PERFORMANCE OF AN ADVANCED VIDEO CODEC ON A GENERAL-PURPOSE PROCESSOR WITH MEDIA ISA EXTENSIONS

Ville Lappalainen
Nokia Research Center, Tampere, Finland

Abstract--**This paper analyses the performance of the state-of-the-art media ISA (Instruction Set Architecture) extensions in a general-purpose processor[1], when executing a video encoder based on an affine motion model.**

In addition to SIMD (Single Instruction Multiple Data) fixed-point instructions, these ISA extensions include SIMD floating-point instructions, special-purpose SIMD fixed-point instructions, and cacheability control instructions. In this study, eight time-consuming kernels of the video encoder were hand-optimized, using instructions in all four instruction categories of these media ISA extensions (the FLP version). These kernels were also hand-optimized using only SIMD fixed-point ISA extensions, without special-purpose instructions (the FXP version).

For the FLP version, this study resulted in an average kernel-level speedup of 1.37X and an application-level speedup of 1.11X, compared to the FXP version, and an application-level speedup of 3.41X, compared to the C version.

Index Terms--**Performance, SIMD, media ISA extensions, video coding.**

I. INTRODUCTION

IN order to meet the high computational requirements of emerging multimedia applications such as video conferencing and 3D graphics, current systems may use a combination of general-purpose processors accelerated with dedicated hardware performing spesialised computations. This kind of hardware could include DSP (Digital Signal Processing) or media processors as well as ASICs (Application Specific Integrated Circuit). However, because general-purpose processors offer several benefits, they are used increasingly for media processing applications. These benefits include ease of programming, higher performance growth as well as reduced costs, for example. Thus, SIMD-style media Instruction Set Architecture (ISA) extensions are used for most high-performance general-purpose processors, to reduce the need for specialised hardware. The ISA extensions proposed in [25], [26], and [30] operate on fixed-point data. Lately, several processor vendors have also extended their ISA with SIMD floating-point instructions [2], [28], [8], [29].

A thorough study on the performance of several image and video processing benchmarks with general-purpose processors and fixed-point media ISA extensions is presented in [27]. It also analyses the impact of alternative architectures (e.g., by varying cache sizes) and software prefetching. Additional studies, that concentrate more on specific ISA extensions, include [24], [23], and [4], for example.

3D geometry computation has traditionally been one of the most floating-point intensive applications, whereas video coding operations have typically been operating on integer data. However, as described in this paper, also advanced video coding algorithms can utilise the benefits of floating-point ISA extensions such as large dynamics and fast approximation instructions.

This study concentrates on measuring the impact of SIMD floating-point ISA extensions and special-purpose SIMD fixed-point ISA extensions on the performance of video coding operations, as they represent the most recent extensions to the state-of-the-art ISAs. Additionally, the impact of the use of media ISA extensions on the static code size and on the distribution of dynamic instructions is reported, to better characterise the benchmarking kernels.

This paper is organised as follows. Section II introduces the basic components of general video codecs. Section III introduces the differences in the basic components of advanced video codecs compared to those of general codecs. Section IV describes how video coding operations can benefit from SIMD floating-point operations, special-purpose SIMD fixed-point ISA extensions, and cacheability control instructions (e.g., prefetching instructions). Sections V and VI describe the performance evaluation methodology and the results, respectively. Section VII concludes the work.

II. GENERAL VIDEO CODECS

General video codecs, such as H.263 and MPEG-4 are basically motion compensated DCT (hybrid) codecs [14], [13].

A. Motion Compensated Prediction

Motion compensated prediction is a widely recognised technique for compression of video sequences. In this

Manuscript received June 13, 2000.

V. Lappalainen is with Nokia Research Center, P.O. Box 100, FIN-33721 Tampere, Finland (telephone: +358 3 272 5311, e-mail: ville.lappalainen@nokia.com).

[1] Intel Pentium III (Coppermine) with Streaming SIMD Extensions, running at 733 MHz.

Original manuscript received June 19, 2000

technique, one of the previously coded and transmitted frames, called reference frame $R_n(x,y)$, is used to predict the pixel values of the current, coded frame $I_n(x,y)$. The predicted frame denoted here $P_n(x,y)$ is found using (1).

$$P_n(x,y) = R_n\big(x + d_x(x,y), y + d_y(x,y)\big) \quad (1)$$

The pair $(d_x(x,y), d_y(x,y))$ is the motion vector of the pixel in the coded frame at location (x,y). Motion vectors are calculated by the motion estimation component (see Section II.C) in the encoder. The set of motion vectors of all pixels of the current frame is called the motion vector field.

The prediction error, defined in (2), is the difference between the coded frame and the prediction frame $P_n(x,y)$.

$$E(x,y) = I_n(x,y) - P_n(x,y) \quad (2)$$

The prediction error is compressed and sent to the decoder together with the motion vector field. The operating principle of motion compensated encoders is to minimise the prediction error.

To indicate that the compression of the prediction error is typically lossy, the reconstructed (coded and decoded) prediction error is denoted as $\tilde{E}_n(x,y)$. In the decoder, the reconstructed frame, defined in (3), is predicted according to (1) and then the reconstructed prediction error is added.

$$\tilde{I}_n(x,y) = R_n\big[x + d_x(x,y), y + d_y(x,y)\big] + \tilde{E}_n(x,y) \quad (3)$$

Fig. 1 illustrates the operating principle of a video encoder based on motion compensated prediction.

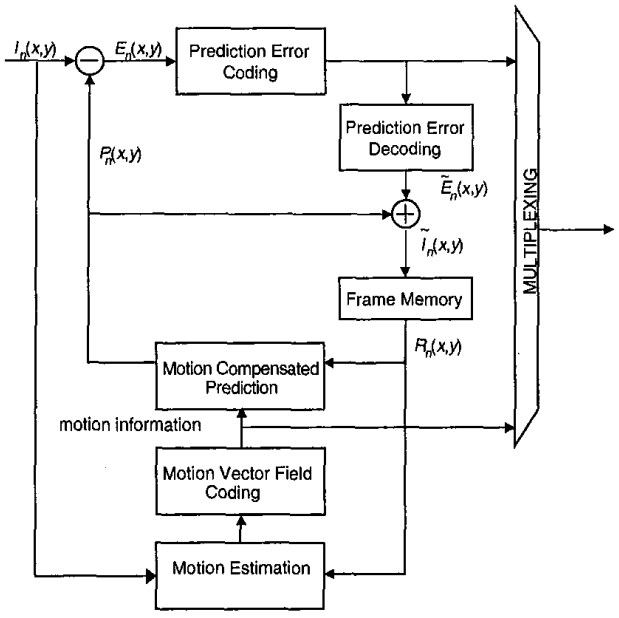

Fig. 1. A video encoder using motion compensated prediction.

B. Motion Vector Field Modeling

Frames in a typical video sequence contain a number of objects with different motion. Therefore, motion compensated prediction is usually performed by dividing the frame into several segments and estimating the motion of these segments between the current and the reference frame. If block-shaped segments are used, this estimation procedure is called block matching.

A popular way of dividing the frame into segments is to initially create 16x16 macroblocks, which can be further split in 8x8 blocks.

The motion vectors of the pixels in each segment S_k can be modeled using a parametric function:

$$\begin{aligned} d_x(\mathbf{a}_k, x, y) &= \sum_{i=1}^{m/2} a_i f_i(x,y), \\ d_y(\mathbf{a}_k, x, y) &= \sum_{i=m/2+1}^{m} a_i f_i(x,y) \end{aligned} \quad (4)$$

where only the parameters $\mathbf{a}_k = (a_1, a_2, \ldots, a_m)^T$, called motion coefficients, have to be transmitted to the decoder. The motion field basis functions f_i are known both to the encoder and the decoder.

A simple example of a motion model based on polynomial basis functions is the translational motion model (m=2, f_1=f_2=1 in (4)), which is commonly used [14], [13]:

$$d_x(\mathbf{a}_k, x, y) = a_1, \quad d_y(\mathbf{a}_k, x, y) = a_2. \quad (5)$$

This kind of motion model can describe only translations of blocks and yields a large prediction error in the presence of non-translational motion such as rotation or zoom.

C. Motion Estimation

The role of motion estimation is to calculate the motion coefficients $\mathbf{a}_k$ for a given segment S_k so as to minimise the measure of prediction error for this segment. A commonly used measure of prediction error is the Sum of Absolute Differences (SAD) defined in (6).

$$\sum_{(x,y)\in S_k} \left| I_n(x,y) - R_n(x + d_x(\mathbf{a}_k, x, y), y + d_y(\mathbf{a}_k, x, y)) \right| \quad (6)$$

The calculation of the SAD is usually a dominant operation during the motion estimation.

A popular technique used to reduce computational complexity and to provide potentially better motion vectors is to use hierarchical motion estimation [3]. Following the same principle as in motion estimation using multiresolution image pyramid [5], the reference and current frames are low-pass filtered and subsampled (smoothed) in both the horizontal and vertical direction by a factor of two, for example. In a 2-level hierarchy, motion estimation is performed first on the smoothed versions of the frames, and the result is fed to the motion estimation stage using non-smoothed frames.

D. Image Interpolation

If motion vectors can have non-integer values during the motion estimation, there will be a need to evaluate the pixel

values at non-integer locations.

Bilinear interpolation (1-D interpolation applied to 2-D data) is needed if the half-pixel precision is used during the motion estimation, as described in Fig. 2. "/" indicates division by truncation.

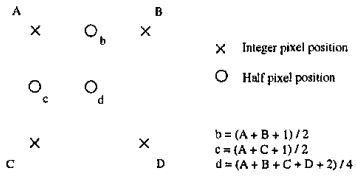

Fig. 2. Half-pixel prediction by bilinear interpolation [14].

Bilinear interpolation can also be used in hierarchical motion estimation. The typical calculation used to perform the subsampling is $Y = (A + B + C + D + 2)/4$ as illustrated in Fig. 2 above.

E. Motion Vector Field Coding

A common way to code the motion vector field is to use Variable Length (Huffman) Coding (VLC), which encodes the data as a stream of variable-length symbols based on statistical analysis of the frequency of symbols.

F. Prediction Error Coding

The Discrete Cosine Transform (DCT) is a very widely used method for the coding of the prediction error. The most usual block size is 8x8. An important issue during the implementation of the IDCT (Inverse DCT) is the accuracy. For example, an H.263 encoder should meet the accuracy requirements stated in Annex A of [14]. If an implementation of the IDCT is not accurate enough, the resulting error will cumulate to the subsequent frames.

Encoders create an encoded bit stream based on the data after the DCT and quantization. A popular way to do this is to use VLC.

G. Other Operations

1) Motion Compensation of B-frames

In MPEG-2 encoded video streams, bidirectionally predicted B-frames are used frequently. During the motion compensation of B-frames, the averaging of two pixel values is required. The accuracy of the calculations during the motion compensation is important, especially for encoders, since their local decoder should track the operation of the decoder.

2) Colour Conversion

Both encoders and decoders use colour conversion. Often encoders receive data in a format other than what they can directly encode (non-compatible chrominance space, interleaved vs. planar data, etc.). Decoders sometimes have to write the decoded picture to the memory of a graphics card in a colour space (e.g., RGB) other than the colour space that naturally is produced from the decode (e.g., YUV); this also requires a colour conversion.

3) Block Edge Filtering

One technique to reduce blocking artifacts in a coded frame is to use a block edge filter, as in one of the optional coding modes of H.263 (Deblocking Filter mode (Annex J)). This specific filter operates on the picture that is used for the prediction of subsequent pictures and thus lies within the motion estimation loop. The filtering is performed on 8x8 block edges.

III. Advanced Video Codecs

Advanced codecs, such as MVC [15], [18], H.263++, and H.26L, offer improved coding efficiency compared to general codecs presented in the previous section. However, their computational complexity is significantly higher than that of general codecs.

A. Motion Compensated Prediction

Motion compensated prediction is widely used in both advanced and general codecs.

B. Motion Vector Field Modeling

An advanced example of a motion model based on polynomial basis functions is the affine motion model (m=6, f_1=f_4=1, f_2=f_5=x, f_3=f_6=y in (4)), in which the values of motion vectors are given by (7).

$$\begin{aligned} d_x(\mathbf{a}_k, x, y) &= a_1 + a_2 x + a_3 y, \\ d_y(\mathbf{a}_k, x, y) &= a_4 + a_5 x + a_6 y \end{aligned} \tag{7}$$

The affine motion model is capable of describing rotation, change of scale and translation at the same time [32]. Thus, the affine motion model gives a more realistic approximation of the changes taking place in image sequences compared to the translational model.

The affine motion model has been studied in the ITU-T (International Telecommunication Union, Telecommunication Standardisation Sector) study groups focusing on the development of the next version of the H.263 standard (H.263++) as well as the H.263L standard [17], [15].

C. Motion Estimation

In addition to the SAD, another commonly used measure of prediction error is the Sum of Square Differences (SSD) defined in (8).

$$\sum_{(x,y)\in S_k} (I_n(x,y) - R_n(x + d_x(\mathbf{a}_k, x, y), y + d_y(\mathbf{a}_k, x, y)))^2 \tag{8}$$

Equation (8) is a multidimensional non-linear function. There are no techniques that always find its absolute minimum and have an acceptable computational complexity. So-called differential algorithms such as Gauss-Newton are usually used to minimise such functions [7].

The Gauss-Newton (GN) algorithm assumes that the function to be minimised can be locally approximated by a quadratic function of the parameters. Due to this assumption it can converge only towards local minima, unless the initial parameters lie in the attraction domain of the global minimum. Thus, it is necessary to feed this algorithm with a sufficiently good initial guess of the actual optimum.

Block matching is one of the methods that can be used to

feed the Gauss-Newton algorithm with an initial guess.

Another technique that can be used to improve the convergence of the Gauss-Newton algorithm is 2-level hierarchical motion estimation (see Section II.C).

The convergence can also be improved by repeating motion estimation and segmentation (Split Decision in Fig. 4 below) recursively, until the minimum size of the block (e.g., 8x8) is reached or the prediction error is below a certain threshold.

Fig. 3 shows how the above-mentioned methods are combined into an advanced motion estimation scheme.

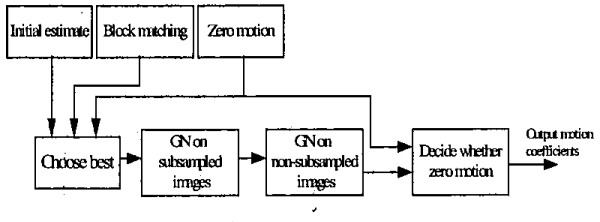

Fig. 3. Block diagram of an advanced motion estimation scheme [15].

D. Image Interpolation

One interpolation technique that works well with the above mentioned motion estimation scheme is called cubic convolution interpolation [22]. The Gauss-Newton method requires first derivatives of the reference frame at non-integer coordinates, and it works much better if the derivatives are continuous. Thus, bilinear interpolation cannot be used. See [21] for details about selecting the interpolation technique.

E. Motion Vector Field Coding

Since the complexity of motion usually varies between frames and between segments, the complexity of motion model should also vary to accurately model the motion. Thus, the affine motion model can be adapted so that some less significant motion coefficients of a segment are removed.

An adaptive motion vector field coding system together with the motion estimation scheme presented above is shown in Fig. 4.

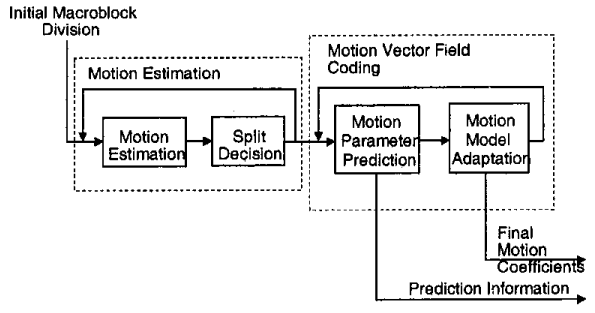

Fig. 4. Block diagram of motion estimation and motion vector field coding.

The goal of Motion Vector Field Coding is to find, for each segment, the motion model that minimises the distortion subject to the total bit budget constraint. The distortion D is defined as the square error between the original and the coded frame and the rate T is equal to the number of bits spent on coding this frame, i.e., on coding the motion coefficients and the prediction error. This optimization problem is equivalent to minimising the Lagrangian cost function defined in (9).

$$L = D + \lambda T \tag{9}$$

The parameter λ is the rate-distortion trade-off parameter supplied from higher level controls such as the rate control mechanism.

In order to find the optimal solution to this cost function, a computationally expensive exhaustive search is required. However, approximate solutions are described in the following.

1) Matrix Representation of the Motion Vector Field

Encoding the motion vector field will require obtaining motion coefficients for each segment in a frame with different combinations of motion models. It would be very computationally expensive to minimise each time the prediction error function using iterative, differential techniques. Therefore, a new representation, defined in (10), of the motion vector field is calculated at the beginning of motion vector field coding. It can be used later to calculate motion coefficients more efficiently.

The prediction error function, defined in (8), can be approximated as

$$\left(\mathbf{E}_k\mathbf{a}_k - \mathbf{v}_k\right)^T\left(\mathbf{E}_k\mathbf{a}_k - \mathbf{v}_k\right) \tag{10}$$

Let us order all the P pels belonging to the segment S_k and denote their co-ordinates as (x_i, y_i). The ith row of matrix $\mathbf{E}_k$ is then given by

$$\left[\frac{\partial R_n(x'_i, y'_i)}{\partial x} \quad \frac{\partial R_n(x'_i, y'_i)}{\partial y}\right] \otimes \begin{bmatrix}1 & x_i & y_i\end{bmatrix},$$

and the ith element of vector $\mathbf{v}_k$ is equal to

$$I_n(x_i, y_i) + \frac{\partial R_n(x_i', y_i')}{\partial x} d_x^k\left(\mathbf{a}_k^c, x_i, y_i\right) + \frac{\partial R_n(x_i', y_i')}{\partial y} d_y^k\left(\mathbf{a}_k^c, x_i, y_i\right) - R_n(x_i', y_i')$$

where $\mathbf{A} \otimes \mathbf{B}$ denotes the Kronecker product of the two matrices. The approximation (10) is easier to minimise than the original prediction error function (8) (see [15] for the details). This minimisation involves calculation of partial derivatives and solving a set of equations using the method of normal equations (which includes, e.g., building a least squares system, Cholesky factorization of a matrix, and solving of matrix equations by back substitution, see Section V.A).

2) Motion Model Adaptation

The goal of Motion Model Adaptation is to find, for each segment, the subset of the affine model basis functions and their corresponding motion coefficients as to minimise the Lagrangian cost function of this segment. In this phase, also INTRA and NOT CODED coding modes (in addition to INTER) for the segment could be considered. See [15] for further details.

The Motion Model Adaptation can be carried out for each segment independently.

For an efficient implementation of Motion Model Adaptation, it is essential to apply a low complexity method to 1) approximate the increase of the prediction error and 2) re-estimate motion coefficients when one of the basis functions is removed. The details of implementation of this kind of method are described in [20]. This implementation requires the calculation of the Givens rotation [9], for example.

3) Motion Parameter Prediction

Motion coefficients can be predicted from the block above or left of the block being currently coded, for example.

F. Prediction Error Coding

Since the prediction error has usually a statistically varying nature, it is very difficult for a single coding method (e.g., 8x8 DCT) to represent it. This has been taken into account when preparing the proposals for the future ITU video coding standard called H.26L. Telenor's proposal suggests the use of 7 different block sizes (16x16, 16x8, 8x16, 8x8, 8x4, 4x8, 4x4) for the DCT-coding of the prediction error in motion compensated prediction [16].

In addition to multi-shape DCT, diagonal KLT (Karhunen-Loeve Transform) has been found efficient for the coding of prediction error [15].

IV. Video Coding with Media ISA Extensions

Media ISA extensions commonly found in general-purpose processors can be divided into four categories: SIMD fixed-point instructions, SIMD floating-point instructions, special-purpose SIMD fixed-point instructions, and cacheability control instructions. Table I shows an example of media ISA extensions, excluding (general-purpose) SIMD fixed-point instructions.

SIMD floating-point instructions usually operate on a set of 128-bit or 64-bit registers, which are used to store the corresponding packed data type, namely four or two 32-bit single-precision numbers, respectively. These instructions operate on either all, or the least significant pairs, of packed data operands in parallel.

This section describes how several video coding and image processing operations can benefit from media ISA extensions, excluding SIMD fixed-point instructions. The use of SIMD fixed-point instructions in video coding is studied in [24], [4], and [27], for example.

TABLE I
AN EXAMPLE OF MEDIA ISA EXTENSIONS [28]

Category	Instructions	Packed SP FP	Scalar SP FP	Packed integer
Arithmetic	ADD, SUB, MUL, DIV, MAX, MIN,	X	X	
	SQRT, RCP, RSQRT	X	X	
Logical	AND, ANDN, OR, XOR	X		
Comparison	CMP	X	X	
	COMI, UCOMI		X	
Data movement	MOVAPS (load/store aligned)	X		
	MOVUPS (load/store unaligned)	X		
	MOVLPS, MOVLHPS, MOVHPS,	X		
	MOVHLPS	X		
	MOVMSKPS	X		
	MOVSS (load/store)		X	
Shuffle	SHUFPS, UNPCKHPS, UNPCKLPS	X		
Conversions	CVTSS2SI, CVTTSS2SI, CVTSI2SS		X	
	CVTPI2PS, CVTPS2PI, CVTTPS2PI	X		
State management	FXSAVE, FXSTOR, STMXCSR,	X		
	LDMXCSR	X		
Special-purpose (fixed-point)	PINSRW, PEXTRW, PMULHU,			X
	PSHUFW, PMOVMSKB, PSAD,			X
	PAVG, PMIN, PMAX			X
Cacheability control	MASKMOVQ,			X
	MOVNTQ (aligned store)			X
	MOVTPS (aligned store)	X		
	PREFETCH			
	SFENCE			

A. SIMD Floating-Point Instructions

One way to implement SIMD instructions operating on 128-bit packed data, is to treat each 4-wide computational macro-instruction as two 64-bit microinstructions. However, in a superscalar processor (e.g., two execution ports), a full 4-wide SIMD operation can also be done every clock cycle, assuming instructions alternate between two asymmetric execution ports (e.g., add-multiply-add-multiply). With this approach, 128-bit SIMD calculations can theoretically achieve a full 4X performance gain; 2X is a more realistic gain in practice, partly because of micro-instruction pressure in the microarchitecture. However, this emulation of 4-wide SIMD calculations is not an efficient implementation for scalar operations. Thus, there are usually explicit scalar instructions, which can execute only a single micro-instruction [28].

While traditional video coding operations usually do not include frequent floating-point calculations, 3D applications calculate using floating-point numbers extensively. It is reported that SIMD floating-point ISA extensions can boost the performance on 3D kernels (transformation and lighting) over 2X (1.4X to 2.75X) that of optimized scalar code [33].

This study reports a 1.42X kernel-level speedup for a routine that uses arithmetic SIMD floating-point instructions, operating on packed data, see Section VI.

A considerable speedup can also be achieved using the approximation instructions: *RCP* (*Reciprocal*) and *RSQRT* (*Reciprocal of the SQRT*). These instructions are less precise (e.g., 50% less bits in mantissa) but much faster (e.g., 9X/15X in terms of latency) than the DIV/SQRT instructions. A greater precision is also available (e.g., 92% that of DIV/SQRT), when using the approximation instructions with the Newton-Raphson (NR) method [11].

For the reciprocal operation, the NR method involves two multiplies and a subtraction. Thus, the overall latency and especially the throughput for the NR method are lower than for DIV/SQRT, which usually have a poor throughput [12].

It should be noted that these approximation instructions could also make scalar operations much faster than DIV/SQRT. This study reports kernel speedups of 1.80X and 1.72X for a combined use of the RCP and RSQRT instructions (with the Newton-Raphson method) operating on scalar data. Additionally, a kernel speedup of 1.26X for the use of the sole RCP instruction operating on packed and scalar data is reported. See Section VI for the details.

B. Special-purpose SIMD Fixed-Point Instructions

Video coding operations can benefit significantly from several instructions in this category. These instructions operate typically on 64-bit packed integer data.

Every video encoder, which uses the SAD as an error measure, can benefit from the *Packed SAD* (*PSAD*) instruction, which calculates the SAD for, e.g., 8 pixels at a time. This instruction has been found to increase the performance of the motion estimation in an MPEG-1/MPEG-2 encoder by a factor of two [28].

This study reports a 1.56X kernel-level speedup for a routine that utilises this instruction, see Section VI.

The B-frame motion compensation component in an MPEG-2 video decoder can benefit from the *Packed Average* (*PAVG*) instruction. This instruction performs the averaging operation (using 8-bit or 16-bit accuracy, for example) by rounding the result to the nearest integer as required by the MPEG-2 specification. The use of this instruction could enable a 25% kernel speedup and a 4 to 6% application level speedup [28].

Subsampling in the hierarchical motion estimation as well as in the motion estimation with half-pixel precision are sped up using the PAVG instruction. While one PAVG instruction performs 2-value averaging, it is possible to use three PAVG instructions to approximate 4-value averaging with an accuracy of 87.5% as shown in the following pseudo-code [1]:

$$Y = pavg(pavg(A,B), pavg(C,D) - 1).$$

The *Packed Shuffle* (*PSHUFW*) instruction can be used to rearrange the data within a register. It can perform rotate, shift, swap, and broadcast operations on 16-bit data, for example. Without this instruction, several instructions are needed to perform these kind of operations (e.g., broadcast required three instructions).

This study reports speedups of 1.05X to 1.06X for DCT routines operating on different block sizes while utilising PSHUFW as the only special-purpose fixed-point instruction.

The *Packed Minimum* (*PMIN*) and *Packed Maximum* (*PMAX*) instructions can be used to clip the data values into the desired range such as [0,255]. Note that the clipping can be performed without conditional branching, which could be unpredictable. Block edge filtering specified in the Deblocking Filter mode (Annex J) of the H.263 standard benefits from these instructions, because clipping is a dominant operation during the filtering operation.

During the VLC, especially in the case of B-frames, there are often many zero values that must be detected and omitted. To simplify this kind of data-dependent branching, the *Move Byte Mask to Integer* (*PMOVMSKB*) instruction can be used to evaluate eight values as shown in the following pseudo-code [1]:

```
pxor      mm7, mm7    // zero mm7
movq      mm0, [esi]  // get eight Q values
pcmpeqb   mm0, mm7    // find zeros
pmovmskb  eax, mm0    // 8 flags into eax
```

If eax holds 0xff, then all eight values are zero.

C. Cacheability Control Instructions

These instructions usually include prefetching and streaming store instructions. In the following, an example implementation of both the prefetching and streaming store instructions (in a 2-level cache hierarchy) is briefly described.

The prefetch instructions are used to load data ahead of use, thereby hiding load latency so that the CPU can take full advantage of memory bandwidth. If the processor loads data to cache when a cache line is written to, prefetches can also reduce latency for storing data. To get the most efficient use of prefetch, loops should be unrolled so that each iteration prefetches and processes the same amount of data (e.g., one cache line).

The prefetch instructions are usually most useful for memory bound applications, whose working set of data is non-temporal (i.e., read and used once before being discarded) and does not fit into the cache. The *Non-temporal Prefetch* (*PREFETCHNTA*) instruction fetches data only into the L1 cache (closer to the processor than the L2 cache), thus not filling the L2 cache with non-temporal data. The *Temporal Prefetch0* (*PREFETCHT0*) fetches data into both the L1 and L2 caches while the *Temporal Prefetch1* (*PREFETCHT1*) fetches data only into the L2 cache. PREFETCHNTA avoids some overhead incurred when data are also loaded to the L2 cache (as usually occurs with normal load and store).

For encoders, colour conversion is typically a memory-bound operation. It loads picture data from main memory, performs some (typically simple) calculation, and writes the data back out to memory. The PREFETCHNTA instruction can speed up colour conversion by bypassing the L2 cache on the load. This prefetch is often the best prefetch for colour conversion since the input will not be needed again by the codec. The store can then be performed using a normal store instruction so that the picture resides in the L2 cache after the colour conversion.

This study reports a speedup of 1.12X for a routine utilising the PREFETCHNTA instruction, see Section VI.

PREFETCHT1 could be used to load the L2 cache with a data set larger than can fit in the L1 cache. Meanwhile a CPU speed-limited algorithm could be executing and

randomly accessing data. As it proceeds, it would find more and more of its data in the L2 cache.

Motion compensation in video decoders is often a memory bound operation. PREFETCHNTA and PREFETCHT0 have both been observed to provide a speedup. Which one offers the best improvement is dependent on how the decoder is implemented. For decoders that are writing the decoded picture to a graphics card memory, streaming store instructions can offer a benefit by not polluting the caches with non-temporal data.

The streaming store instructions can be used to write results to the destination memory buffer without going through the cache hierarchy. The programmer should consider the possible write combining of the processor, when using these instructions, because any access to memory or the L2 cache can cause premature flushing of the write-combining buffers. This results in inefficient use of the memory bus. Thus, while writing out results with the streaming store instructions, data should only be read from the L1 cache, to avoid this performance penalty [12].

V. Performance Evaluation Methodology

A. Workloads

The encoder described in Section III is used as a benchmark application and some of its most time-consuming routines as kernels. Both the application and kernels have been implemented in three versions: C, FXP (coded with fixed-point ISA extensions [26]), and FLP (coded with floating-point ISA extensions [28]). Only the FLP version can use the approximation instructions (Reciprocal and Reciprocal of the SQRT), the special-purpose fixed-point instructions (e.g., Packed SAD and Packed SHUFFLE) and the prefetching instructions. Most of the kernels represent typical, generally used video coding/image processing operations such as DCT, SAD calculation, and FIR filtering.

A freely available software library [10] offers a variety of image processing and DSP routines that are optimized in assembly language. However, they are made for general use and thus, the performance is not necessary the same as that of the hand-coded routines optimized for a specific purpose. See [4] for the evaluation of some of the routines in this library.

All kernels in the FXP and FLP versions are hand-optimized either in assembly language (using inline assembly) or using intrinsics [31], if not otherwise mentioned. The kernels are described in the following.

1) Subsampling Filter

This routine applies an 8-tap 1-D FIR filter in both horizontal and vertical directions on an input image. While filtering, the routine also performs subsampling by a factor of two (by skipping every second pixel) in both directions as well. It uses fixed-point numbers having precision of 14 fractional bits (the filter coefficients).

The FLP version differs from the FXP version by utilising two PREFETCHNTA instructions in an unrolled loop; one in the beginning (prefetches input data 32 bytes ahead of the current iteration), another in the middle of the loop (prefetches 64 bytes ahead). One iteration processes 38 bytes of the input image. It was not trivial to process the optimal amount, one cache line (32 bytes), of data in one iteration. The non-temporal prefetch is the best choice since the input image is not needed after the filtering (in the near future). The optimal prefetch distance of 32 bytes was experimentally obtained.

There is no difference in calculation resolution between the three versions; all obtain exactly the same results.

2) Build Least Squares Equation

The Motion Vector Field Coding algorithm presented in Section III.E obtains the parameters for its affine motion model by minimising the mean square error (MSE) between a block in a reference frame and a block in a current frame. This involves building and solving a least-squares system of equations (of form $\mathbf{C}_k\mathbf{a}_k = \mathbf{d}_k$)($k$=6). This kernel builds the least-squares matrix ($\mathbf{C}_k = \mathbf{E}_k^T\mathbf{E}_k$) for each pixel in a (8x8) block, for which the motion parameters are being estimated. The matrices are then summed together to obtain the complete system matrix for a block. This system is subsequently solved using back substitution, as described below.

The kernel utilises floating-point arithmetic most of the time to avoid round-off error accumulation, which is usually a problem when the input data have a large dynamic range. Due to this extensive use of floating-point arithmetic, fixed-point ISA extensions cannot be widely used in the FXP version. Thus, the FXP version is coded using mainly scalar assembly language. The FLP version takes advantage of packed addition and multiplication instructions, which are very well suited for calculating matrix sums and products.

3) Cholesky Factorization

This kernel performs Cholesky factorization of a (6x6) input matrix ($\mathbf{C}_k$), i.e., calculates the upper triangular output matrix ($\mathbf{U}_k$) such that $\mathbf{U}_k^T\mathbf{U}_k = \mathbf{C}_k$ (k=6) [9]. This involves computations of the reciprocal of the square root following by the reciprocal.

The FXP version is coded using C and the FLP version is coded using intrinsics. In this case, no essential gain is expected for using assembly.

The C version uses 64-bit floating point numbers. The FXP version uses 32-bit floating point numbers internally, but the input and output values use the 64-bit precision. The FLP version operates on scalar data and uses the approximation instructions and the Newton-Raphson (NR) method. The precision of these calculations is effectively one bit less than those of the FXP version.

4) Back Substitution

This routine obtains $\mathbf{a}_k = (a_1, a_2, \ldots, a_m)^T$ by solving the following equation by back substitution: $\mathbf{U}_k\mathbf{a}_k = \mathbf{z}_k$ (m=6) [9]. This involves several computations: divisions, additions, multiplications, and subtractions.

Again, the FXP version is coded using C and the FLP version is coded using intrinsics.

The C version uses 64-bit floating-point numbers. The

FXP version uses 32-bit floating-point numbers when calculating the reciprocals. The FLP version operates on packed and scalar data and uses the approximation instructions.

5) Givens Rotation

The Givens rotation can be used to triangularize a matrix by zeroing matrix elements in a row one at a time. This routine performs this rotation, which involves, e.g., the following calculations: addition, multiplication, and division followed by the reciprocal of the square root [9].

The calculation resolutions for the C and FXP versions are identical to those of the Cholesky Factorization routine as well as the coding methodologies for the FXP and FLP versions (C and intrinsics, respectively).

The FLP version obtains its speedup by using the RCP instruction (with the NR method) and the RSQRT instruction (without the NR method), operating on scalar data.

6) Block Matching

Block matching is used to find the initial motion vector for a (4x4) block. This motion vector is then fed to the affine motion search process. Block matching is performed on a subsampled image. Block matching starts with the evaluation of the zero motion vector, after which the full-search is performed. Partial error check is performed after every eight pixels during the error computation. The range of motion vector components is limited: from -15 to 15.

Error is evaluated as a Sum of Absolute Differences (SAD). The FLP version utilises the Packed SAD instruction, while the FXP version must replace that instruction by several other instructions.

7) 8x8 and 4x8 DCT

Separable, direct matrix multiply 2-D DCT with factorization optimizations is used. Faster DCT algorithms cannot be used, since high accuracy is required. Additionally, fast scalar algorithms require lots of data reorganising within registers when packed instructions are used.

The FLP and FXP versions are almost identical except that the FLP version utilises Packed SHUFFLE instruction for reversing the order of input data. This operation requires four FXP instructions but only a single FLP instruction.

B. Experimentation Environment

The hardware environment consisted of a PC based on a 733 MHz processor[2] with 256 MB of memory. The size of the on-chip L2 cache was 256 KB. The bus speed was 133 MHz.

The benchmark application was run in a commercially available operating system[3]. It was compiled using two compilers: one[4] for compiling all the files containing FLP code, the other[5] for compiling rest of the files (compiler optimizations for both compilers were targeted at maximising speed). During the measurements there was a minimum computational load caused by other programs. In addition, the priority of the encoder process was set to the maximum value. This was done by calling the following functions provided by the operating system: SetPriorityClass and SetThreadPriority with parameters REALTIME_PRIORITY_CLASS and THREAD_PRIORITY_TIME_CRITICAL, respectively. To obtain some more detailed kernel-level results, a commercially available performance analyser (profiling tool)[6] was used.

Three original, uncompressed QCIF-sized (luminance resolution: 176x144) sequences, *Akiyo*, *Mother and Daughter*, and *Carphone*, were used. These sequences were selected from a set of standard test sequences that were used during the development of video coding standards ITU-T Recommendation H.263 [14] and MPEG-4 [13].

C. Performance Metrics

Two different performance comparison tests were executed. During the first test, all test sequences were encoded with a set of bit rates (8, 14, and 24 kbps) using all three versions. First 300 frames of all sequences were encoded with target and reference frame rates of 8.33 (25/3) fps and 25 fps, respectively. The rate control of the encoder resulted in a constant target frame rate, i.e., the total amount of encoded frames was constant. To make the comparisons as fair as possible, the different implementations always encoded exactly the same frames (every third frame of the original sequence). The encoding speed (in frames/s) was measured by using the ANSI C clock function.

Due to the different nature of test sequences, a unique set of target bit rates was selected for each sequence. The GSM HSCSD network offers 14.4 kbps and 28.8 kbps channels, for example. In a typical video phone application, 8 kbps is reserved for video in case of 14.4 kbps, and 22-24 kbps in case of 28.8 kbps [6].

This test demonstrated the application-level speedup of the FLP version. Differences in video quality between the three versions were also measured, in terms of luminance PSNR (Peak Signal to Noise Ratio).

During the second test, *Carphone* was encoded with a target bit rate of 24 kbps using the FXP and FLP versions. The average (total execution time divided by execution frequency), minimum, and total (cumulative) execution times of the optimized routines were measured. This test demonstrated the speedup of each individual routine. The execution time of each optimized routine (in processor clock cycles) was measured using a special instruction (Read from Time Stamp Counter), which returns clock cycles (more accurate than the ANSI C clock function).

This study also reports the instruction level characteristics of the kernels. The number of static assembly instructions (static code size) as well as the percentage of dynamic FLP-specific (packed and scalar), FXP-specific (includes special-purpose instructions), and other assembly instructions were measured. In this context, a dynamic instruction is an

[2] Intel Pentium III (Coppermine) with the Intel 82820 AGPset chip set.

[3] Microsoft Windows NT 4.0 (build 1381) with Service Pack 5 installed.

[4] Intel C Compiler 4.0.

[5] Microsoft Visual C++ 5.0.

[6] Intel Vtune Performance Analyzer 4.0.

instruction, which is retired (committed) during the actual run-time of the program. Because the processor used in this study provides speculative execution, the number of retired instructions could be smaller than that of executed instructions. These values were measured with the performance analyser mentioned above.

In order to reduce the influence of, e.g., disk caching on the execution time, the same sequence was encoded three times in a row. The average values and standard deviation were calculated. If the standard deviation exceeded a certain threshold for a specific case, one or more new executions were performed, until the standard deviation was below the threshold. In the first test, the harmonic mean of the encoding speed was used instead of the arithmetic mean used in the second test. The harmonic mean of the encoding speed was chosen, because the arithmetic mean can be justified for the reciprocal of the encoding speed (measured in frames/s) [19].

VI. Results

A. Kernel-level Analysis

Table II shows the number of static assembly instructions as well as the percentage of dynamic FLP-specific, FXP-specific (includes special-purpose instructions), and other assembly instructions (includes prefetching instructions).

TABLE II
STATIC AND DYNAMIC INSTRUCTIONS

Routine	Version	Static instr.	Dynamic FLP instr. (in %)	Dynamic FXP instr. (in %)	Dynamic other instr. (in %)
Build LS Equation	FLP	1660	96.98	1.37	1.65
	FXP	761	0.00	17.68	82.32
Block Matching	FLP	770	0.00	74.12	25.88
	FXP	965	0.00	78.97	21.03
DCT 8x8	FLP	178	0.00	81.37	18.63
	FXP	193	0.00	81.15	18.85
Cholesky Factoriz.	FLP	521	96.66	0.00	3.34
	FXP	485	0.00	0.00	100.00
Givens Rotation	FLP	62	11.76	0.00	88.24
	FXP	49	0.00	0.00	100.00
Back Substitution	FLP	184	70.83	0.00	29.17
	FXP	206	0.00	0.00	100.00
Subsampling Filt.	FLP	720	0.00	66.67	33.33
	FXP	718	0.00	67.81	32.19
DCT 4x8	FLP	124	0.00	74.01	25.99
	FXP	136	0.00	78.19	21.81

Note that in Table II, the column that shows the percentage of dynamic FXP instructions does include also the special-purpose SIMD fixed-point instructions, which are available only in the FLP versions of the kernels. By using the performance analyser mentioned in the previous section, it is not possible to separate the special-purpose SIMD fixed-point instructions from other (general-purpose) SIMD fixed-point instructions. The FLP versions of the following kernels utilise none of the FLP instructions but special-purpose SIMD fixed-point instructions: Block Matching, DCT 4x8, and DCT 8x8.

The FLP version provides reductions in the static code size for the following kernels: Block Matching (20%), DCT 8x8 (8%), DCT 4x8 (9%), and Back Substitution (11%). The reductions are due to the use of special-purpose instructions (PSAD and PSHUFW) and packed SIMD calculations (in Back Substitution). The static code size is increased for the rest of the kernels (the increase is insignificant for Subsampling Filter): Build Least Squares Equation (118%), Cholesky Factorization (7%), and Givens Rotation (27%). The increases are due to the use of the Newton-Raphson method (Cholesky Factorization and Givens Rotation) and extensive loop unrolling (Build Least Squares Equation).

Table III shows the average, minimum and total execution times, as well as the average speedup for each kernel, when encoding *Carphone* (in QCIF) at 8.33 fps and 24 kbps. The average and minimum execution times are reported for one execution of each kernel. Two subsequent rows contain information on each kernel, the upper reporting the FLP version, the lower the FXP version. The kernels are sorted by total execution time (of the FXP version, in descending order), i.e., the most time-consuming kernels of the encoder are listed first.

TABLE III
AVERAGE, MINIMUM AND TOTAL EXECUTION TIMES, AND AVERAGE SPEEDUP FOR EACH KERNEL

Routine	Version	Average time (in cycles)	Min. time (in cycles)	Total time (in millions of cycles)	Average speedup vs. FXP
Build LS Eq.	FLP	4837	1930	1430.88	1.42
	FXP	6869	4067	2063.36	-
Block Matching	FLP	4051	73	127.73	1.56
	FXP	6330	78	201.56	-
Cholesky Factoriz.	FLP	601	460	72.88	1.72
	FXP	1031	833	127.29	-
DCT 8x8	FLP	445	400	119.81	1.05
	FXP	467	425	127.00	-
Givens Rotation	FLP	108	64	67.29	1.80
	FXP	195	97	123.37	-
Back Substit.	FLP	236	71	63.92	1.26
	FXP	296	88	81.83	-
Subsampling Filter	FLP	367701	353836	72.80	1.12
	FXP	412854	380293	81.75	-
DCT 4x8	FLP	270	212	29.30	1.06
	FXP	287	230	31.97	-

The average speedup on kernels ranges from 1.05 to 1.80. The average kernel-level speedup is 1.37X (arithmetic mean). The speedup based on weighted means (execution frequencies as weighting factors) is 1.49X. This shows that in the case of the benchmark application used in this study, the kernels having better speedup than the average speedup (1.37X) are executed more frequently than those having a lower speedup.

The share of the total execution time for these kernels is 27% for the FLP version and 35% for the FXP version.

The routines using SIMD floating-point calculation (Build LS Equation and Back Substitution) have speedups close to the average speedup: 1.42X and 1.26X, respectively. However, routines which use the floating-point approximation (RCP and RSQRT) and special-purpose fixed-point instructions (PSAD) achieve the highest

speedups. They operate on both packed (Block Matching, speedup: 1.56X) and scalar data (Givens Rotation, speedup: 1.80X). If Block Matching used larger block size than the current version (4x4), the speedup would be higher because of better utilisation of the PSAD instruction. Cholesky Factorization cannot utilise the approximation instructions as effectively as Givens Rotation. Thus, the speedup, 1.72X, is a bit lower. Back Substitution operates on packed data and it does not calculate any square roots but reciprocals. Thus, the speedup (1.26X) is lower than what could be achieved when calculating square roots with the approximation instructions.

The DCT routines cannot use special-purpose instructions extensively; they use only PSHUFW. Thus, the speedups are low, 1.06X and 1.05X for DCT 4x8 and DCT 8x8, respectively. The low speedup of Subsampling Filter (1.12X) is because of the small working set. For QCIF-sized frames, 65% of the luminance data fits into the L1 cache (16K). Thus, prefetching does not help as much as for CIF-sized frames, for example.

The reason for almost similar minimum execution times for both versions of the Block Matching kernel is the use of branches for some special cases. When these branches are taken, the function makes an early exit without executing all of the optimized code.

B. *Application-level Analysis*

Table 2 shows the encoding speeds and the average values (harmonic mean) of all eight cases for each version, when encoding the QCIF-sequences at 8.33 fps.

TABLE IV
ENCODING SPEEDS (IN FRAMES/S)

Bit rate	Version	Akiyo	Carphone	Mother&Daughter	Avg.
8 kbps	FLP	21.16		18.85	14.73
	FXP	18.79		16.75	13.22
	C	5.82		5.42	4.32
14 kbps	FLP	17.99	12.84	13.09	
	FXP	16.47	11.36	11.96	
	C	5.32	3.50	4.02	
24 kbps	FLP	18.97	10.66	11.55	
	FXP	17.49	9.42	10.37	
	C	6.00	3.03	3.61	

The average application-level speedup of the FLP version is 1.11X, compared to the FXP version, and 3.41X, compared to the C version. The ranges are from 1.08X to 1.13X and from 3.17X to 3.65X, respectively.

Table V shows the PSNR (Peak Signal to Noise Ratio) values of the C version.

TABLE V
LUMINANCE (Y) PSNR VALUES (IN DB) OF THE C VERSION

Bit rate	Akiyo	Carphone	Mother&Daughter
8 kbps	35.36		31.48
14 kbps	38.28	30.81	33.44
24 kbps	41.15	32.58	35.54

On the average, the losses in the PSNR, compared to the C version, are 0.01 dB (FLP) and 0.001 dB (FXP). The loss ranges from –0.05 to 0.02 dB (FLP). In practice, this means that no loss in subjective quality could be observed.

VII. CONCLUSIONS

SIMD floating-point and special-purpose SIMD fixed-point ISA extensions can be used to speed up common video coding kernels as well as the affine motion model related kernels (matrix floating-point computations). These common kernels include motion estimation (SAD calculation), motion compensation, image interpolation (e.g., bilinear interpolation), variable length encoding, colour conversion, and block edge filtering.

Abel et al. report speedups of 2X for motion estimation, and application-level speedups of 1.3X when utilising special-purpose fixed-point ISA extensions [1].

In this study, eight time-consuming kernels of the video encoder were hand-optimized using SIMD fixed– and floating-point and special-purpose SIMD fixed-point ISA extensions as well as cacheability control instructions (the FLP version). Another version of the encoder was optimized using only SIMD fixed-point ISA extensions (without special-purpose instructions) (the FXP version).

For the FLP version, this study resulted in an average kernel-level speedup of 1.49X (weighted arithmetic mean) and an application-level speedup of 1.11X, when comparing to the FXP version, and an application-level speedup of 3.41X, when comparing to the C version.

In this study, the kernels optimized in the FLP version do not consume a large fraction of the execution time (about 27%), because the encoder has already been extensively optimized for the FXP version. Thus, the application-level speedup is limited by that fraction and the kernel-level speedup of 1.49X corresponds to the application-level speedup of 1.1X (Amdahl's Law). Applications, which are able to exploit the prefetching instructions more frequently, could achieve higher speedups than those reported in this study.

The processor used in this study shares a number of fundamental similarities with the floating-point media ISA extensions proposed for other processors [2], [8], [29], and is representative of the media ISA extensions used in modern general-purpose processors.

VIII. ACKNOWLEDGEMENTS

The author would like to thank Roberto Castagno, Pawel Defee, Petri Haavisto, Ilkka Haikala, Antti Hallapuro, Petri Liuha, Marko Luomi, Hannu Nieminen, and Joni Vahteri for useful discussions and comments.

IX. REFERENCES

[1] J. Abel, K. Balasubramanian, M. Bargeron, T. Craver, and M. Phlipot, "Applications Tuning for Streaming SIMD Extensions," *Intel Technology Journal*, Q2, 1999.

[2] "Enhanced 3DNow! Technology for the AMD Athlon Processor," via http://www.amd.com/products/cpg/athlon/3dnow_wp.html, Dec. 1999.

[3] B. Vasudev and K. Konstantinos, *Image and Video Compression Standards, Algorithms and Architectures*, Kluwer Academic Publishers, pp. 116-119, 1995.

[4] R. Bhargava, L. K. John, B. L. Evans, and R. Radhakrishnan, "Evaluating MMX Technology Using DSP and Multimedia Applications," in *Proc. MICRO-31*, pp. 37-46, Dec. 1998.

[5] P. J. Burt, "The Pyramid as a Structure for Efficient Computation,"

Multiresolution Image Processing and Analysis, ed. Rosenfeld, Springer Verlag, pp. 6-35, 1984.

[6] I. D. D. Curcio and A. Hourunranta, "QoS of Mobile Videophones in HSCSD Networks," in *Proc. 8th Int. Conference on Computer Communications and Networks ICCCN*, pp. 447-451, Oct. 1999.

[7] R. Fletcher, *Practical Methods of Optimization*, Second Edition, John Wiley & Sons, Chapter 3 and Chapter 6, 1987.

[8] S. Fuller, "Motorola's AltiVec Technology," available via http://www.motorola.com/SPS/PowerPC/AltiVec/facts.html, Dec. 1999.

[9] G. H. Golub and C. van Loan, *Matrix computations*, 2nd edition, The Johns Hopkins University Press, 1989.

[10] "Intel Performance Library Suite," available via http://developer.intel.com/vtune/perflibst/, Sept. 1999.

[11] "Increasing the Accuracy of the results from the Reciprocal and Reciprocal Square Root Instructions Using the Newton-Raphson Method," Application Note AP-803 v. 2.1, available via www.intel.com, Jan.1999.

[12] "Intel Architecture Optimization Reference Manual," available via http://developer.intel.com/design/pentiumii/manuals/245127.htm., Oct. 1999.

[13] International Organisation for Standardisation, MPEG-4 Video Group, "Overview of the MPEG-4 Standard," ISO/IEC JTC1/SC29/WG11N2323, Dublin, Ireland, available via http://drogo.cselt.stet.it/mpeg/standards/mpeg-4/mpeg-4.htm, July 1998.

[14] ITU-T, Recommendation H.263, "Video Coding for Low Bit Rate Communication," Feb. 1998.

[15] ITU-T Study Group 16, "MVC Video Codec – Proposal for H.26L," (Q15-F-24), Nov. 1998.

[16] ITU-T Study Group 16, "Improvements to the Telenor proposal for H.26L: More blocksizes for prediction and RD constrained quantization of transform coefficients," (Q15-H-10), Aug. 1999.

[17] ITU-T Study Group 16, "Video Coding Using Long-Term Memory and Affine Motion-Compensated Prediction," (Q15-G-21), Feb. 1999.

[18] ITU-T Study Group 16, "MVC Decoder Description," (D.456), Feb. 2000.

[19] R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, Inc., pp.188-189, 1991.

[20] M. Karczewicz, J. Nieweglowski, P. Haavisto, "Video Coding Using Motion Compensation with Polynomial Motion Vector Fields," *Image Communication Journal Special Issue on MPEG-4*, Vol. 10, Nos. 1-3, 1997.

[21] M. Karczewicz, *Modeling of Data Using Polynomial and Spline Functions and Its Application to Signal Processing*, Dr. Tech. Thesis, Tampere University of Technology, pp. 29-32, Aug. 1997.

[22] R. G. Keys, "Cubic convolution interpolation for digital image processing," *IEEE Trans. Acoust., Speech, Signal Processing*, Vol.29, No.6, pp.1153-1160, 1981.

[23] V. Lappalainen, *Implementation of H.263 Video Encoder Using Intel MMX Technology*, M. Sc. Thesis, Tampere University of Technology, Aug. 1997.

[24] V. Lappalainen, "Performance Analysis of Intel MMX Technology for an H.263 Video Encoder, " in *Proc. ACM MULTIMEDIA* '98, pp. 309-314, Sep. 1998.

[25] R. B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, vol. 16(4), pp. 51-59, Aug. 1996.

[26] A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture", *IEEE Micro*, vol. 16(4), pp. 42-50, Aug. 1997.

[27] P. Ranganathan, S. Adve, and N. P. Jouppi, "Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions," in *Proc. ISCA26*, pp. 124-135, May 1999.

[28] T. Thakkar and T. Huff, "The Internet Streaming SIMD Extensions," *Intel Technology Journal*, Q2, 1999.

[29] R. Thekkath, "An Architecture Extension for Efficient Geometry Processing," in *Proc. HOTCHIPS11*, pp. 263-274, Aug. 1999.

[30] M. Tremblay, M. O'Connor, V. Narayanan, and H. Liang, "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16(4), pp. 10-20, Aug. 1996.

[31] J. H. Wolf, "Programming Methods for the Pentium III Processor's Streaming SIMD Extensions Using the Vtune Performance Enhancement Environment," *Intel Technology Journal*, Q2, 1999.

[32] S. F. Wu and J. Kittler, "A differential method for simultaneous estimation of rotation, change of scale and translation," *Signal Processing: Image Communication*, Vol. 2, No. 1, pp. 69-80, May 1990.

[33] P. M. Zagacki, D. Buch, E. Hsieh, D. Melaku, V. Pentkovski, H-H. Lee, "Architecture of a 3D Software Stack for Peak Pentium III Processor Performance," *Intel Technology Journal*, Q2, 1999.

X. BIOGRAPHY

Ville Lappalainen was born in Lempäälä, Finland, on September 1, 1974. He received his M. Sc. ("with distinction") in computer science from Tampere University of Technology, Finland, in 1997. In 1996, he joined Nokia Research Center in Tampere, where he works as a research engineer in the Media Processors group. He is currently working toward his Dr. Tech. degree. He has authored and co-authored several conference articles related to performance of video coding algorithms. His current research interests are in the area of video coding, mainly in the issues regarding architecture developments for efficient implementations of video coding algorithms.