	8.54 GB

Table 7: **Physical DVD Storage Formats**

Although designed as a unified medium for storing audio, video, and data, DVD is best known for its DVD-Video component for storing movies using MPEG-2 encoding. Stand alone DVD-Video players were introduced to in Japan in December 1996 and March 1997 in the United States [Naka98]. Software players have become a popular solution for inexpensive digital versatile disc (DVD) playback on personal computers, applications which are in large part possible because of the advent of SIMD multimedia extensions. DVD-Video was the first DVD based consumer product, so we will refer to DVD-Video simply as DVD.

In addition to superior video quality in comparison to video cassettes, DVD offers advances in sound reproduction with AC3 (Dolby Digital) 5.1 channel audio, as well as allowing for a degree of interactivity. DVD players can randomly access any scene in a movie via a menuing system as well as provide for ancillary information such as multiple audio track languages, multiple camera angles and multiple language subtitles. Different DVD titles support these features to varying degrees.

An MPEG system bit stream consists of three sub streams: audio, video and sub picture (or system). The total peak bit rate for the MPEG system stream supported by DVD-Video is 9.8 Mbps. During the design of DVD it was found that a variable bit rate MPEG-2 stream at a peak bit rate of 7.5 Mbps (average of 3.5 Mbps) is sufficient to achieve good picture quality for a wide range of content (animation, movies, etc.). Because approximately 135 minutes of storage is required to cover 95% of Hollywood movies, the single sided, single layer DVD format can be used for typical length movies [Yama97]. The audio stream bit rate for 5.1 channel Dolby Digital is 384 Kbps.

2.9.2 High Definition Television (HDTV)

In the United States, the NTSC analog television standard has been around since 1941 (extended to color in 1955). Based on analog technology, terrestrial broadcast NTSC suffers from "ghosting" and other artifacts due to the electromagnetic interference of other electronic devices and ground structures. In addition, the resolution and color accuracy of reproduced images is far from perfect. The purpose of high definition television (HDTV) is to eliminate these deficiencies.

Like MPEG, the HDTV specification was also spurred by an international committee. The Advanced Television Systems Committee (ATSC) is an international organization begun in 1982 to begin developing voluntary technical standards

for the entire spectrum of advanced television systems. In the United States, the ATSC standard was adopted by the Federal Communications Commission (FCC) in December of 1996, with the goal of eliminating all terrestrial broadcasts of NTSC programming by 2006 (to put this in perspective, it is estimated that there are currently 250,000,000 NTSC televisions in American households [HDTel]). Extremely limited terrestrial broadcasts of HDTV signals began in late 1998.

MPEG-2 was chosen as the basis of the proposed high definition television format. The ATSC Digital Television Standard specifies a bit rate of 19.28 Mbps for a 6 MHz terrestrial channel (radio frequency modulation and transmission) and 38.57 Mbps for a 6 MHz cable television channel [ATSC95a], [ATSC95b] (see Table 8).

Format	Aspect	Horiz	Vert	Bitrate (Mbps)
DVD	4:3/16:9	720	480	9.8
HDTV 720P	16:9	1280	720	19.28/38.57
HDTV 1080I	16:9	1920	1080	19.28/38.57

Table 8: **Proposed HDTV Formats**

Both of the bit rates specified are far too demanding for current DVD designs. There is currently no official specification for storing HDTV resolutions on a DVD type medium, but it is reasonable to expect that because most HDTV's will be designed to handle to the bit rate of 38.57 Mbps, this rate will be a logical choice for the bit rate stored on high definition digital video discs (HD-DVDs). A two-hour movie at the higher quality rate of 38.57 Mbps will potentially consume over 34.7 gigabytes of space. It is likely that a new storage medium will need to be introduced to support the higher bit rate demands of recorded HDTV or HD-DVD. Current developments in the area of high frequency blue and violet lasers (allowing for the more compact storage of bits) are a possible technological solution to extending the DVD format to greater storage capacity.

2.9.3 Data Set Selection

In order to represent current and future applications, three data sets were developed for MPEG-2 encoding and decoding. Each data set consists of 16 frames (one group of pictures or GOP), a frame from each data set is depicted in Figure 23.

2.10 POVray

Ray tracing (also referred to as *ray casting*) is a technique for the photo-realistic rendering from a *scene* or 3D description of objects, light sources, textures, etc. For each pixel in the viewing window, an *eye ray* is fired from the center of projection through the pixel's center into the scene. The pixel's color is then set to that of the object at the closest point of intersection. Rays of light are traced from the camera or viewer to the objects in the scene rather than the reverse direction, as occurs in reality, for the sake efficiency - only rays visible to camera need be computed. [Foley] If an eye ray intersects an object, secondary *shadow rays* are generated towards each light source to measure the contribution to the object's illumination by each light source as well as determining which are in shadow due to being blocked by other objects. All objects are associated with material properties, such that if a surface is reflective, additional reflected rays are generated and followed in the same way. If a surface is transparent, refracted rays are generated. The illumination at any point is computed from the object's material properties as well as the angles between rays and the light sources. [Maed94]

The Persistence of Vision raytracer (POVray) element of the Berkeley multimedia workload is a high quality freely available ray tracing application widely used by graphics artists. The data set used is a POVray scene file by artist Robert A. Mickelsen titled "Desert Ammonites" [Mick95], and rendered at 640 x 480 resolution in 24-bit color. He gave permission to include this data set in the Berkeley multimedia workload. The final rendered image is shown in Figure 24.

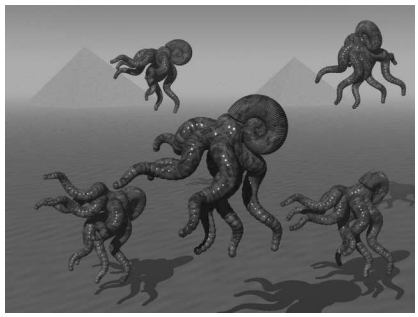


Figure 24: **POVray Data Set**

2.11 RASTA

Automatic speech recognition has as its task the decoding of the linguistic message embedded in a digitized speech signal. The linguistic message is coded in the movements and shape of the human vocal tract. The RASTA speech recognition technique relies on the fact that the rate of change of the non-linguistic components in speech often lie outside the typical rate of change in vocal tract shape. By suppressing the spectral components that change more slowly or quickly than the typical range of change of speech the performance and accuracy of speech recognition can be improved. RASTA is able to compensate for additive noise (noise uncorrelated to the speech signal) and spectral distortion in the input speech samples. [Herm94]

For each frame of speech to analyze, the following operations are performed:

1. compute critical-band power spectrum
2. transform through a compressing static non-linear transformation
3. filter the time trajectory of each component
4. transform through expanding static non-linear transformation

(a) DVD

(b) HDTV 720P

(c) HDTV 1080i

Figure 23: **MPEG-2 Video Data Sets**

5. multiply by the equal loudness curve and raise to power of 0.33 to simulate the power law of hearing

6. compute an all-pole model of the resulting spectrum

As such, RASTA is not a complete speech recognition application - there is no step searching a known dictionary of phrases or words for a match. Rather, this represents the back end processing that must occur in order for each utterance to be recognized. This limitation was deemed acceptable due to the lack of a full open source speech recognition application at the time of our study.

The data set for GSM consists of an excerpt of 24 seconds of U.S. Vice President Al Gore recorded in 16-bit precision mono at an 8 kHz sampling rate: "I've traveled to every part of this country during the last six years. During my service in the United States Congress I took the initiative, in creating the Internet. I took the initiative in moving forward a whole range of initiatives that have proven to be important to our country's economic growth: environmental protection, improvements in our educational system." The source of the sound clip is from an interview with CNN's Wolf Blitzer on March 9, 1999.

2.12 Rsynth

Speech synthesis, like speech recognition, attempts to endow computers with the ability to communicate with people in the way most natural to them. Early attempts at speech synthesis concatenated pre-recorded words into sentences, producing extremely unnatural sounding speech. This is due to the fact that in continuous speech word durations are typically shorter than when individual words are spoken, in addition to coarticulation effects in which the articulation of a word is affected by the preceding phoneme (A *phoneme* is the smallest unit of speech that distinguishes one utterance from another in a particular language. An *allophone* is the acoustic manifestation of a phoneme. [Hall95]). A *formant* speech synthesizer works by deriving an approximation to a speech waveform through a set of rules formulated in the acoustic domain. Synthesized speech is generated by modeling the human speech system as shown in Figure 25. According to the model, there are three classes of sound which compose English speech:

1. voicing - quasi-periodic vibration of the vocal folds (cords)

2. aspiration - generation of turbulence noise by the rapid flow of air past a narrow constriction at the level of the vocal folds

3. frication noise - generation of turbulence noise by the rapid flow of air past a narrow constriction above the larynx

Voicing is generated within the synthesizer by digital resonators, while aspiration and frication noise are created through a linear congruential random number generator. These three sources are then combined and pass through a transfer function for either laryngeal or frication sources. These transfer functions model the effect of the human vocal tract on the sound sources.

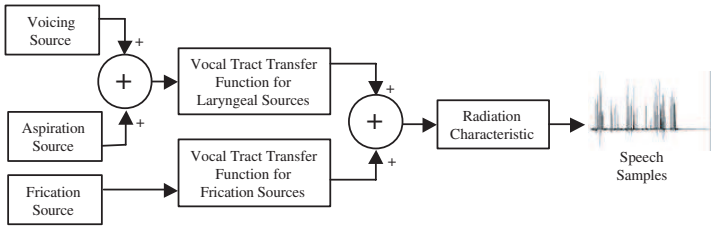


Figure 25: **Klatt Cascade/Parallel Formant Speech Synthesizer**

Rsynth is the result of Nick Ing-Simmons' work to integrate several pieces of public domain code into a text to speech application. The speech synthesizer portion of Rsynth is based on the synthesizer from [Klatt80] with a modified voicing source added by Klatt in 1982. (The original published Klattalk source code was in Fortran, but was recoded to C by Ing-Simmons.) The data set spoken is the first two paragraphs (181 words) of text from the United States Declaration of Independence, requiring 90 seconds to synthesize. This particular text was chosen because it is widely available in ASCII text form.

2.13 Timidity

TiMidity is a MIDI file rendering applications which uses instrument sounds encoded in Gravis Ultrasound-compatible patch files to generate digital audio data from MIDI files. Music delivered by MIDI files is the most common use of MIDI today, and is found in many PC games, web pages, etc. Most personal computers are capable of playing standard MIDI files out of the box. The reason for the popularity of MIDI files is that, unlike sampled digital audio files (.wav, .aiff, etc.) or even compact discs or cassettes, a MIDI file does not need to capture and store actual sounds. Instead, the MIDI file is just a list of events which describe the specific steps that a soundcard or other playback device must take to reproduce a piece of music. Each action of a musical performance is assigned a specific binary code. This way, MIDI files are considerably smaller than digital audio files, and the events are also editable, allowing the music to be rearranged, edited, even composed interactively, if desired. [MMA] The MIDI data set is a rendition of the X-files television show theme song (originally by Chris Carter) rendered by Mark Snow [Snow96].

3 Appendix C - Kernels

In order to optimize a computational kernel so that it is as fast as possible, but still correct, it is first necessary to understand the basis of the algorithm behind it. In this section we will give an overview of the algorithms which dominated processing time in the Berkeley multimedia workload suite. The C source code for each kernel, as originally extracted

from the Berkeley multimedia workload applications, is given in Appendix C.

Tables 9, 10, 11, 12 and 13 list the dominant, computationally important kernels for each application. Note that these kernels do not necessarily correspond to a single procedure within the source application. Instead, we have listed semantically different tasks as the kernels rather than source procedures. This is due to the fact that the programmers of each application divided the algorithmic tasks into an arbitrary number of procedures or functions. More experimental applications, not concerned with speed, tend to be split into very small granularity tasks for ease of understanding and debugging. Applications that are more highly optimized tend to use very large procedures to reduce the amount of overhead.

For each kernel, the *Lines* column refers to the number of static source code lines which make up the kernel. The *Refs*, *Cycles*, and *Insts* columns each indicate the dynamic contribution of that kernel to total line references, CPU cycles and instructions respectively within the application. This gives three ways to understand the degree of coverage a particular kernel is responsible for. Kernels which accounted for less than 5% of any of these measures are not usually listed in the tables unless there was some extenuating reason to do so.

3.1 Audio

3.1.1 ADPCM Codec

The ADPCM encoding and decoding applications each consist of a single small kernel. In encoding, the IMA ADPCM algorithm computes the difference between the current samples, $X[n]$, and the previous predicted sample, $X_p[n]$, and uses the difference to compute the quantizer level, $D[n]$ for each sample. The predictor in this case is not adaptive (does not change according to the rate of change of the waveform), and so no side information is needed to encode how to reconstruct the predictor. The IMA implementation of the ADPCM algorithm simply uses the previous sample value as a predictor for the next sample, so $X_p[n]$ is simply a time delayed version of $X[n]$. Adaptation to the audio signal takes place only within the quantizer.

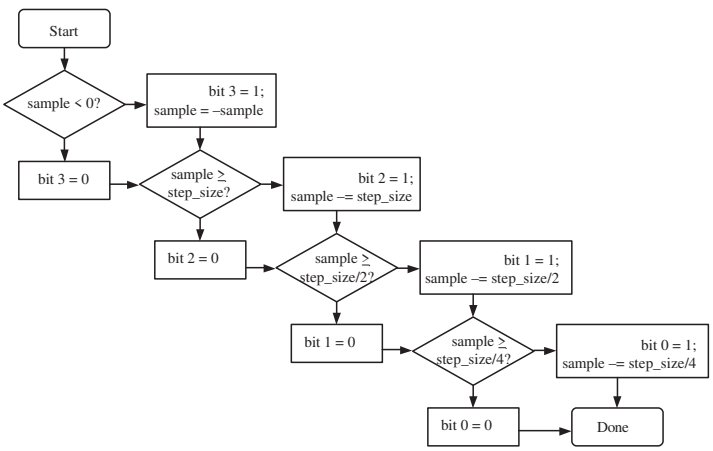


Figure 26: **IMA ADPCM Quantizer**

Quantization for the IMA ADPCM algorithm proceeds according to the diagram in Figure 26. The quantizer adapts the step size based on the current step size and the quantizer output of the immediately previous input. This adaptation is performed as two table lookups. The three bits representing the number of quantizer level serves as an index into the first table lookup whose output is an index adjustment for the second table lookup. This adjustment is added to a stored index value, and the range limited result is used as the index to the second table lookup. The summed index value is stored for use in the next iteration of the step size adaptation. The output of the second table lookup is the new quantizer step size.

The ADPCM decoder reconstructs the audio $X_p[n]$ sample by adding the previous decoded audio sample $X_p[n-1]$ to the result of the signed magnitude code word, $C[n]$ and the quantizer step size plus an offset of one-half the step size:

$$X_p[n] = X_p[n-1] + stepsize[n] \cdot C[n] \tag{1}$$

The value of $stepsize[n]$ is approximated by the product of the previous value, $stepsize[n-1]$, and a function of the code word, $F(C[n-1])$:

$$stepsize[n] = stepsize[n-1] \cdot F(C[n-1]) \tag{2}$$

3.1.2 Fast Fourier Transform

The most common application of Fourier transforms is the spectral analysis of discretely sampled data (the value of some real process is recorded at evenly spaced intervals of time, Δt). This is possible because a physical process can either be described in the time domain, by the values of some quantity h as a function of time t or in the frequency domain where the process is described as an amplitude H (generally a complex number indicating magnitude as well as phase) as a function of frequency f. The Fourier transform allows us to go back and forth between $h(t)$ and $H(f)$, which can be thought of as two different representations of the same function. [Pres92]

The discrete Fourier transform (DFT) has the same properties as the continuous transform. An N point DFT is given by:

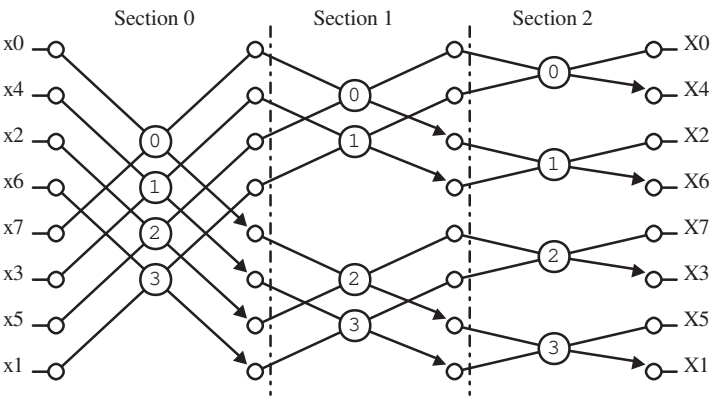
$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \tag{3}$$

Direct computation of the DFT is computationally expensive because it is $O(N^2)$. Fortunately, there is a method for computing the DFT in $O(Nlog_2N)$ time, called the fast Fourier transform (FFT), which was introduced by Cooley and Tukey in 1965 [Cool65]. This algorithm works by recursively splitting an N-point DFT into two N/2 point DFTs, continuing until only 2-point DFTs remain [Dobb95]. The output of the FFT is shuffled, but it is in a structured way. If the bits of the index into the array on which the FFT was performed are reversed, the correction index of the desired value is generated. This is known as *bit reversed addressing.*

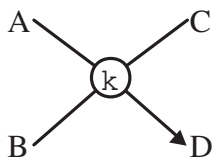
Audio					
Application	**Kernel Name**	**Lines**	**Refs**	**Cycles**	**Insts**
ADPCM Encode	ADPCM Coder	45	100.0%	98.3%	99.9%
ADPCM Decode	ADPCM Decoder	30	100.0%	98.3%	99.9%
LAME	FFT	176	18.6%	18.4%	13.8%
	Max Value	8	15.7%	15.7%	12.0%
	Quantize	55	11.6%	11.6%	15.3%
	Calc Quantization Noise	71	10.7%	10.6%	15.3%
	Count Encoding Bits	26	9.5%	9.3%	7.3%
	Psychoacoustic Model	262	5.3%	5.2%	6.0%
mpg123	Synthesis Filtering	67	45.7%	45.2%	39.6%
	DCT64	105	20.3%	20.2%	22.6%
	Dequantize	305	14.8%	15.1%	16.1%
	Parse Bitstream	100	9.3%	9.3%	9.2%
	DCT36	41	6.0%	6.1%	7.0%
Timidity	Mix	152	49.0%	48.5%	35.5%
	Resample	167	41.4%	42.1%	58.0%
	Convert Sample Format	6	6.2%	5.8%	3.6%

Table 9: Audio Kernels

Because the FFT is central to many digital signal processing applications, many DSP chips actually have bit reversed addressing modes implemented in hardware.



where a *butterfly* operation,



is defined:
$C = A + B$
$D = W_k(A - B)$
$W_k = cos\left(\frac{-2\pi k}{N}\right) + i \cdot sin\left(\frac{-2\pi k}{N}\right)$

Figure 27: **Decimation in Frequency Flow Diagram**

The literature contains a plethora of algorithmic variations on the FFT as originally proposed by Cooley-Tukey. The original algorithm is classified *decimation in time* (DIT) and can be recognized by its bit reversed reordering of the input data. A different category of FFT, originally presented by Sande-Tukey, is termed *decimation in frequency* (DIF) as it rearranges the output data in bit reversed order. The motivation for moving where bit-reversed addressing is performed is to avoid the bit reversed reordering step through the appropriate combination of DIT and DIF forward and inverse Fourier transforms if operations are to be performed on the data in the frequency domain, with the result being converted back to the time domain. In reality there is little difference computationally between the approaches as bit reversal represents only a small portion of an FFT's operation count. [Pres92] Other variations attempt to recursively split an N-point DFT into different or *multiple radii* besides the original radix-2 of the Cooley-Tukey algorithm. The idea behind other radix transforms is to take advantage of special symmetries for a given radix, splitting an N-point DFT into multiple transforms of whatever radix highly optimized transforms are available. The standard way to describe an FFT algorithm is through a flow diagram, which is given for the 8-point DIF FFT in Figure 27.

Spectral analysis is perhaps the most widespread application of the FFT, and is what we are interested in for our multimedia applications. Both the LAME MPEG-1 layer III audio encoding and Rasta speech recognition applications utilize the fast Fourier transform method for computing the discrete Fourier transform. Because most applications of the FFT (LAME and Rasta included) exclusively involve real input sequences, there are several techniques for computing the FFT in these cases beside the trivial and inefficient method of zeroing the imaginary part of the input data to a complex FFT.

1. mass production - also called a double sequence FFT. Computes two N-length real FFTs simultaneously by

placing one set of input data in the real portion of the complex array to transform, and the other in the imaginary part. The resulting array can than be split into two complex results based on a set of symmetries. This method works well if the condition of having two real sequences to transform at the same time can be met.

2. trigonometric recombination - uses a complex FFT of length N/2 to compute the DFT of a real input sequence of length N. Before the complex FFT is performed, the real input sequence is split into a real part consisting of the even indexed input elements, and an imaginary part from the odd indexed input elements. The correct output can then derived from a set of recombination computations. [Embr]

3. real value FFT (RVFFT) - RVFFT algorithms are specifically optimized for computing the FFT on real data. They take advantage of symmetries due to the real-valued nature of the input data that can be used at every stage of the FFT algorithm to remove redundant operations. [Sore87]

FFT algorithms are almost exclusively written in floating point because the accuracy of the FFT degrades quickly with fixed point calculations. In order to have perfect accuracy for an N point FFT for a k-bit input signal, it is necessary to have k+log2(N) bits of precision during intermediate calculations. So in the case of the 4,096 point transform used in the LAME application, 28 bits of accuracy are required for a numerically correct FFT implemented in fixed point. Single precision floating point numbers have only 24-bits of actual precision, but floating point (as opposed to fixed point) tends to hide the problem so that it is less noticeable due to its larger dynamic range.

3.1.3 Maximum Value in Array

The maximum value kernel found in the Lame MPEG layer III audio encoding application searches an array of 576 32-bit signed integers for the largest absolute value and returns that value.

3.1.4 Modified Discrete Cosine Transform (MDCT)

Traditional signal processing transforms (e.g. the unmodified DCT or FFT) suffer from blocking artifacts due to the processing of discrete length blocks of input data. Lapped transforms have a 50% overlap between successive blocks, which greatly reduces these artifacts [Cheng99]. The modified discrete cosine transform (MDCT) is a linear lapped transform based on the idea of time domain aliasing cancellation (TDAC). The MDCT is critically sampled, meaning that it is 50% overlapped such that a single block of inverse MDCT data does not directly correspond to the original block on which the forward MDCT was performed. This overlapping characteristic makes the MDCT very useful for quantization as it effectively removes the otherwise easily detectable blocking artifacts between transform blocks. [Linc98]

The forward MDCT (FMDCT) is defined:

$$X(m) = \sum_{k=0}^{n-1} f(k) \cdot x(k) \cdot cos(\frac{\pi}{2n}(2k + 1 + \frac{n}{2})(2m + 1)) \quad for\ m = 0 \ldots \frac{n}{2} - 1 \tag{4}$$

and inverse MDCT (IMDCT):

$$y(p) = \frac{4 \cdot f(p)}{n} \cdot \sum_{m=0}^{\frac{n}{2}-1} X(m) \cdot cos(\frac{\pi}{2n}(2p + 1 + \frac{n}{2})(2m + 1)) \quad for\ p = 0 \ldots n - 1 \tag{5}$$

where for either the FMDCT or IMDCT, $f(x)$ is a window with certain properties. The sine window:

$$f(x) = sin\left(\pi \frac{x}{n}\right) \tag{6}$$

satisfies these properties.

In MPEG audio encoding (only layer III), the size, of the MDCT is $n = 36$ (the 36 input samples from the synthesis filtering bank).

3.1.5 Mixing and Conversion

Audio mixing consists of multiplying a vector of k input signals $(S_1, S_2, \ldots S_k)$ by a vector of mixing coefficients $C_1, C_2, \ldots C_k$ and summing the result. In Timidity, fixed point integer computations are used to mix the various signed 16-bit instrument sounds into a 32-bit output buffer. After all of the required instrument sounds have been mixed in and the buffer is full, it is then rounded and scaled to produce 16-bit signed integer results and written out to the output file (this is the conversion step). In this way, the 32 bit result of multiplying the 16-bit sound sample of an instrument by a 16-bit mixing coefficient is accumulated in 32-bits.

3.1.6 Quantization

Quantization is a process by which a real value is converted to a discrete integer representation. The quantize kernel is taken from the LAME MPEG audio layer III encoding application. In it, an array of real double precision floating point values, $xr[]$, is converted to an array of 32-bit integers, $xi[]$, according to the function:

$$ix[i] = \sqrt{\sqrt{xr[i]*} + 0.4054}$$

This is extremely time consuming to compute due to the square root operations, so the quantize kernel employs a look up table for certain precomputed ranges of floating point input values.

3.1.7 Resampling

In the digital signal processing of sampled audio and speech signals it is often necessary to change the sampling rate of a piece of sampled data. *Resampling* or *sample rate converting* a waveform sampled at a frequency F_{sample} causes the output sample data to have the same spectral components when

played back at a the new sample rate, $F_{playback}$. For example, if speech is sampled at 8 kHz, but is to be played back on a CD, which has a fixed playback rate of 44,100 (44.1 kHz) samples per second, the speech must be *resampled* to sound the same at the higher playback rate.

Increasing the sample rate is termed *interpolation* or *upsampling*, while conversion in the opposite direction is *decimation* or *downsampling*. Decimation reduces the number of samples per second required to represent a signal. The input signal $x(n)$ is characterized by a sampling rate $F_x = 1/T_x$ and the output signal $y(m)$ is characterized by the sampling rate $F_y = 1/T_y$, where T_x and T_y are the corresponding sampling intervals. Let us define the ratio:

$$\frac{F_y}{F_x} = \frac{U}{D}$$

where U and D are integers. Decimation can be accomplished by retaining every Dth sample and discarding the intervening $D-1$ samples. Unless the signal F_x is band limited to below the Nyquist frequency of the new sampling rate, N_y, aliasing will occur for all of the original spectral components above N_y. Recall that the Nyquist frequency is the minimum frequency at which a signal must be sampled so as to be able to perfectly recreate the original signal from the samples. The Nyquist frequency is twice the highest frequency present in the signal. In order to prevent aliasing the original signal must first be low pass filtered to remove spectral components which are not representable at the new sampling rate.

An increase in sampling rate by a factor U can be accomplished by interpolating $U-1$ new samples between successive values of the original signal, F_x. Classical *linear* (first order) or *polynomial* (higher order) interpolation can be used to generate these new samples, but the input signal must be restricted to a very narrow band so that the output will not have a large amount of aliasing. It is due to this concern that *bandlimited interpolation* is usually favored over simple polynomial interpolation. Bandlimited interpolation consists of the following steps: [Embr]

1. Expand output sequence F_y to be U times longer than the input sequence F_x by inserting $U-1$ zeros between every input sample (this is termed *zero packing*). This replicates the original spectrum U times within the output spectrum at the new sampling rate.

2. Low pass filter the expanded signal to remove the undesired $U-1$ spectra above the original input spectrum (the passband should be from $0 \ldots N_x$ and have a gain of U to compensate for the inserted zeros so that the original signal amplitude is preserved).

For sample rate conversion by a factor U/D, the third step is simply to decimate a signal which has first been interpolated by the factor U.

Timidity utilizes linear (not bandwidth limited) interpolation in fixed point (32-bit signed integers) arithmetic.

3.1.8 Synthesis Filtering

All three layers of MPEG audio coding/decoding utilize a set of 512-tap finite impulse response (FIR) filters during encoding (analysis) and decoding (synthesis). The purpose of the filter bank is to divide the original audio signal into 32 equal-width frequency subbands. During the analysis phase, the audio signal is shifted into a 512 sample buffer, 32 samples at a time. The contents of the buffer are multiplied by a windowing function, and divided into eight 64-element vectors which are summed to form the input to the modified discrete cosine transform (MDCT). The output of the MDCT yields the final 32 subband values. Decoding (or synthesis) mirrors the analysis phase. The steps of synthesis filtering in MPEG audio are shown in Figure 28. [Shli94] For the synthesis filter, the input and intermediate computations are done in floating point, but the output values are quantized to 16-bit signed integer samples.

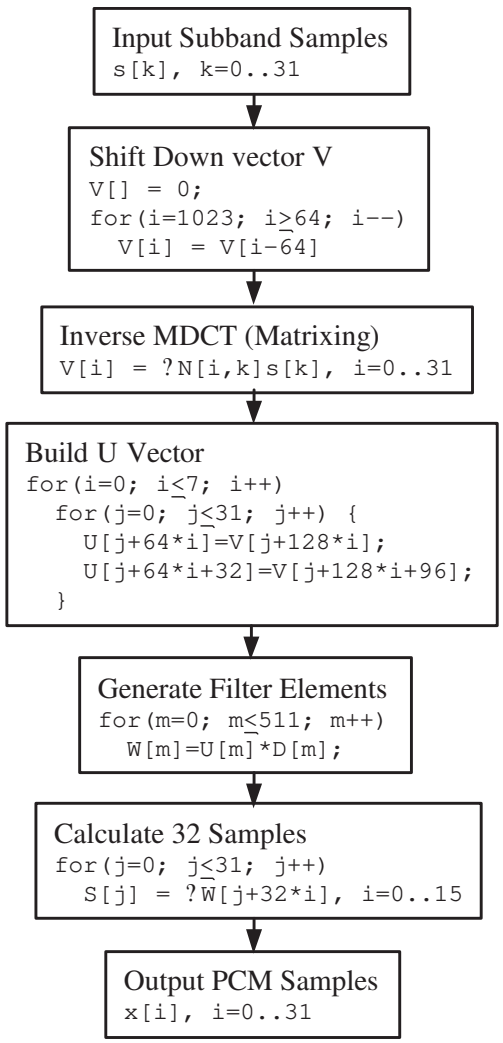


Figure 28: **MPEG Audio Synthesis Filtering**

Speech					
Application	**Kernel Name**	**Lines**	**Refs**	**Cycles**	**Insts**
GSM Encode	`Short Term Analysis Filter`	15	38.0%	35.8%	20.2%
	`Calc LTP Parameter`	61	30.9%	32.4%	51.4%
	`Weighting Filter`	16	10.2%	10.2%	5.5%
	`Sample Preprocessing`	27	6.4%	6.2%	5.2%
	`Autocorrelation`	40	6.2%	6.0%	6.6%
GSM Decode	`Short Term Synthesis Filter`	15	77.4%	77.0%	72.7%
	`Sample Postprocessing`	7	7.2%	6.9%	5.6%
	`Long Term Synthesis Filter`	11	6.1%	6.3%	10.8%
Rasta	`FFT`	168	31.0%	29.3%	17.0%
	`Estimate Noise`	162	14.6%	13.7%	9.7%
	`Critical Band Search`	26	12.9%	12.1%	7.3%
	`Fill Frame`	26	5.0%	4.7%	3.3%
Rsynth	`Parwave`	43	49.1%	49.0%	38.7%
	`Resonator`	4	22.7%	22.3%	27.8%
	`Natural Source`	8	12.1%	11.8%	9.7%

Table 10: **Speech Kernels**

3.2 Speech

3.2.1 Short Term Filtering

An *analysis filter* bank is a collection of filters $H_k(z), 0 \leq k \leq M-1$ which splits a signal $x(n)$ into M subband signals $x_k(n)$, $0 \leq k \leq M-1$. A synthesis filter bank is a set of filters $F_k(z), 0 \leq k \leq M-1$, which combines M signals $v_k(n), 0 \leq k \leq M-1$ into one signal $x'(n)$, typically termed the *reconstructed signal.* [Vaid88]

Both the short term analysis and short term synthesis filtering kernels are taken from GSM speech compression. A filter's output can depend on more than just a single input value; it can also retain state. The linear predictive short term filter (the first stage of GSM compression and last stage of decompression) models the vocal and nasal tract. It is excited by the output of a long-term predictive filter that converts its input (the residual pulse excitation or RPE) into the mixture of glottal wave and voiceless noise that makes up human speech.

The short term analysis filter is implemented according to the structure depicted in Figure 29 where:

$$\begin{aligned} d_0(k) &= s(k) \\ u_0(k) &= s(k) \\ d_i(k) &= d_{i-1}(k) + r'_i \cdot u_{i-1}(k-1) \qquad i = 1 \ldots 8 \\ u_i(k) &= u_{i-1}(k-1) + r'_i \cdot d_{i-1}(k) \qquad i = 1 \ldots 8 \\ d(k) &= d_8(k) \end{aligned} \tag{7}$$

Likewise, the short term synthesis filter of the decoder is implemented according to the structure shown in Figure 30 where:

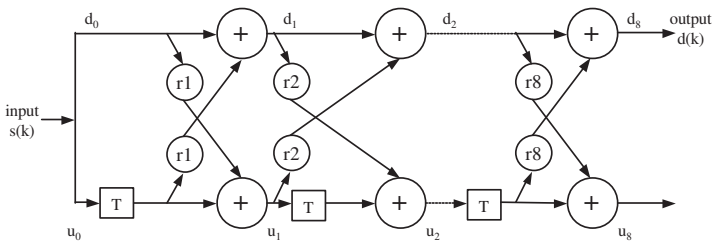


Figure 29: **GSM Encode Short Term Analysis Filter**

$$\begin{aligned} s_{r(0)}(k) &= d'_r(k) \\ s_{r(i)}(k) &= s_{r(i-1)}(k) - r_{r'(9-i)} \cdot v_{8-i}(k-1) \qquad i = 1 \ldots 8 \\ v_{9-i}(k) &= v_{8-i}(k-1) + r_{r'}(9-i) \cdot s_{r(i)}(k) \qquad i = 1 \ldots 8 \\ s_{r'}(k) &= s_{r(8)}(k) \\ v_0(k) &= s_{r(8)}(k) \end{aligned} \tag{8}$$

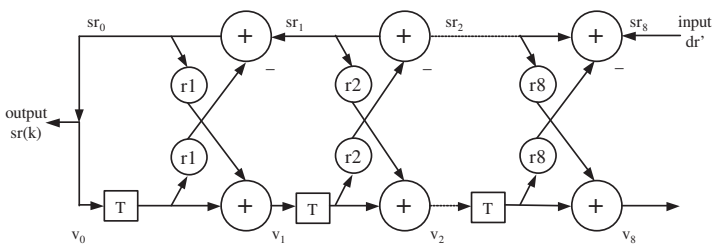


Figure 30: **GSM Decode Short Term Synthesis Filter**

3.2.2 Calculation of the LTP Parameters

The long term predictor analysis of GSM encoding selects a sequence of 40 reconstructed short-term residual values that

resemble the current values. The prediction has two parameters: the *LTP lag*, which describes the source of the copy in time, and the *LTP gain*, which is the scaling factor. To compute the LTP lag, the algorithm looks for the segment of the past that most resembles the present, regardless of scaling. This resemblance is computed by the correlation between two sequences, $x[]$ and $y[]$, which is just the sum of the products $x[n] * y[n-lag]$ for all n. The correlation is a function of the lag and of the time between every two samples that are multiplied. The LTP gain is the maximum correlation divided by the energy of the reconstructed short term residual signal (the energy of a discrete signal is the sum of its squared values). [Dege94]

Thus, the `Calculation_of_the_LTP_parameters()` procedure in the GSM encoder computes the LTP gain and the LTP lag for the long term analysis filter. This is done by calculating a maximum of the cross-correlation function between the current sub-segment short term residual signal (output of the short term analysis filter) and the previous reconstructed short term residual signal. In this GSM implementation, a dynamic scaling must be performed to avoid overflow.

3.2.3 Parwave

The `parwave()` function of the rsynth text to speech synthesizer calls all of the functions necessary for converting a frame of parameter data into sound samples. Besides calling other functions as needed, it is also responsible for the linear congruential random number generator used in the frication source. Strictly speaking, `parwave()` is not kernel - it does not perform much in the way of computation, but it is called once for each millisecond of speech synthesized, and so its contribution to CPU time due to loop overhead and function calls is significant. This is mainly a result of the way in which the rsynth speech synthesizer is designed - as an experimental tool, designed for ease of understanding and tweaking, instead of a finalized product.

3.2.4 Resonator

The basic building block of the rsynth synthesizer is a digital resonator having the properties depicted in Figure 31. The characteristics of a given resonator are specified by the resonant (formant) frequency, F, and the resonance bandwidth, BW [Klatt80]. Samples of the output of a digital resonator, $y(nT)$, are computed from the input sequence, $x(nT)$, by the equation:

$$y(nT) = Ax(nT) + By(nT - T) + Cy(nT - 2T) \qquad (9)$$

where $y(nt-T)$ and $y(nT-2T)$ are the previous two sample values of the output sequence $y(nT)$. T is the reciprocal of the sampling rate of the discrete signal being synthesized. The constants A, B, and C are related to the resonant frequency and bandwidth by:

$$\begin{aligned} C &= -exp(-2\pi \cdot BW \cdot T) \\ B &= 2 \cdot exp(-\pi \cdot BW \cdot T) \cdot cos(2\pi \cdot F \cdot T) \\ A &= 1 - B - C \end{aligned} \qquad (10)$$

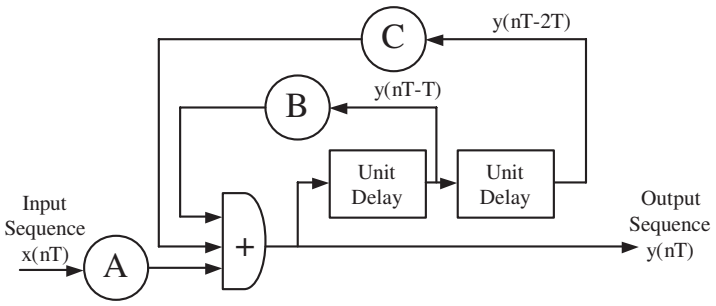


Figure 31: **Digital Resonator**

3.3 Document and Image

3.3.1 Forward/Backward Filter

The `forward_filter()` and `backward_filter()` functions, from the DjVu encoder and decoder respectively, implement the elementary wavelet transforms of the IW44 algorithm. These are based on a fast five stage lifting decomposition using Deslauriers-Dubuc interpolating wavelets with four analyzing moments and four vanishing moments [Haff98]. Although beyond the scope of this discussion, some of the details of these wavelet algorithms are presented in [Adel87] and [Swel96].

3.3.2 Encode/Decode Buckets

The DjVu wavelet coefficients resulting from the forward filter kernel are progressively encoded using a type of arithmetic coding termed ZP-coding. The ZP coder is a binary adaptive arithmetic algorithm optimized for speed. It is based on a generalization of the Golomb run-length coder, which code the length of contiguous strings of identical symbols as a binary number. A Golomb coder is defined by the parameter M, which is the number of bits in that binary number. Large values of M are preferable when long strings of repeating symbols are likely, while small values of M are preferable when long strings are unlikely. The ZP coding algorithm generalizes this technique by automatically adapting M to its optimal value. [Bott98]

3.3.3 Color Space Conversion

A color space is a model for representing color numerically in terms of three or more coordinates. Most people are familiar with the RGB color space from computer monitors, where the color and intensity of each pixel of an image is represented by the combination of three values of the colors: red (R), green (G) blue (B). The number of bits per color component is usually equally divided in RGB color spaces, so that, for example, a 24 bit per pixel display will devote 8-bits to the red component, 8-bits to the green component, and 8-bits to the blue component of each pixel.

YUV, another color space standard, has a luminance component (Y), which is a greyscale version of the image and two chrominance components (U and V) which provide color information. The exact RGB to YUV color space transformation is defined:

Document					
Application	**Kernel Name**	**Lines**	**Refs**	**Cycles**	**Insts**
DJVU Encode	`Encode Buckets`	95	49.0%	48.4%	41.0%
	`Forward Filter`	37	21.0%	22.4%	22.1%
	`Create`	28	8.7%	8.0%	9.3%
	`Init`	47	7.8%	7.2%	12.2%
	`Read Liftblock`	10	4.9%	4.7%	6.1%
DJVU Decode	`Backward Filter`	37	29.8%	28.8%	29.1%
	`Decode Buckets`	101	21.5%	22.5%	18.9%
	`Image`	39	11.9%	10.5%	10.7%
	`YCC->RGB Color Space Cnvt`	12	7.2%	6.1%	4.7%
	`Save PPM`	12	4.9%	5.1%	3.4%
Ghostscript	`printf()`		56.9%	56.9%	76.7%
	`memmove()`		13.3%	12.6%	9.2%
	`PPGM Print Row`	26	8.0%	7.7%	3.6%
JPEG Encode	`Huffman Coding`	236	54.4%	54.0%	65.0%
	`Foward DCT`	111	28.7%	28.9%	18.6%
	`RGB->YCC Color Space Cnvt`	22	10.3%	10.4%	9.8%
JPEG Decode	`Huffman Decoding`	132	21.9%	22.3%	29.9%
	`IDCT`	119	35.9%	35.7%	28.0%
	`YCC->RGB Color Space Cnvt`	25	21.7%	21.0%	22.0%

Table 11: **Document Kernels**

$$\begin{aligned} Y &= +0.299R + 0.587G + 0.114B \\ U &= 0.5643(B - Y) \\ V &= 0.7132(R - Y) \end{aligned} \tag{11}$$

The color space standard used by MPEG is international standard ITU-R 601 (also known as CCIR 601) is similar to YUV format except the U and V components are scaled and offset to produce C_b and C_r respectively. International standard CCIR-601-1 specifies 8-bit digital coding for component video with black at luma code 16 and white at luma code 235, along with chroma in 8-bit two's complement form (centered on 128 with a peak at code 224). This coding has a slightly smaller range for luma than for chroma; luma has 219 possible values compared to 224 for C_b and C_r [Poyn]. The relation between YC_bC_r and YUV color space is the following:

$$\begin{aligned} Y_{601} &= (219/256)Y + 16.5 \\ C_{b_{601}} &= (224/256)U + 128.5 \\ C_{r_{601}} &= (224/256)V + 128.5 \end{aligned} \tag{12}$$

The YUV (or YC_bC_r) color space is natural for image and video compression applications, as the eye is more sensitive to luminance (brightness) than chrominance (color). By *sub-sampling* chrominance information, the size of an image can be reduced by encoding less color information per pixel than luminance information with little or no perceptual effect. JPEG typically use 4:2:0 sub-sampling (four luminance samples for every two chrominance samples). Raw image data is frequently stored in files in RGB color space, but compression algorithms such as JPEG and MPEG utilize YC_bC_r color space.

3.3.4 Discrete Cosine Transform (DCT)

The discrete cosine transform (DCT) is the algorithmic centerpiece to many lossy image compression methods. It is similar to the discrete Fourier transform (DFT) in that it maps values from the spatial domain to the frequency domain, producing an array of coefficients representing spatial frequencies, from an array of source data. The first coefficient (the DC coefficient) is simply the average value of the input block. Later coefficients (AC coefficients) represent successively higher spatial frequencies [Kien99]. A 1D DCT is given by:

$$C(i) = \frac{a(i)}{2} \sum_{x=0}^{N-1} s(x) cos\left(\frac{\pi i(2x+1)}{2N}\right) \tag{13}$$

where the function a(x) is defined as:

$$a(x) = \begin{cases} \frac{1}{\sqrt{2}} & x = 0 \\ \\ 0 & otherwise \end{cases} \tag{14}$$

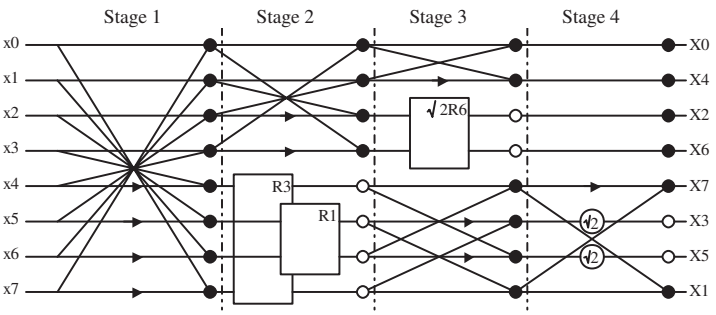
Because images are two dimensional, it is necessary to use a 2D DCT. A straightforward evaluation of a 2D DCT is quite compute intensive. Fortunately, it is possible to evaluate the 2D DCT by computing the 1D DCT of each column of pixels, followed by the 1D DCT of each row. This separability is a crucial step in developing a fast 2D DCT algorithm. A separable N point 2D DCT is defined:

$$\begin{aligned} C(j,i) &= \tfrac{a(j)}{2} \textstyle\sum_{y=0}^{N-1} cos\left(\frac{\pi i(2y+1)}{2N}\right) \\ &\cdot \left[\tfrac{a(i)}{2} \textstyle\sum_{x=0}^{N-1} s(y,x) cos\left(\frac{\pi i(2x+1)}{2N}\right)\right] \end{aligned} \tag{15}$$

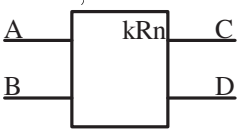
where $C(j,i)$ are the output DCT coefficients from the NxN element input array $s(y,x)$.

JPEG and MPEG encoding apply the DCT by partitioning a source image into 8x8 blocks, and computing the DCT for each block. The DCT and its mathematical inverse, the inverse discrete cosine transform (IDCT) are not lossy. Lossy compression occurs when the resulting DCT coefficients are multiplied by fixed weights based on their spatial frequencies. Higher frequency components to which the eye is much less sensitive are typically masked with smaller weightings, resulting in primarily zero values for visually unimportant features. Standard compression techniques such as run-length followed by Huffman entropy coding are then used to pack the remaining coefficients into a small number of bits.

Like the FFT, the DCT is very compute intensive, and so many different algorithms are available to evaluate it. Generally, choosing the algorithm comes down to finding one which is tuned to the architecture of interest. Although there are numerous algorithms available in the literature, one of the most commonly implemented is LLM [Loef89]. DCT algorithms are traditionally compared by counting the number of required additions and multiplications rather than in metrics more familiar to computer architects. For example, the 1D 8-point LLM DCT requires 11 multiplications and 29 additions. Flow diagrams are the standard way in which papers on the DCT present their algorithms (an 8-point example is shown in Figure 32). Shaded circles represent addition, while arrows symbolize negation. A rotation operation is depicted by a box.



where a *rotation* operation,



is defined:

$C = Akcos\left(\frac{n\pi}{2N}\right) + Bksin\left(\frac{n\pi}{2N}\right)$

$D = -Aksin\left(\frac{n\pi}{2N}\right) + Bkcos\left(\frac{n\pi}{2N}\right)$

Figure 32: **LLM DCT Flow Diagram**

In order to recover the original 2D 8x8 block it is necessary to apply the inverse discrete cosine transform (IDCT). The equation for the 2D IDCT is given by:

$$s(y,x) = \sum_{j=0}^{N-1} \frac{a(j)}{2} cos\left(\frac{\pi j(2y+1)}{2N}\right) \cdot \left[\sum_{i=0}^{N-1} \frac{a(i)}{2} C(j,i) cos\left(\frac{\pi i(2x+1)}{2N}\right)\right] \quad (16)$$

The IDCT can be computed using the same LLM algorithm shown in Figure 32, except that the process is done in reverse (from right to left). In this way, the DCT and IDCT are algorithmically identical. For many applications computational accuracy in the IDCT is not as important is in the forward DCT as no precision is permanently lost. This is because JPEG and especially MPEG follow the concept of "1 encoder to N decoders," making for non-symmetric systems with a relatively complex, high-quality encoder on one end and fast, low-complexity, low-cost decoders at the other.

3.3.5 Huffman Coding

Huffman coding is one of several compression techniques in which the way in which data in coded is based on its probability of occurance. By translating values with higher probability of occurrence into codes of shorter length, the overall size of the data is reduced. The data structure used to create Huffman codes is a weighted binary tree of Huffman tree. Huffman trees have the following properties:

1. must be a binary tree.
2. weighted - elements which occur frequently are near the top of the tree
3. each left branch is assigned a value of zero, and each right branch is assigned a value of one (or vice versa)

In order to construct a Huffman tree, two passes must be made through the data. On the first pass, a list of unique data elements and their frequencies is constructed. This is sorted in ascending order, thereby putting the most frequently occurring data elements at the end of the list. Next, the actual Huffman tree is constructed. The tree is built by taking the two elements with the lowest frequencies and making them the leaves of a tree. The parent of the two leaves is the sum of the leaves' frequencies. The tree is then inserted into the list (the parent nodes value is used to determine where to insert the tree), and the two leaves used to make the tree are removed from the list. This process continues until there is only one element left in the list, which is the parent node of the final Huffman tree.

Once the tree is constructed, the next step is to pass through the original data again and output each data element's associated Huffman code. Since the Huffman code representation for most data elements is smaller than the data element itself, data compression is achieved. Finally, the Huffman tree structure or a data structure allowing it to be recreated must also be stored in the output file in order to enable decompression. In the JPEG algorithm, four different Huffman trees are used, each of which is specified in the header of the JPEG file as a list of lists. [Fokk95]

The JPEG specification allows for Huffman coding in baseline mode, and arithmetic coding as an optional extension. Because the form of arithmetic coding specified by the JPEG standard (Q-form) is subject to patents held by IBM, AT&T and Mitsubishi, few JPEG implementations actually use it. The compression is only marginally better than Huffman coding (5%-10%) and requires the appropriate licensing.

3.3.6 System and Formatted I/O

The Ghostscript PostScript rendering applications spends more than 50% of its "computation" time doing formatted input and output with the build in `stdio` C library. These include the functions `int fprintf(FILE *fp, const char *cntrl string, ...);` which writes formatted text into the file associated with `fp`, and `int ferror(FILE *fp);`, which returns a nonzero value if the error indicator has been set for the file associated with `fp`.

Another 13% of the application is spent copying memory with the built in `memmove()` function from the C `string` library. The `memmove()` function has a declaration: `void *memmove(void *dest, void *source, size_t n);` It copies a block of `n` bytes pointed to by `source` to the block of memory pointed to by `dest`. The value of the pointer to is returned. If the blocks of memory overlap, each byte in the block pointed to by `source` is accessed before a new value is written in that byte.

3.4 Video

Video applications share many of the kernels utilized by image and documents. Additional kernels are dedicated to encoding motion between successive images.

3.4.1 Block Match

Motion estimation is used to extract the motion from a video sequence. For a P or B macroblock in MPEG video coding, one or two motion vectors are calculated respectively. This vector indicates the spatial offset from the position in the reference frame to the resulting block in the predicted frame. The coding process for P and B macroblocks finds the block that best matches the macroblock of the current frame in the reference frame within a search window, as shown in Figure 33.

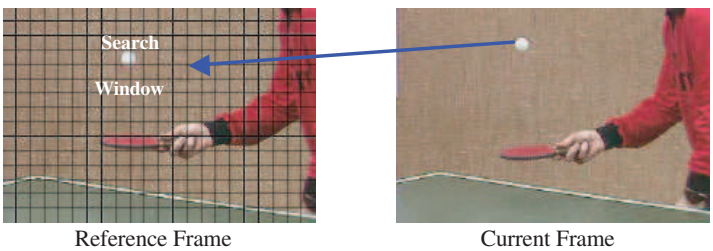


Figure 33: **MPEG Block Search**

In order to compare how well two macroblocks match, a distance criterion, $\vec{d}$, is defined:

$$\vec{d} = \min_{(dx,dy)\exists S} \sum_{(x,y)\exists W} \|L_i(x,y) - L_{i-n}(x-dx, y-dy)\| \; n \geq 1 \quad (17)$$

where W is the measurement window to compare within the search area, S. L_i is the pixel intensity (luminance) at location (x, y) in frame i. The region displacement vector for the interval $n = (i + n) - i)$ is given by dx, dy. The matching function, $\|x\|$, is typically just absolute value, but can be other functions, such as the quadratic norm (x^3).

3.4.2 Add Block

During the block reconstruction phase of motion compensation in the decoder, a block of pixels is reconstituted by averaging between the pixels in different macro blocks. The function of the add block kernel is to move or add an (8x8) sub block to a backwards reference frame or copy a reconstructed sub block to the current frame, combining motion predictions. The add block function requires 9-bits of intermediate precision.

3.5 3D Rendering

3.5.1 Geometry

The computation in the geometric transformation stage of 3D rendering is mostly floating point intensive, involving vector space operations such as matrix multiplication and inner products, and is also easily parallelizable. The rasterization stage, on the other hand, consists mostly integer arithmetic, involving simple additions and comparisons, but requires coordinated access to shared data structures during visibility computation, and is therefore more difficult to parallelize. [Chiu97] As depicted in Figure 19, the geometry computation stage consists of the following sub-stages: [Yang98]

1. view and modeling transform - Graphics primitives are transformed to the viewer's frame of reference through matrix multiplications of 1x4, 4x4, 1x3 or 3x3 vector and matrix sizes. A global coordinate system is used where 3D models (objects) are constructed and manipulated (translated, rotated, scaled).

2. lighting - The light position, color and material properties are used to calculate the object color.

3. projection - 3D objects are mapped to 2D space through matrix multiplication of 1x4 vectors and 4x4 matrices

4. clipping - objects are clipped to the viewable area to avoid unnecessary rendering

5. division by w - the $\{x, y, z\}$ components of each vertex are divided by their w component (geometry processing usually works in the homogeneous coordinate system, where all vertices are represented by $\{x, y, z, w\}$).

In our set of kernels, projection and clipping tests are combined into a single function (this mirrors the original Mesa implementation). Transform and normalize represents the view and modeling transform step of the geometry pipeline.

3.5.2 Rasterization

Rasterization is the process of converting primitives like lines and triangles (already converted into window coordinates by the transformation stage) into updates to pixel values in the window's drawable frame buffer region. Mesa's rasterization

Video					
Application	**Kernel Name**	**Lines**	**Refs**	**Cycles**	**Insts**
MPEG2 DVD Encode	Block Match	157	71.5%	70.4%	63.4%
	FDCT	14	9.2%	9.2%	12.3%
	Horizontal Sub Sample	17	2.4%	2.5%	2.6%
	Read PPM	46	2.2%	2.1%	2.9%
	IDCT	69	2.1%	2.3%	2.5%
	Quantize	15	1.8%	2.0%	2.0%
	Vertical Sub Sample	25	1.8%	1.8%	2.1%
MPEG2 DVD Decode	IDCT	75	33.3%	33.9%	29.7%
	Parse Bitstream	222	33.1%	32.7%	29.7%
	Form Prediction	124	29.3%	28.9%	26.2%
	Add Block	34	9.6%	9.4%	13.7%
	Dither	72	7.8%	7.7%	12.1%
MPEG2 720P Encode	Block Match	157	69.6%	68.4%	61.5%
	FDCT	14	9.3%	9.3%	12.4%
	IDCT	75	2.5%	2.7%	2.7%
	Horizontal Sub Sample	17	2.4%	2.5%	2.6%
	Read PPM	46	2.2%	2.1%	2.9%
	Quantize	15	1.8%	1.9%	2.0%
	Vertical Sub Sample	25	1.8%	1.8%	2.2%
MPEG2 720P Decode	Parse Bitstream	240	37.1%	36.9%	35.1%
	IDCT	75	31.6%	32.0%	27.3%
	Form Prediction	124	30.4%	30.0%	27.5%
	Add Block	34	8.2%	8.1%	12.0%
	Dither	72	6.7%	6.6%	10.6%
MPEG2 1080I Encode	Block Match	157	72.1%	71.1%	64.0%
	FDCT	14	8.9%	8.9%	12.0%
	IDCT	35	1.5%	1.7%	1.5%
	Horizontal Sub Sample	17	2.3%	2.4%	2.5%
	Read PPM	46	2.1%	2.0%	2.8%
	Quantize	15	1.7%	1.8%	1.9%
	Vertical Sub Sample	25	1.7%	1.8%	2.1%
MPEG2 1080I Decode	IDCT	75	32.7%	33.0%	29.3%
	Parse Bitstream	240	31.7%	31.5%	28.4%
	Form Prediction	67	17.2%	17.1%	15.6%
	Add Block	34	9.5%	9.3%	13.5%
	Dither	72	7.7%	7.6%	12.0%

Table 12: **Video Kernels**

3D Graphics					
Application	**Kernel Name**	**Lines**	**Refs**	**Cycles**	**Insts**
Doom	`Render Column`	15	30.6%	30.4%	37.7%
	`Render Span`	16	22.7%	21.4%	18.9%
	`Render Segment`	80	15.9%	15.8%	10.7%
Mesa Gears	`Resterize`	126	77.1%	76.2%	75.0%
	`memset()`		5.3%	5.8%	19.3%
	`Transform/Normalize`	41	4.4%	4.1%	1.0%
	`Project/Cliptest`	41	3.5%	3.3%	1.1%
	`Lighting`	77	2.7%	2.5%	0.8%
Mesa Morph3D	`Frame Buffer`	19	64.7%	60.4%	40.0%
	`Lighting`	134	13.3%	13.8%	10.7%
	`Rasterize`	43	7.1%	8.7%	25.9%
	`Transform/Normalize`	18	2.6%	3.1%	1.9%
	`Project/Cliptest`	29	2.6%	2.7%	2.2%
	`memset()`		0.7%	0.9%	8.7%
Mesa Reflect	`Transparency`	17	28.5%	28.4%	17.0%
	`Stencil`	55	15.5%	15.4%	11.7%
	`Frame Buffer`	35	13.4%	12.9%	15.3%
	`Rasterize`	7	1.7%	2.5%	25.2%
	`memset()`		0.3%	0.4%	1.5%
	`Lighting`	80	0.5%	0.5%	0.4%
	`Transform/Normalize`	18	0.2%	0.2%	0.1%
	`Project/Cliptest`	29	0.2%	0.2%	0.1%
POVray3	`Synthesize Texture`	118	23.6%	22.0%	20.4%
	`Bounding Box`	89	14.9%	14.7%	10.5%
	`Vista Buffer`	38	8.7%	8.2%	7.6%
	`Lighting`	179	11.7%	11.5%	9.3%
	`memmove()`		5.8%	6.0%	14.5%

Table 13: **3D Kernels**

stage can be divided into four sub-stages: *primitive decomposition*, *texturing, fog*, and *per-fragment operations:* [Kilg95]

1. primitive decomposition - Transforms geometric primitives like points, lines, and polygons as well as image primitives like pixel rectangles and bitmaps into window coordinates and determines which pixel locations are occupied by each primitive. For each occupied pixel location, a fragment is generated. A *fragment* is a pixel location accompanied by an assigned color, depth, and texture coordinates as required. The per-fragment operations that follow primitive decomposition use the fragment's associated data to update the pixel corresponding to the fragment.

2. texturing - *Texturing* maps a portion of a specified image (called a texture) onto each primitive for which texturing is enabled. Based on the fragment's texture coordinates, the associated texture sample (or *texel* value) within the texture is combined with the fragment's color based on the texture function.

3. fog - *Fog* blends the fragment's post-texturing color with the current fog color based on the eye-coordinate distance and fog mode. The post-fog fragment is then used to update the fragment's associated frame buffer pixel.

4. per fragment operations - The frame buffer contains more than color values. Logically, there are also ancillary (or helper) buffers that hold per-pixel information in the depth, stencil, accumulation, alpha, and auxiliary buffers. And as the section on the display stage discussed, there may be multiple color buffers for double buffering (left, right) and stereo (front, back).

References

[Adel87] Edward H. Adelson, Eero Simoncelli, "Orthogonal Pyramid Transforms for Image Coding," *Proceedings of the SPIE,* Vol. 845, *Visual Communications and Image Processing II*, October 1987, pp. 50-58

[Adobe] Adobe Systems Incorporated, *PostScript Language Reference, Third Edition*, (C)1999 Addison-Wesley Publishing Company, *http://www.adobe.com/print/postscript /pdfs/PLRM.pdf*, retrieved April 24, 2000

[ATSC95a] Advanced Television Systems Committee, "ATSC Digital Television Standard," *http://www.atsc.org/Standards/A53*, retrieved April 24, 2000

[ATSC95b] Advanced Television Systems Committee, "Guide to the Use of the ATSC Digital Television Standard," *http://www.atsc.org/Standards/A54*, retrieved April 24, 2000

[Bott98] L. Bottou, P. G. Howard, Y. Bengio, "The Z-coder adaptive binary coder," *Proceedings of the DCC '98 Data Compression Conference*, Snowbird, UT, USA, March 30-April 1 1998, pp.13-22

[Cheng99] Mike Cheng, "The Modified Discrete Cosine Transform (MDCT) and MPEG Audio Encoding," *http://fox.uq.net.au/~zzmcheng/mdct/mdct.ps*, June 28, 1999, retrieved April 24, 2000

[Chiu97] Tzi-cker Chiueh, Wei-jen Lin, "Characterization of Static 3D Graphics Workloads," Proceedings of the 12th ACM SIGGRAPH/Eurographics Graphics Hardware Workshop, Los Angeles, California, August 3-4, 1997, pp. 17-24

[CNN99] Cable News Network, "Wolf Blitzer Interview with Vice President Al Gore on CNN's Late Edition," *http://www.cnn.com/ALLPOLITICS/stories/1999/03/09/president.2000/transscript.gore/*, retrieved April 24, 2000

[Cool65] J.W. Cooley, J.W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, Vol. 19, 1965, pp. 297-301

[Dege94] Jutta Degener, "Digital Speech Compression," *Dr. Dobb's Journal*, Vol. 19, No. 15, December 1994, pp. 30-89 (5)

[Dobb95] J. G. G. Dobbe, "Faster FFTs," *Dr. Dobb's Journal*, Vol. 20, No. 2, February 1995, pp. 125-151(6)

[Doom] "The Official Doom FAQ," *http://www.gamers.org/docs/FAQ/doomfaq/*, retrieved April 24, 2000

[Embr] Paul M. Embree, Bruce Kimble, *C Language Algorithms for Digital Signal Processing*, 1991, Prentice-Hall, Inc.

[Fokk95] Jeroen Fokker, "Functional Specification of JPEG Decompression, and an Implementation for Free," *Proceedings of the 5th Eurographics Workshop, Programming Paradigms in Graphics*, Maastricht, the Netherlands, September 2-3, 1995, pp. 102-120

[Foley] James D. Foley, Andires van Dam, Steven K. Feiner, John F. Hughes, Richard L. Phillips, *Introduction to Computer Graphics*, 1994, Addison-Wesley Publishing Company

[Haff98] Patrick Haffner, Leon Bottou, Paul G. Howard, Patrice Simard, Yoshua Bengio, Yann Le Cun, "Browsing through High Quality Document Images with DjVu," *Proceedings of the 1998 IEEE Advances in Digital Libraries Conference*, April 22-24, 1998, Santa Barbara, California, pp. 309-318

[Hall95] William I. Hallahan, "DECtalk Software: Text-to-Speech Technology and Implementation," *Digital Technical Journal*, Vol. 7, No. 4, 1995, pp. 5-19

[HDTel] "HDTV Questions & Answers," *http://www.cemacity.org/mall/product/video/files/qahdtv.htm*, retrieved April 24, 2000

[Herm94] Hynek Hermansky, "RASTA Processing of Speech," *IEEE Transactions on Speech and Audio Processing*, Vol. 2, No. 4, October 1994, pp. 578-589

[IDSA] Interactive Digital Software Association, "1999 State of the Industry Report," *http://www.idsa.com/IDSA_SOTI_REPORT.pdf*, retrieved September 9, 2000

[Kabl96] J. G. F. Kablau, "The DVD Physical Format," *Digest of Technical Papers IEEE International Conference on Consumer Electronics*, 1996, pp. 348-349

[Kien99] Tim Kientzle, "Implementing Fast DCTs," *Dr. Dobb's Journal*, Vol. 24, No. 3, March 1999, pp. 115-119

[Kilg95] Mark J. Kilgard, "Hardware for Accelerating OpenGL, v1.9," *The X Journal*, September/October 1995, *http://toolbox.sgi.com/TasteOfDT/documents/OpenGL/glutCol6/hwaccel.htm*, retrieved April 24, 2000

[Klatt80] Dennis H. Klatt, "Software for a cascade/parallel formant synthesizer," *Journal of the Accoustical Society of America*, Vol. 67, No. 3, March 1980, pp. 971-995

[Kodak] "Kodak Digital Image Offering," *http://www.kodak.com/digitalImages/samples /imageIntro.shtml*, retrieved April 24, 2000

[Kuni99] A. Kunimatsu, N. Ide, T. Sato, Y. Endo, H. Murakami, T. Kamei, M. Hirano, M. Oka, A. Ohba, T. Yutaka, T. Okada, M. Suzuoki, "5.5 GFLOPS Vector Units for 'Emotion Synthesis'," *Proceedings of Hot Chips 11*, Palo Alto, California, August 15-17, 1999, pp. 71-82

[Linc98] Bosse Lincoln, "An Experimental High Fidelity Perceptual Audio Coder," March 8, 1998, *http://ccrma-www.stanford.edu/~bosse/proj/proj.ps*, retrieved April 24, 2000

[Loef89] Christoph Loeffler, Adriaan Ligtenberg, George S. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications," *Proceedings of the International Conference on Acoustical, Speech, and Signal Processing, (ICASSP-89)*, Vol. 2 (of 4), Glasgow, Scotland, 1989, pp. 988-991

[Maed94] Roman E. Maeder, "Ray Tracing and Graphics Extensions," *The Mathematica Journal*, 1994, Vol. 4, No. 3, pp. 1-16, *ftp://ftp.inf.ethz.ch/doc/papers/ti/scs/ray.ps.gz*, retrieved April 24, 2000

[Mick95] Robert A. Mickelsen, "Dessert Ammonites POVray3 Dataset," *http://www.povray.org/people/ ram/datasets/ammndata.zip*, *POVzine2*, March/April 1995, retrieved April 24, 2000

[MMA] Midi Manufactures Association, "What Is MIDI?," *http://www.midi.org/abtmidi.htm*, retrieved April 24, 2000

[Naka98] Kazuhiro Nakamura, Minoru Ohta, Toshinori Odaka, Mikhail Tsinberg, "An MPEG-2 Encoder/Authoring System for DVD Title Production," *Proceedings of the 1998 17th Conference on Consumer Electronics*, Los Angeles, California, 1998, pp. 100-101

[Pan93] Davis Yen Pan, "Digital Audio Compression," *Digital Technical Journal*, Vol. 5, No. 2, Spring 1993, pp. 28-40

[Pope94] Pope Music, “Carmen Ballet for Strings and Percussion - First Intermezzo, by Rodion Konstantinovich Shchedrin (1932-Present), played by the State Symphony Orchestra 'Young Russia”, conducted by Mark Gorenstein,” *PM2002-2*, Copyright 1994 by Pope Music

[Poyn] Charles Poynton "Color FAQ," *http://www.inforamp.net/~poynton/ColorFAQ.html*, retrieved April 24, 2000

[Pres92] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes in C - The Art of Scientific Computing, Second Edition*, 1992, Cambridge University Press, *http://www.ulib.org/webRoot/Books /Numerical_Recipes/bookc.html*, retrieved April 24, 2000

[Rama98] Rajeev Raman, "Image Processing Data Flow in Digital Cameras," *Proceedings of the SPIE, Vol. 3302, Digital Solid State Cameras: Designs and Applications*, San Jose, California, January 24-30, 1998, pp. 83-89

[Rose92] M. Rosenblum, J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System,” *ACM Transactions on Computer Systems*, Vol. 10, No. 1, February, 1992, pp. 26-52

[Segal94] Mark Segal, Kurt Akeley, “The Design of the OpenGL Graphics Interface,” White Paper, 1994, *http://trant.sgi.com/opengl/docs /white_papers/design.ps*, retrieved April 24, 2000

[Shli94] Seymour Shlien, "Guide to MPEG-1 Audio Standard," *IEEE Transactions on Broadcasting*, Vol. 40, No. 4, December 1994, pp. 206-218

[Snow96] Mark Snow, “X-files Theme Song MIDI file,” *http://w3.one.net/~kklasmei/DamnGood/X_files4.mid*, August 29, 1996, retrieved April 24, 2000

[Sore87] Henrik V. Sorensen, Douglas L. Jones, Michael T. Heideman, C. Sidney Burrus, "Real-Valued Fast Fourier Transform Algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-35, No. 6, June 1987, pp. 849-863

[Swel96] W. Sweldens, “The lifting scheme: A custom-design construction of biorthogonal wavelets,” *Applied and Computational Harmonic Analysis*, Vol. 3, No. 2, April 1996. pp. 186-200

[Vaid88] P. P. Vaidyanathan, “A tutorial on multirate digital filter banks,” *Proceeding of the 1988 IEEE International Symposium on Circuits and Systems*, Espoo, Finland, June 7-9 1988, pp. 2241-2248 (Vol. 3)

[Wall91] Gregory K. Wallace, “The JPEG Still Picture Compression Standard,” *Communications of the ACM*, Vol. 34, No. 4, April 1991, pp. 30-44

[Yama97] Hisashi Yamada, "DVD Overview," *Digest of Technical Papers IEEE International Conference on Consumer Electronics*, 1996, pp. 346-347

[Yang98] Chia-Lin Yang, Barton Sano, Alvin R. Lebeck, "Instruction Level Parallelism in Geometry Processing for Three Dimensional Graphics Applications," *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, November 30 - December 2, 1998, Dallas, Texas, USA, pp. 14-24