REUSE OF HIGH PRECISION ARITHMETIC HARDWARE TO PERFORM MULTIPLE CONCURRENT LOW PRECISION CALCULATIONS

Daniel F. Zucker and Ruby B. Lee

Technical Report CSL-TR-94-616

April 1994

This research has been supported by NASA under grant number NAG2-842.

Reuse of High Precision Arithmetic Hardware to Perform Multiple Concurrent Low Precision Calculations

by

Daniel F. Zucker and Ruby B. Lee

Technical Report CSL-TR-94-616

April 1994

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-4055

Abstract

Many increasingly important applications, such as video compression, graphics, or multimedia, require only low-precision arithmetic. However, because the widespread adoption of the IEEE floating point standard has led to the ubiquity of IEEE double precision hardware, this double precision hardware is frequently used to do the low precision calculations. Naturally, it seems an inefficient use of resources to use 54 bits of hardware to perform an 8 or 12 bit calculation.

This paper presents a method for packing operands to perform multiple low precision arithmetic operations using regular high precision hardware. Using only source level software modification, a speedup of 15% is illustrated for the Discrete Cosine Transform. Since no machine-specific optimizations are required, this method will work on any machine that supports IEEE arithmetic. Finally, an analysis of speedup and suggestions for future work are presented.

Key Words and Phrases: Arithmetic, floating point, discrete cosine transform, data compression, double enhancement

Table of Contents

List of Figures

1. Introduction

IEEE floating point arithmetic hardware has become widely available on most workstations and personal computers. This trend towards high precision calculations has been largely motivated by scientific applications where higher precision translates to a more accurate calculation.

There are many increasingly important applications, however, where high precision is unnecessary. Video, graphics, multimedia, and data compression algorithms generally operate on only 8 or 16 bit words. Yet since floating point arithmetic is faster on many machines, these algorithms are often computed using floating point data types. To use a 64 bit data path to calculate on 8 or 16 bit quantities is clearly an inefficient use of resources.

This paper proposes a methodology in which a single high precision multiplier can be reused to perform multiple low precision operations in parallel. The technique presented is implemented completely in high-level software--no machine specific optimizations are required.

The Discrete Cosine Transform, popular in many image compression algorithms, is performed as an example to show the feasibility of this methodology. Two operations were performed in parallel. This is referred to as "double enhanced." Three separate algorithms are used to calculate the DCT. The most realistic case produced a speedup of 15%.

Finally, a model to describe obtainable speedup is presented, and suggestions are made for further enhancements.

2. Methodology

2.1 Overview

The basic idea is to use simple arithmetic operations to pack two operands into the left and right parts of a single register. Standard floating point hardware is then used to operate on these packed operands, now performing two operations in parallel. Only multiplication by a constant, addition, or subtraction will produce correct results. In fact, there are a number of criteria that must be met for this methodology to be worthwhile. This technique is not suitable for speeding up all calculations, but is designed for DSP-style calculations in which these four criteria will be met:

1) Because additional instructions must be added to pack and unpack operands before and after arithmetic calculations, it is desirable to have a long series of calculations between the packing and unpacking so the packing/unpacking cost is effectively amortized over a large number of operations.

2) Because this technique relies on packing multiple lower precision operations into a single high precision unit, it is essential that the functions it calculates require only lower precision calculations.

3) Because careful attention must be paid to operand length and magnitudes in order to insure correct results, information about the range and domain of the data must be known beforehand.

4) Because calculations done on each operand concurrently must be the same, the application is best suited for multiple loop iterations done in parallel.

The DCT calculation, used in the JPEG image compression standard, meets all these criteria. 16 1-D DCTs must be calculated to perform a single 8x8 2-D DCT, so there are many identical loop iterations that can be done in parallel. Baseline JPEG is defined for 8 bits of precision, so condition 2 is easily met. Furthermore, from the input and output restrictions on the range of data, and from the mathematical properties of the DCT, we have the necessary information referred to in condition 3. Though the DCT is used in this paper to illustrate the double enhancement technique, it is generally applicable to similar DSP-style applications.

2.2 Packing

Packing is performed by a multiplication by a constant and an addition. If two words, L and R, are to be packed into a single word X, then

$$X = L*2^C + R \qquad (1)$$

where c is a suitably chosen constant. In binary form, this might look like

$$X = eeeeee000000rrrrrr \qquad (2)$$

where e and r are the bits in L and R, respectively. One word is on the left, one word is on the right, and a buffer is between them. Assuming an architecture with a floating point multiply-add instruction[12], packing can be performed with just a single instruction. Otherwise two instructions are needed

Multiplication and addition can be performed in this packed format by using the distributive property. For example, to multiply by a constant, k, we have

$$X*k = (L*k)*2^C + (R*k).$$

So, both L and R are multiplied simultaneously by the same constant, k. Or to add Y, a similarly packed number, to X

$$Y = A*2^C + B \text{ and}$$
$$X + Y = (L+A)*2^C + (R+B).$$

By using this simple algebraic method to pack two numbers, the sign information for the R term is automatically encoded in the empty bits between the L and R terms. The sign for the L term is encoded in the sign bit for the entire floating point quantity, and the bits between L and R indicate whether R is the same or opposite sign of L.

2.2.1. Example 1

Consider L=5, R=3, the constant of packing, c, equal to 4, and a decimal system. The packed quantity is obtained as:

$$
\begin{aligned}
X &= 5*10^4 + 3 \\
&= 50000 + 3 \\
&= 50003.
\end{aligned}
$$

Since both numbers are positive, the bits between them are zero.

2.2.2. Example 2

If, now, R=-3, then

$$
\begin{aligned}
X &= 5*10^4 - 3 \\
&= 50000 - 3 \\
&= 49997.
\end{aligned}
$$

Now, the bits in the buffer area are non-zero, indicating that the R term is opposite in sign to the L term.

2.2.3. Example 3

Now consider the same example in binary: L=101, R=11, and the constant of packing, c, equals 5. The packed quantity is obtained as:

$$
\begin{aligned}
X &= 101*2^5 + 11 \\
&= 10100000 + 11 \\
&= 10100011.
\end{aligned}
$$

Again, since both numbers are positive, the bits between them are zero.

2.2.4. Example 4

If, now, R=-11, then

$$
\begin{aligned}
X &= 101*25 - 11 \\
&= 10100000 - 11 \\
&= 10011101.
\end{aligned}
$$

Again, the bits in the buffer area are non-zero, indicating that the R term is opposite in sign to the L term.

2.2.5. Example 5

If L=-5 and R=3, the packed quantity becomes:

$$\begin{aligned} X &= -5*10^4 + 3 \\ &= -50000 + 3 \\ &= -(50000 - 3) \\ &= -49997. \end{aligned}$$

In this case, the minus sign for the entire fp number is stored in the sign bit. Also, the fact that R has the opposite sign from L is indicated by the non-zero digits in the buffer area between L and R.

2.3. Unpacking

Unpacking is accomplished by rounding X just so that all bits of R are eliminated. L is recovered by a multiplication, and R is recovered with a subtraction.

$$L = \text{round}(X) * 2^{-c} \qquad (3)$$
$$R = X - \text{round}(X). \qquad (4)$$

2.3.1. Example 1

Consider a base 10 system where L=12, R=-15, and C=4.

$$X = 120000 - 15 = 119985.$$

To recover, X must be rounded to the 10^3 place, since R is 3 digits long (including the sign). This is one digit to the left of R. Then,

$$\begin{aligned} \text{round}(X) &= 120000 \\ L &= 120000 * 10^{-4} = 12 \\ R &= 119985 - 120000 = -15. \end{aligned}$$

2.3.2. Example 2

If X were truncated instead of rounded, then an incorrect result may be obtained. For example for the above example with L=12, R=-15, and C=4:

$$\begin{aligned} \text{trunc}(X) &= 119000 \\ L &= 119000 * 10^{-4} = 11.9 \\ R &= 119985 - 119000 = 985. \end{aligned}$$

2.3.3. Example 3

This will also work for a base 2 system. Consider L=7, R=-6, and C=6.

$$X = 111000000 - 110 = 110111001.$$

To recover, X must be rounded to the 2^4 place, since R is 4 digits long. Then,

```
round(X) = 111000000
L = 111000000 * 2^-6 = 111
R = 110111001 - 111000000 = -110.
```

2.3.4. Example 4

Similarly, in a base 2 system, if X were truncated instead of rounded, then an incorrect result may be obtained. Consider the above example with L=7, R=-6, and C=6:

```
trunc(X) = 110110000
L = 110110000 * 10^-6 = 110.11
R = 110111001 - 110110000 = 1001.
```

2.4. Length, Absolute Magnitude, and Relative Magnitude

It is extremely important to keep track of length and magnitude of the operands to ensure that the final results can be successfully separated into the original components. The relative magnitude of R and L is defined as |L/R| and is a measure of the distance between the left most bits of the two parts. For most cases in a floating point system, only the relative magnitude is important since large changes in absolute magnitudes can be handled by simply changing the exponent field (assuming the relative magnitude is unchanged). For a fixed point system, absolute magnitude is also important since care must be taken that X does not overflow.

Imagine a packed word, X, composed of two parts, L and R, with the bits in L represented by e, and the bits in R represented by r.

```
X = eeeeee000000rrrrrr   (2)
```

As L becomes longer the buffer space between the numbers will become smaller. Therefore, one must assure that the buffer space is large enough to accommodate the growth of L. As R becomes longer there will be no effect in a floating point system, since the hardware will left align the word to the leftmost bit of L, and round R so that it will fit into the number of bits available in mantissa hardware. In a fixed point system, the length of R must also be considered since X will be right aligned to the rightmost bit of R. If R is too large, X will overflow.

As the relative magnitude increases in a floating point system, precision in R will be lost. That is, as the space between L and R increases, more and more of R will have to be rounded off to fit into the available hardware. As the relative magnitude decreases, the space between the parts will shrink and there will be more of a tendency to overlap, though R will have more bits of precision.

In order to ensure that the final result is separable, we must guarantee that the maximum length of L and the relative magnitude are such that the rightmost bits of L do not overlap with the leftmost bits of R.

2.5. Implementation for Experiments

2.5.1. Packed Formats

In order to determine how to pack the operands for the DCT examples, we must first determine the largest possible value. If the DCT is used in JPEG, then the inputs are restricted to 8 bits or less. (We ignore, for these experiments, the case where the input data is 12 bits.) Mathematically, we know that the terms can grow by at most 3 bits within an 8 point 1-D DCT, so we know that at most 11 bits will be required to hold the data. However, since the sign bit for L will be held in the sign bit for the entire fp number, only 10 bits are needed to represent L. We begin by allocating 21 bits to hold the integer portions of L and R. Since we have 53 bits available in the IEEE double precision format, this leaves 32 bits.

Next, we must worry about the increase in length of L after every multiply (remember that the R component will get automatically rounded so there is no problem here). From the above calculation, we know that L can grow by at most 32 bits. Multiplication by a constant will result in an increase in length equal to the length of the constant. To be able to perform a 1D Lee's DCT[3], we must be able to do 4 sequential multiplies. So we must express the constants as 32/4 = 8 bit quantities so that the left component grows as shown below. Note that the 11th bit of L is stored as the sign bit for the entire fp quantity.

```
left part  mult 1   mult 2  mult 3 mult 4  right part
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyy
<---10---><--8---><--8---><--8---><--8---><----11--->
```

Inspection of images compressed using these 8 bit constants showed no perceptible difference with images compressed using full 53 bit constants.

The figure below shows the signal to noise ratio in decibels for an image compressed with a range of quantization factors compared to the original uncompressed image. The two lines compare images in which the DCT was performed with multiplication by constants of only 8 bits, and with constants of full 53 bit precision. A low Quality Factor constant indicates large quantization, and a Quality Factor of 100 indicates quantization only to the nearest integer. A small deviation between the two lines is evident only at high Q Factors.

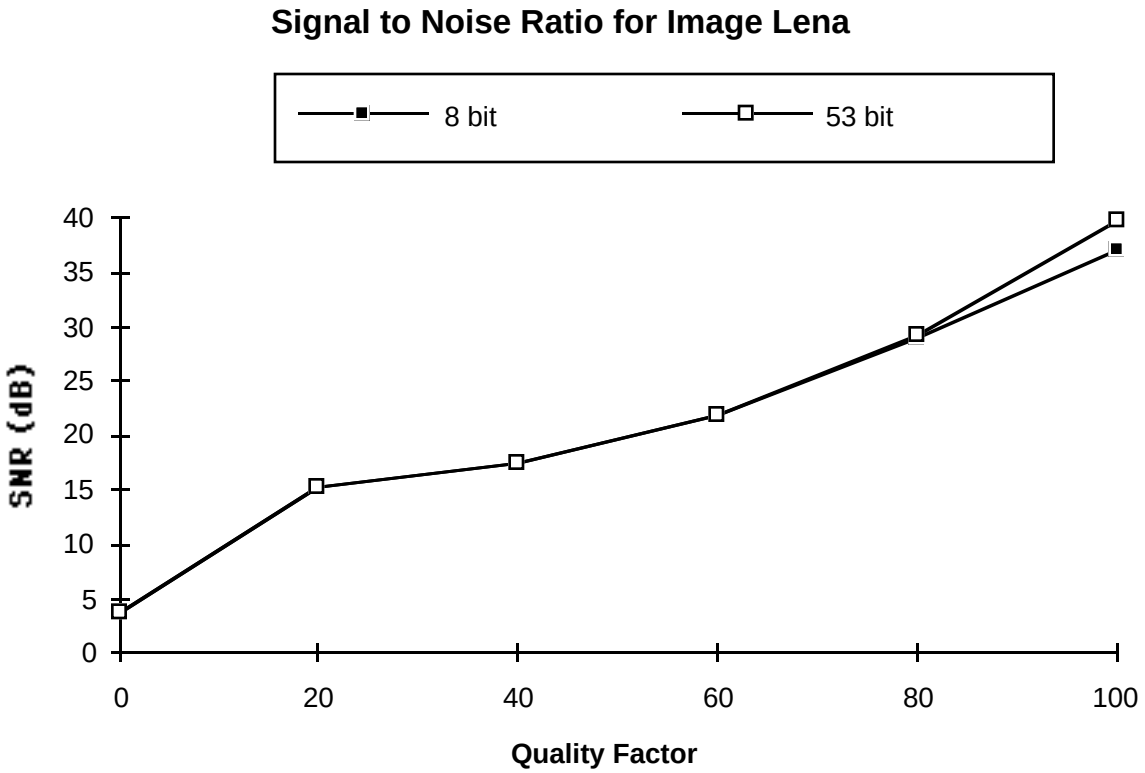


figure 1 Signal to noise ratio vs. quality factor

The implemented format shows that L must be spaced 43 bits over from R. Therefore, for these experiments, the constant of packing was set equal to 2^{43}.

2.5.2. Rounding Implementation

Any method may be used to implement the round operation required to unpack the operands as described in equations (3) and (4) above. For these experiments, the round was implemented by performing an addition and subtraction with a large predetermined constant.

This method takes advantage of the round that must be performed by the hardware when a number becomes too large to be fit into available hardware. First, the large constant is added to force the round at the desired position, and second, the constant is subtracted to restore the correct value to the result.

2.5.2.1. Example 1

Assume a 6 digit decimal system where it is desired to round X to the nearest 1000. For this example, the correct constant is 100000000.

```
X = 498657
X + 100000000 = 100498657
```

```
X + 100000000 = 100499 * 10^3   Round must be performed to fit
                                      result into available 6
                                      decimal places
(X + 100000000) - 100000000 = 499000.
```

The correctly rounded result is obtained.

2.5.2.2. Example 2

This method will also work for binary. Assume a 6 binary decimal system where it is desired to round X to the nearest $8 = 1000_2$. For this example also, the correct constant is 100000000_2.

```
X = 110111
X + 100000000 = 100110111
X + 100000000 = 100111 * 2^3    Round must be performed to fit
                                      result into available 6
                                      binary places
(X + 100000000) - 100000000 = 111000.
```

The correctly rounded result is obtained again.

Implementing the round in this manner with an add and subtract means that 2 operands can be completely unpacked with a multiply, an add, and 2 subtracts. Some of these operations can be combined with multiply-add or multiply-subtract instructions.

2.5.3. Method for Extracting Parallelism

The calculation of a 64 element 8 point two dimensional DCT can be done by performing 8 one dimensional DCTs on the rows followed by 8 one dimensional DCTs on the columns of the input matrix. Therefore, it might seem reasonable to use the double enhanced methodology to calculate two rows or two columns in parallel. However, a problem is reached when the switch is made between doing calculations on the rows and the columns. The operands would have to be unpacked and then packed again in a transposed layout in order to do the switch correctly. The amount of packing and unpacking overhead would be doubled, since each element would now have to be packed and unpacked twice.

To avoid this additional overhead, calculations are done on two independent 2D DCTs in parallel, rather than two rows or two columns within a single DCT. The amount of parallelism achieved is identical, and no additional packing and unpacking is required.

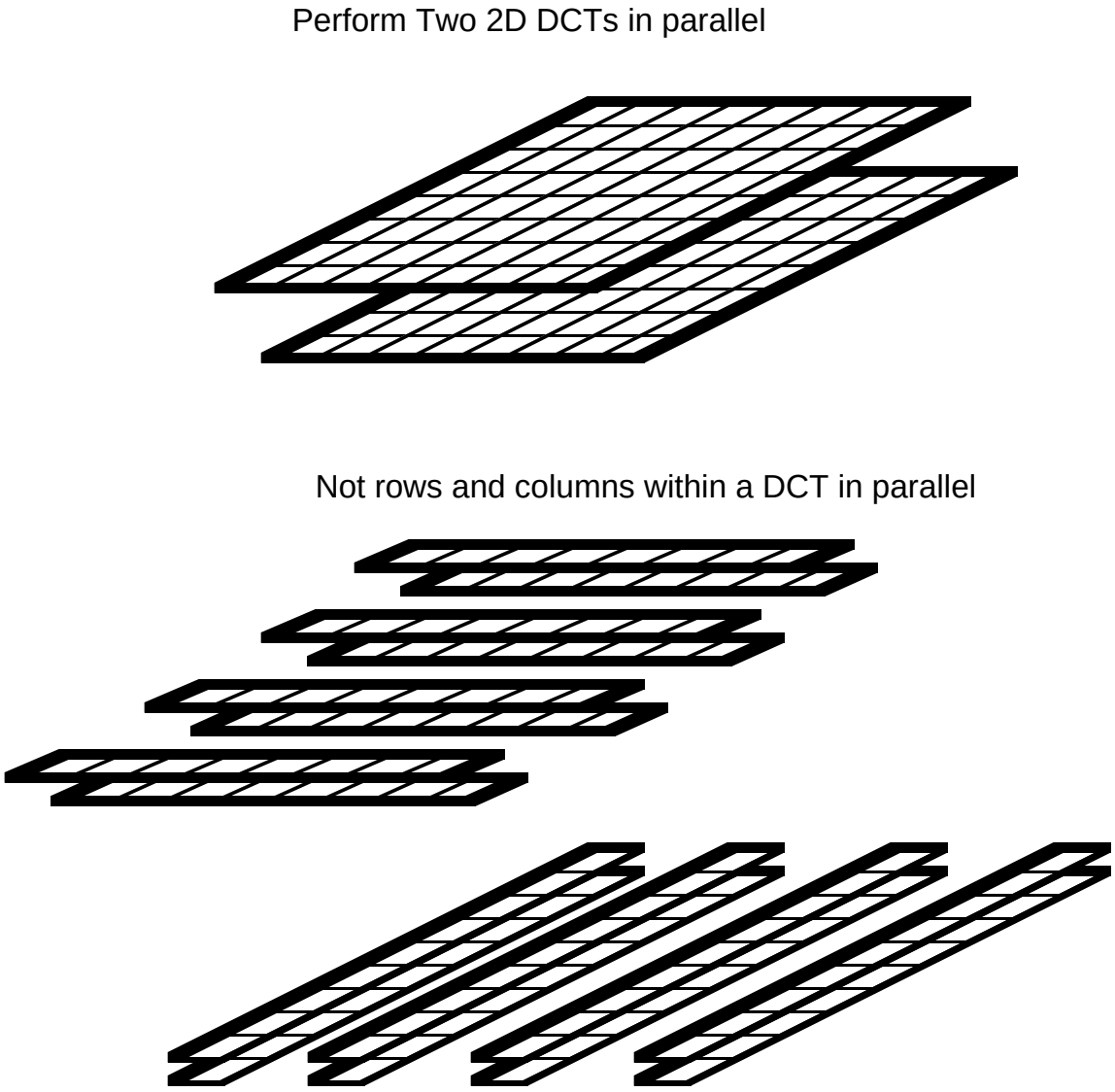


figure 2 Method for extracting parallelism

The disadvantage with performing the two 2-D DCTs in parallel is the slight initial delay in having to decode two 8x8 blocks before the DCT may be performed in parallel on both these blocks. Also, fast storage is needed for two 64 element blocks rather than just one. For JPEG, the additional startup delay is not important, and the cache sizes of most workstations can usually accommodate two 64 element blocks quite easily since 128x8 = 1 Kbytes.

3. Data

3.1 Overview

Three methods were used to compute 64 element 2D DCTs. The algorithms used were Lee's, Loeffler's[4], and an earlier slower version of Lee's which was coded with unnecessarily high overhead. DCT input data was obtained from a standard “ppm” image.

Data was first loaded from memory, then the DCT calculations were performed, and finally the results were stored to memory. The same calculations were performed using both regular fp arithmetic and double enhanced technique so that accurate comparisons could be made. For the regular arithmetic, two 64 element DCTs were calculated sequentially. For the double enhanced case, the data for two 64 element arrays was loaded and packed, a single DCT was calculated that performs the two calculations in parallel, and finally the data was unpacked and stored to memory.

The C code for the DCTs was compiled on a Hewlett Packard 9000/720 workstation using the HP92453-01 A.09.34 HP C Compiler with optimization set to -O. No hand assembly code optimizations were performed. The data below was obtained from simulations assuming an infinite cache size. However, this is not unreasonable since most workstations can easily accommodate two 64 element blocks in cache, and hence no additional cache miss performance degradations are likely in going from the regular to the double enhanced methods.

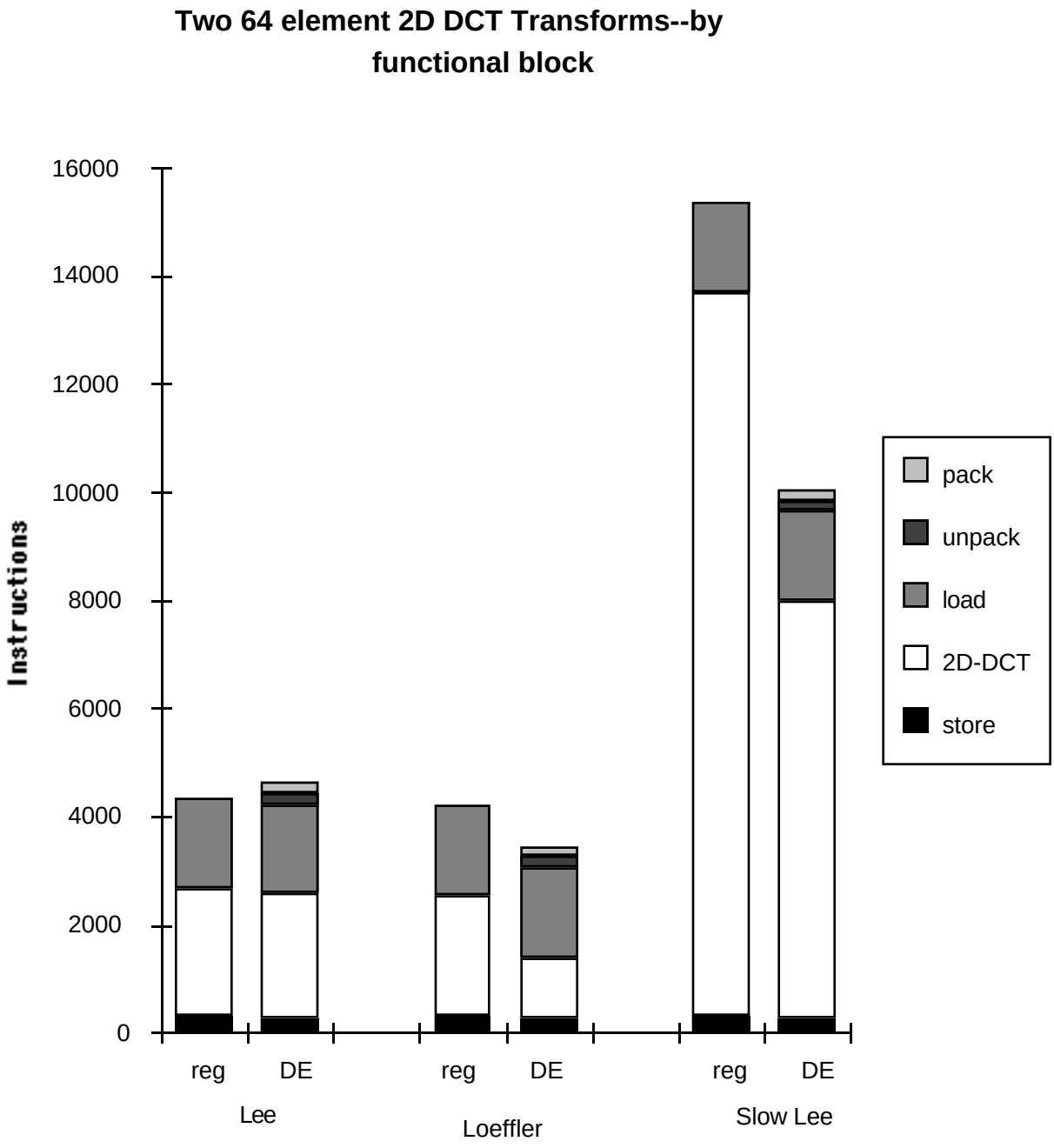


figure 3 Total instruction count

This graph shows the dynamic instruction count broken up by the five functions of pack, unpack, load, 2D-DCT, and store for the three algorithms. The change in dynamic instruction count going from the regular to the double enhance (DE) implementation are -6.4%, 21.5%, and 53.2% for Lee, Loeffler, and Slow Lee algorithms respectively. The load, store, pack, and unpack blocks are the same regardless of DCT algorithm used.

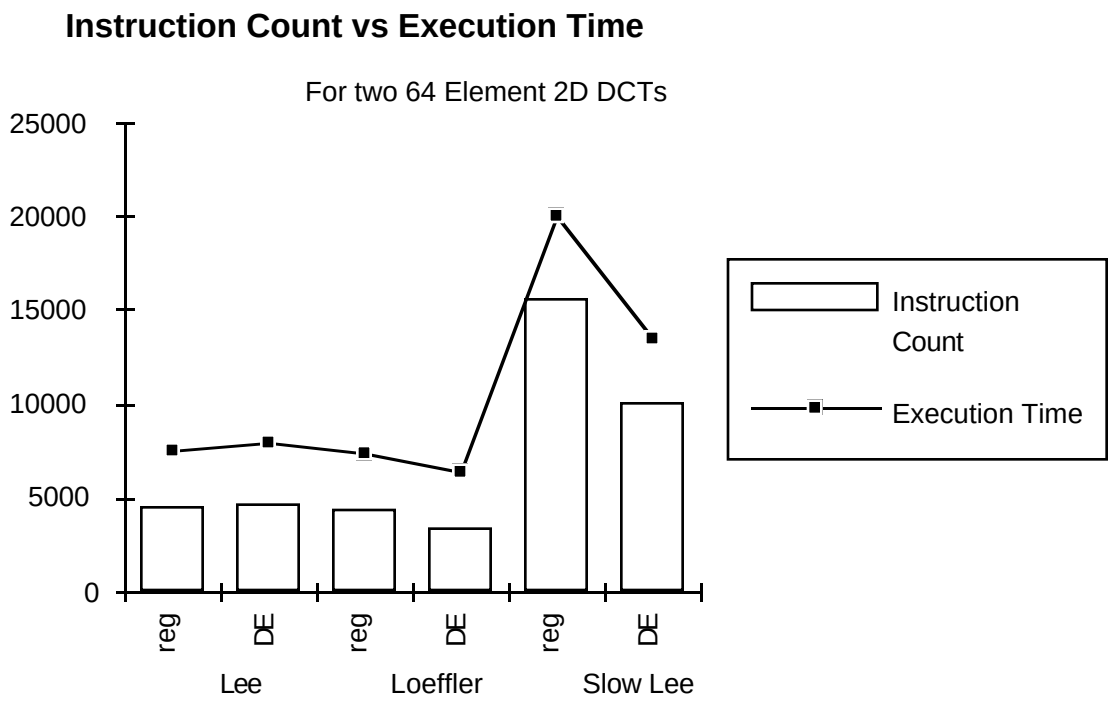


figure 4 Instruction count vs. execution time

This graph compares dynamic instruction counts with actual execution times. From the graph it is clear that these two quantities track each other fairly well. Actual execution time is higher due to cold start cache misses and pipeline interlocks. Speedups of the DE over the regular implementation obtained from execution times, are -5.8%, 15.1%, and 48.2% for Lee, Loeffler, and Slow Lee respectively.

3.2 Load and Pack

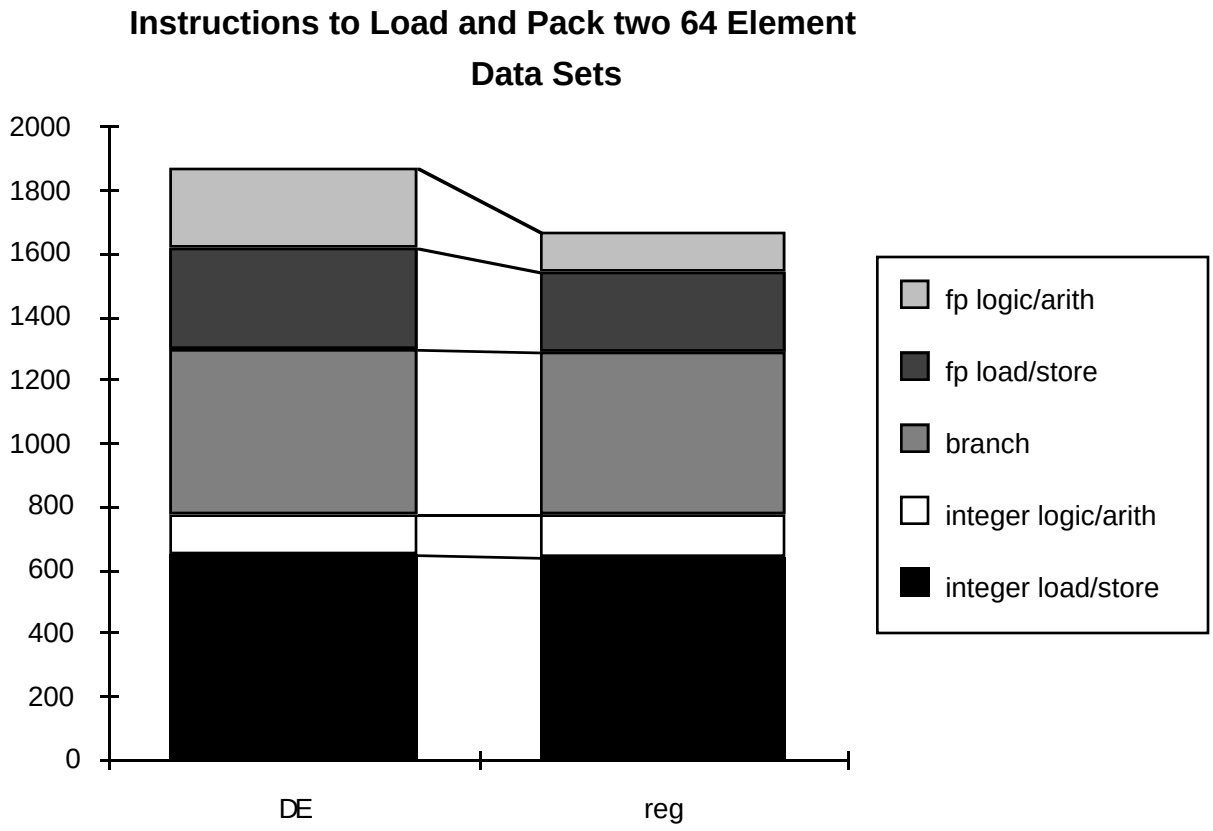


figure 5 Load and pack instruction count

Instruction mix for load and pack

	DE	% of total	regular	% of total
LDW	387		384	
LDO	3		2	
STWS	129		128	
STW	128		128	
integer load/store	647	35%	642	38%
OR	130		130	
integer logic/arith	130	7%	130	8%
COMIBT	129		128	
BV	129		128	
BL	129		128	
ADDIBT	128		128	
ADDIBF	0		0	
COMBT	1		0	

branch	516	28%	512	31%
CLDDS	64		0	
CSTDS	128		128	
CLDWS	128		128	
fp load/store	320	17%	256	15%
FCNVXF,SGL,DBL	128		128	
FMPY,DBL	64		0	
FADD,DBL	64		0	
FSUB,DBL	0		0	
FMPYSUB	0		0	
fp logic/arith	256	14%	128	8%

Aside from slight differences due to overhead, the double enhanced load routine requires 64 extra fp loads, 64 extra fp adds, and 64 extra fp multiplies from the regular method. The 64 extra fp adds and multiplies are understandable since each DCT element requires 1 fp add and 1 fp multiply to pack. The 64 extra fp loads result since the data must be loaded from memory sequentially in 64 element blocks. In the first pass, 64 elements are copied from sequentially accessed memory to a local workspace. In the second pass, the second block of 64 elements are loaded from sequential memory and packed with the elements from the local workspace. An extra fp load is required for each element for this load from the local workspace. This extra load is probably due to the way the code has been structured and is not intrinsically necessary for packing elements from 2 separate 64 element blocks.

3.3 DCT Calculation

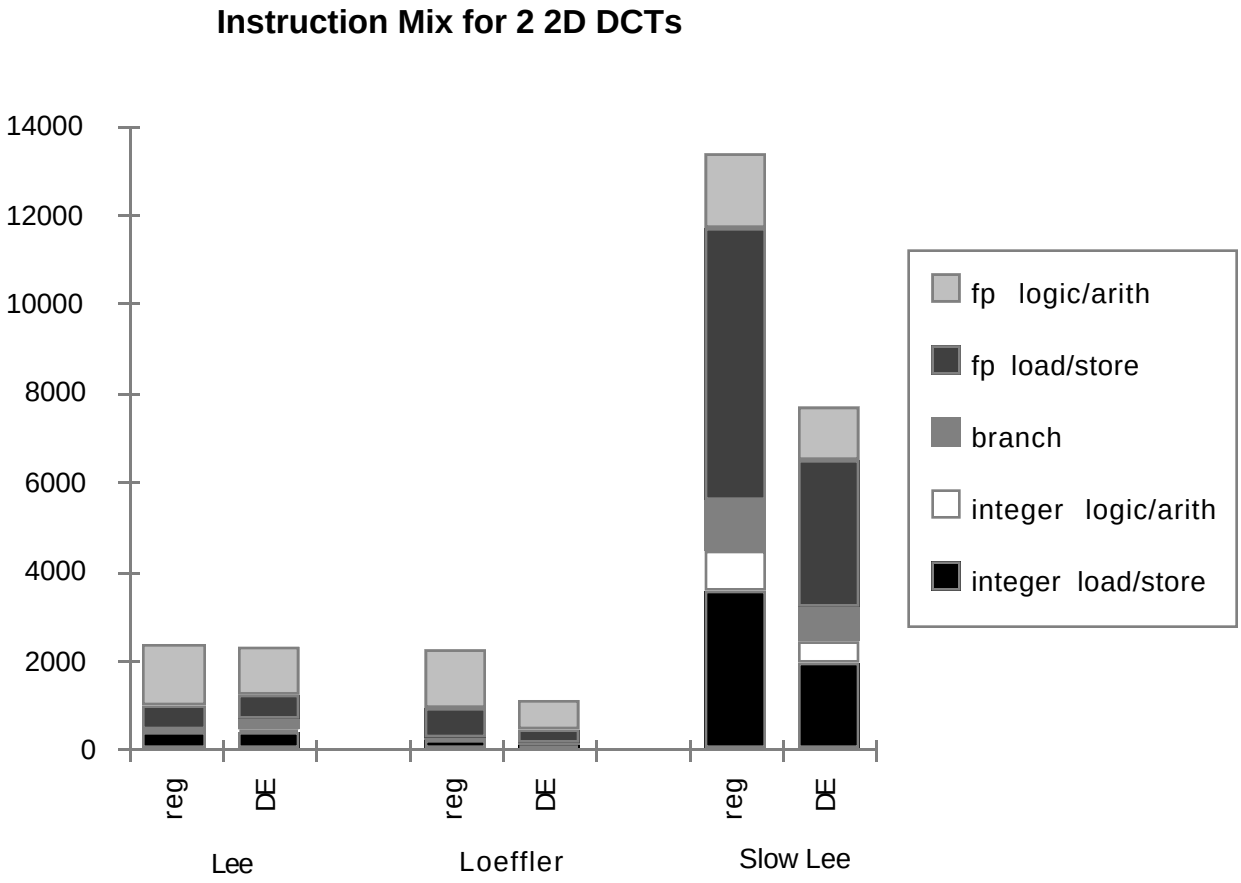


figure 6 DCT instruction count

Instruction mix for DCT

	Reg Slow Lee	DE Slow Lee	Reg Loeffler	DE Loeffler	Reg Lee	DE Lee
LDIL	16	80	1	1	1	65
LDO	1757	1837	106	106	187	315
LDW	6	8	0	0	0	5
LDWM	1	1	0	0	0	1
STW	6	8	0	0	0	4
STWM	1	1	0	0	0	1
integer load/store	1787	1935	107	107	188	391
ADDIL	16	16	0	0	0	0

OR	259	251	1	1	3	64
SH3ADD	96	160	0	0	0	0
SUBI	64	64	0	0	0	0
integer logic/arith	435	491	1	1	3	64
ADDIBF	120	96	16	16	18	18
ADDIBT	320	320	0	0	0	64
BL	17	81	1	1	1	65
BV	17	81	1	1	1	65
COMIBF	112	176	0	0	0	2
branch	586	754	18	18	20	214
CLDDS	776	960	199	199	140	336
CLDDX	872	880	0	0	0	0
CSTDS	688	752	141	141	132	200
CSTDX	720	720	0	0	0	0
fp load/store	3056	3312	340	340	272	536
FADD,DBL	272	464	280	280	176	368
FCPY,DB	16	16	0	0	0	0
FMPY,DBL	336	336	128	128	272	272
FMPYADD	0	0	120	120	32	32
FMPYSUB	0	0	8	8	80	80
FSUB,DBL	192	384	104	104	128	320
fp logic/arith	816	1200	640	640	688	1072

As described above in the section on packed formats, only four multiplies can occur before the left and right operands will overlap. Since each 1D Lee DCT requires four multiplies, if 1D DCTs were performed on the rows and columns with no special treatment, each element would see eight multiplies and incorrect results would result.

To correct this problem, it is necessary to reset the buffer space between the left and right operands. This is accomplished by unpacking the operands, rounding the left operand to the correct length, and repacking them. As evidenced from the table above, this is quite a costly operation costing 1,106 instructions for the Lee DCT and 1,012 instructions for the Slow Lee. Because of this large overhead, the cost for the DE Lee is approximately the same as the cost for the regular DCT, and there is virtually no speedup. Slow Lee is still improved despite the additional overhead, since the initial code was so inefficient.

Although the Loeffler algorithms requires approximately the same number of operations as the Lee, it has the great advantage that each data element sees only 1 multiply per 1D DCT. Thus, only 2 multiplies are required for a 2D DCT and no reset operations are required. Furthermore, it is now possible to increase the precision of the multiplies to 16 bits, since only 2 are required. It is because no reset operation is required that the Loeffler produces the best speed-up as described above. Notice that all instructions, not just arithmetic, are reduced by half. Speedup is caused not only by faster arithmetic, but also by increased data bandwidth.

3.4 Store and Unpack

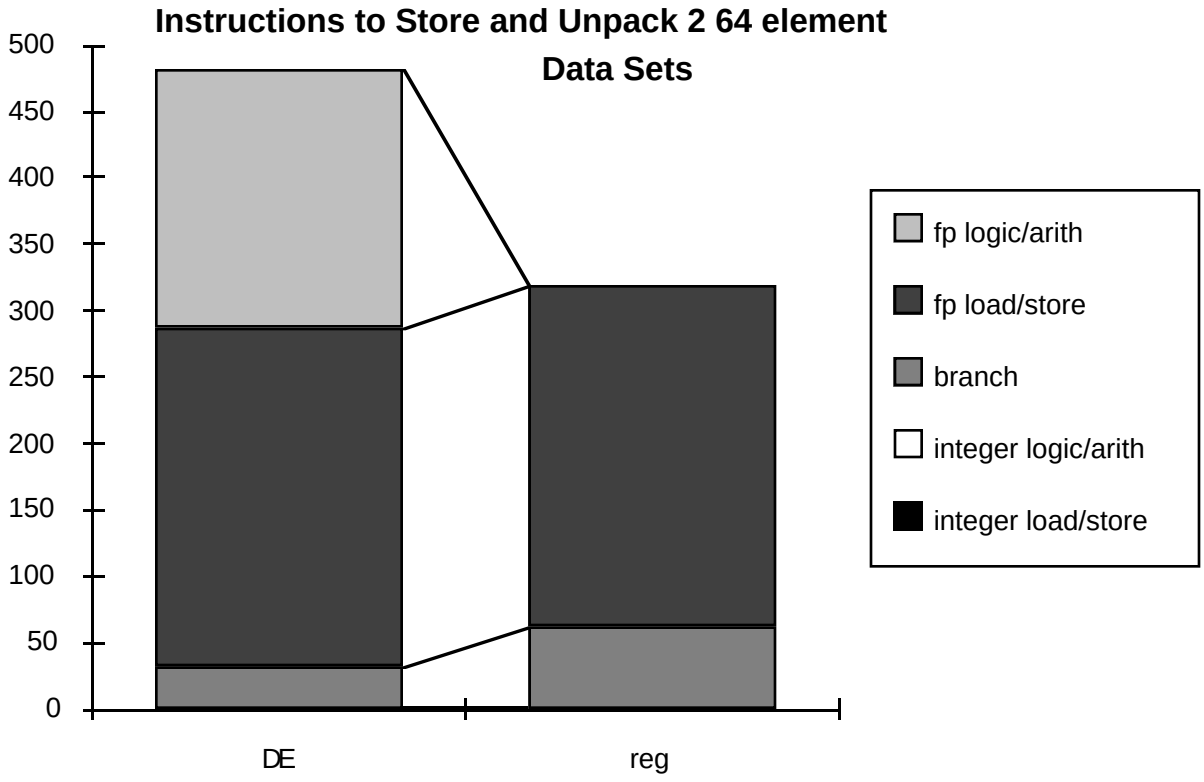


figure 7 Store and unpack instruction count

Instruction mix for store and unpack

	DE	% of total	regular	% of total
LDO	4		6	
integer load/store	4	1%	6	2%
integer logic/arith	0		0	
ADDIBF	31		62	
branch	31	6%	62	19%
CLDDS	128		128	
CSTDS	128		128	
fp load/store	256	53%	256	79%
FMPY,DBL	2		0	
FADD,DBL	64		0	

FSUB,DBL	66		0	
FMPYSUB	62		0	
fp logic/arith	194	40%	0	0%

The DE store block requires approximately 64 extra fp adds, fp subtracts, and fp multiply-subtracts. Surprisingly, it also requires 31 fewer branches. To unpack two values requires an fp add, an fp multiply, and 2 fp subtracts. Here, the compiler has done us the favor of implementing an add and subtract together in a single multiply-subtract instruction. Fewer branches are required since the DE store routine loops through only 64 elements, and unpacks them to store 128 values to memory. The regular routine loops through all 128 elements storing them to memory individually, so the DE store routine uses half as many loops, and therefore needs only half as many branches.

4. Analysis

4.1 Model

In order to determine how to maximize speedup using multiple enhanced arithmetic, the following model was developed. Begin with the following assumptions.

```
Let
   K = code that is not parallelized (load and store)
   H = pack and unpack overhead
   D = code that is parallelized (DCT)
   R = reset overhead
   P = degree of parallelism.
```

Then, speedup, defined as the execution time to perform P iterations of regular code divided by execution time to perform P iterations of multiple enhanced code is

$$Speedup = \frac{P*(K+D)}{(P*K)+(D+R)+(P-1)*H}$$

dividing by P*K gives

$$Speedup = \frac{1+(D/K)}{1+\frac{(D/K+R/K)}{P}+(\frac{P-1}{P})*(H/K)}$$

Using the data obtained above, D/K is 0.5 for Lee and Loeffler, and 6.6 for Slow Lee. R/K is 0 for Loeffler and 0.5 for Lee and Slow Lee, and H/K is 0.2 for all three.

4.2 Comparison of Predicted and Observed Data

Using the numbers above, predicted and observed data is compared as shown below. The slight mismatch between predicted and observed is partially due to rounding errors in the parameters R/K, H/K, and D/K. Also, the savings from reducing the number of store branches by half is not included in the model.

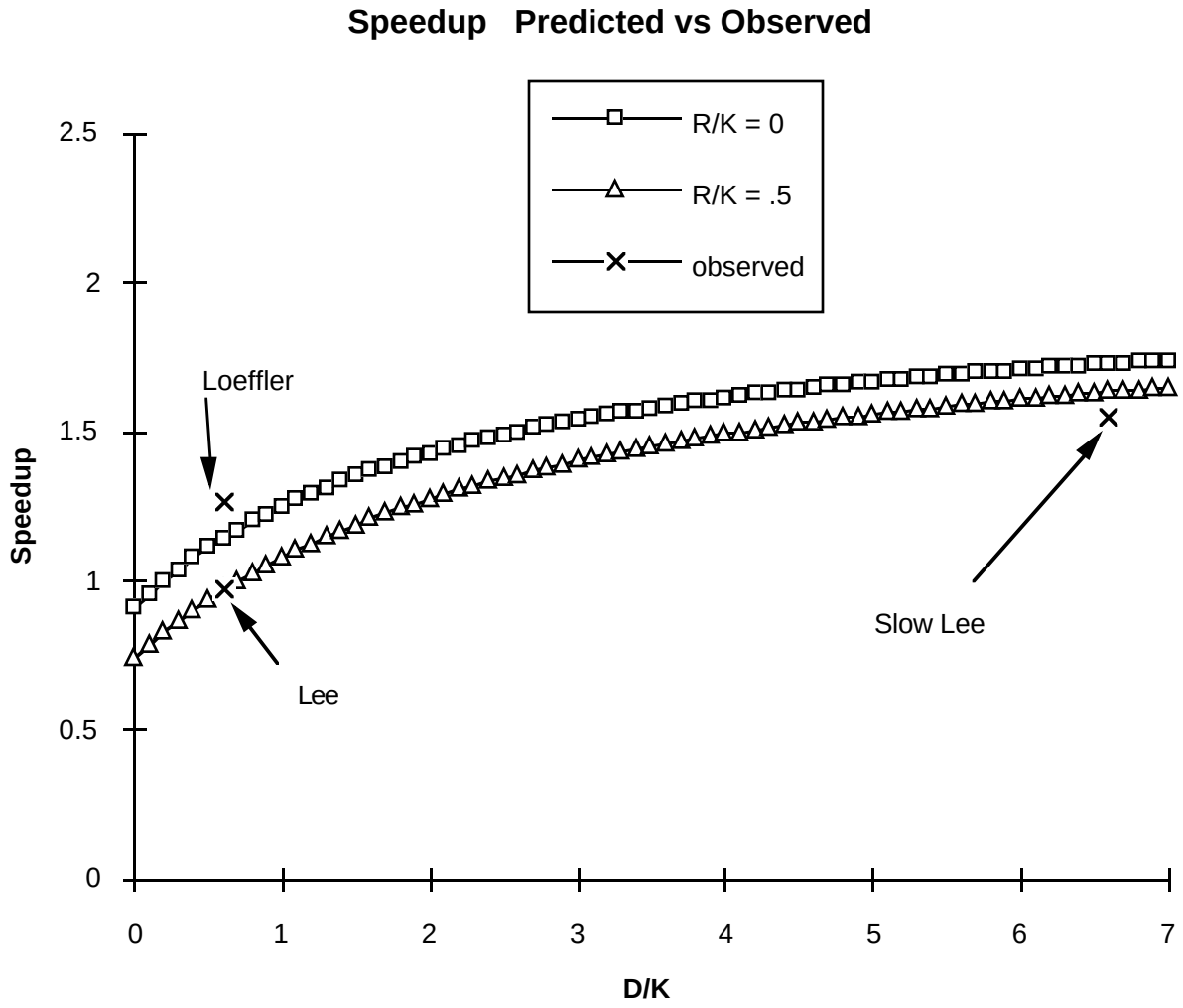


figure 8 Predicted vs. observed speedups

4.3 Explanation of Parameters

4.3.1. Ideal

The best way to understand the model presented above is first to imagine that R/K and D/K are both zero. This simplifies the model to just

$$Speedup = \frac{1+(D/K)}{1+\frac{(D/K)}{P}}$$

This is simply Amdahl's law and states that the speedup asymptotically approaches P as the ratio of parallelizable code to non-parallelizable code grows large. This is shown in the graph below.

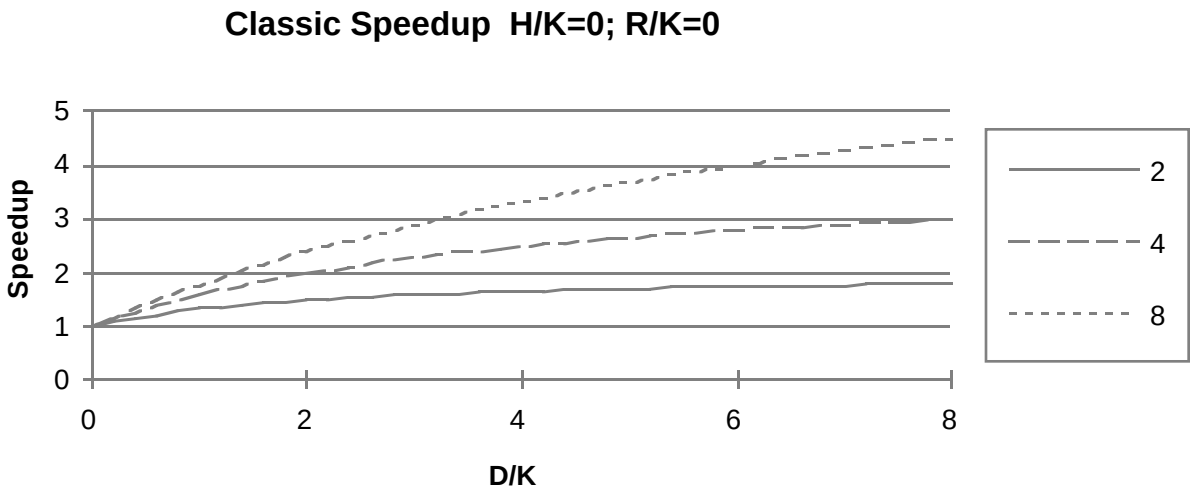
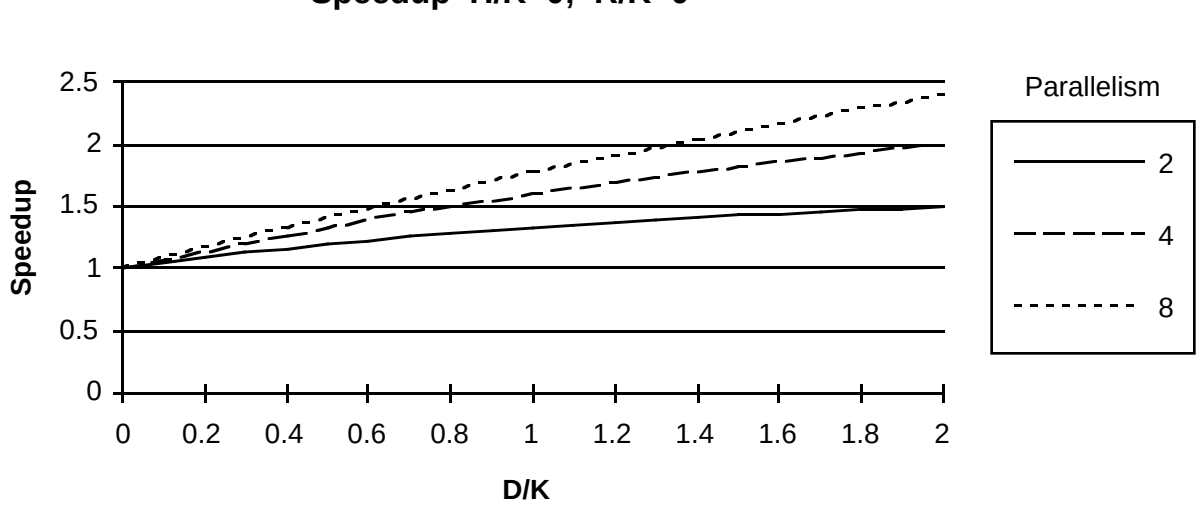


figure 9 Idealized speedup

4.3.2. Effect of H/K

Adding the H/K term simply shifts the curve slightly to the right so that the point where all the lines equal unity is now where D/K equals H/K. This is shown in the graphs below.



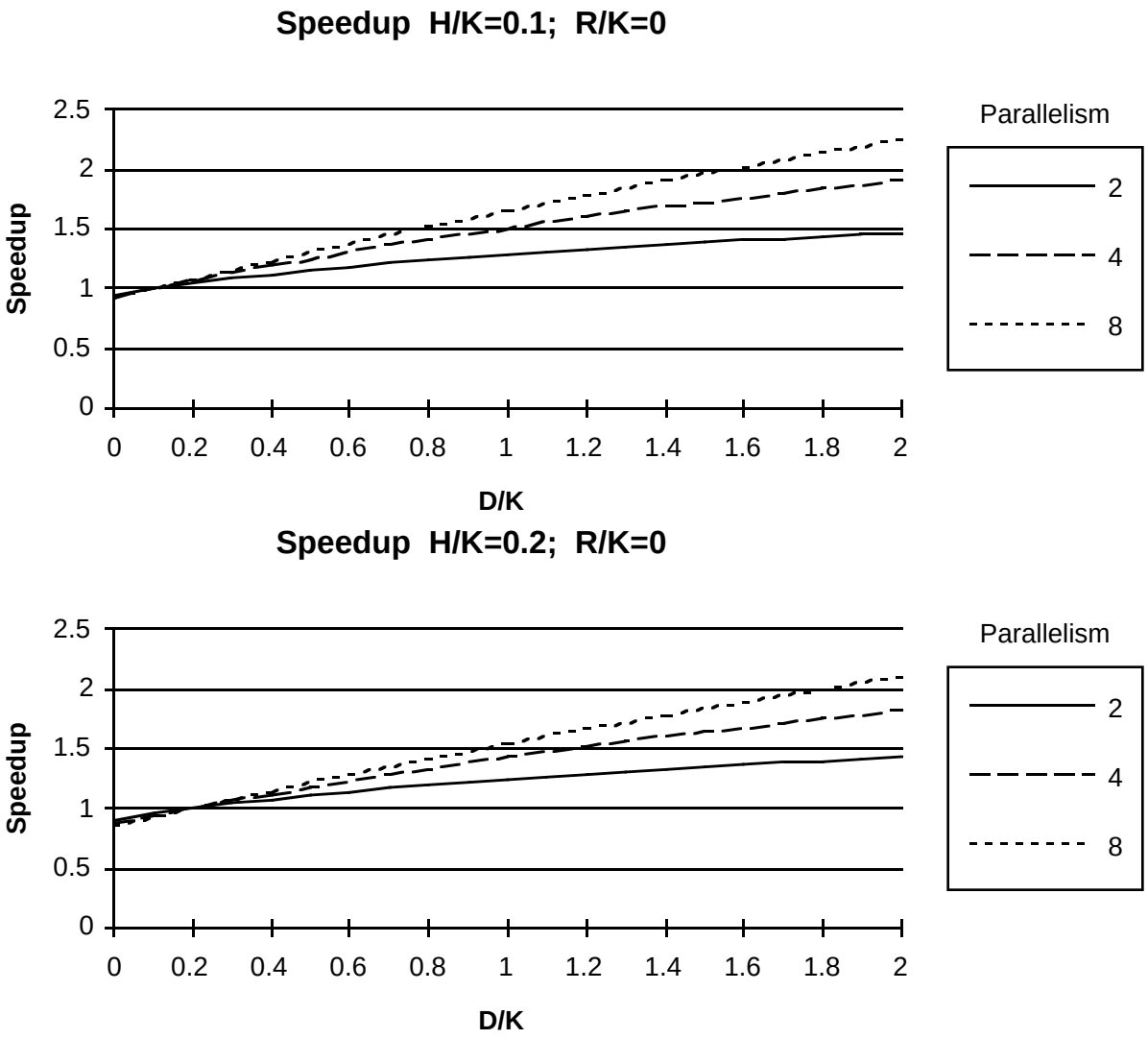


figure 10 Effect of H/K

4.3.3. Effect of R/K

The effect of R/K is to shift the curve down, and to increase the spread between the different degrees of parallelism. This increase in spread results since the added work, R, is parallelized, so that its negative impact is mitigated for higher parallelism. The effect of R/K is shown in the graphs below.

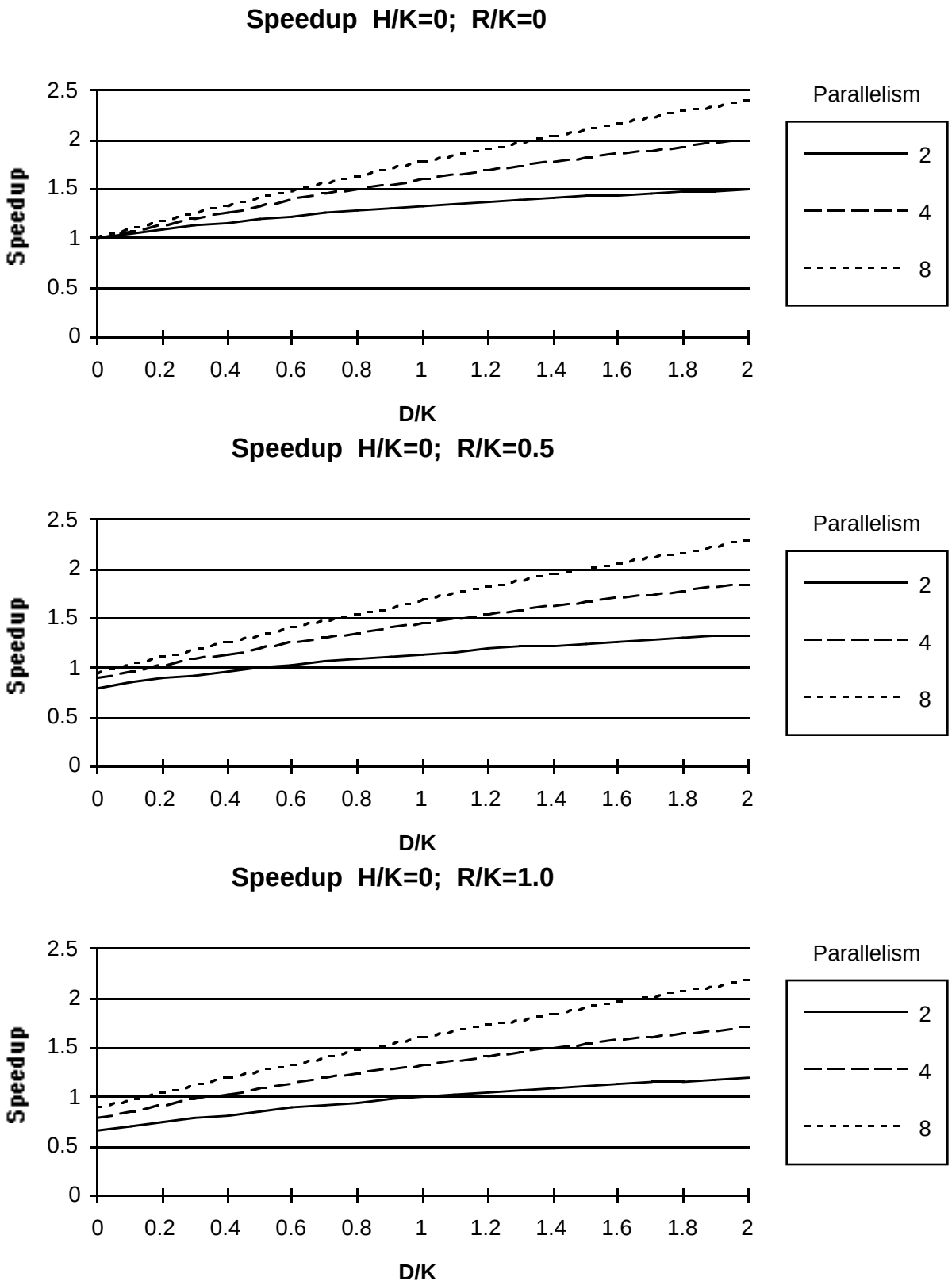


figure 11 Effect of R/K

4.4 Loeffler

The Loeffler algorithm is the best example of multiple enhanced speedup, even though the Slow Lee algorithm resulted in higher actual speedup. This higher speedup largely resulted from the inefficiency of the original algorithm, and so is not a fair measure of speedup. The Loeffler algorithm, furthermore, is better than both Lee and Slow Lee since no reset instructions are needed between 1D DCTs.

Because Loeffler's algorithm needed no reset instructions, the R/K term reduced to zero. The graph below shows how the 15% speedup for Loeffler was achieved. Notice that increasing to higher parallelism will raise speedup only slightly along the line where R/K = 0.5. Furthermore reducing H/K would have the same effect as shifting the operating point along the R/K axis to the right. If H/K were completely reduced to zero, the speedup for a parallelism of 2 would be raised to only about 20%.

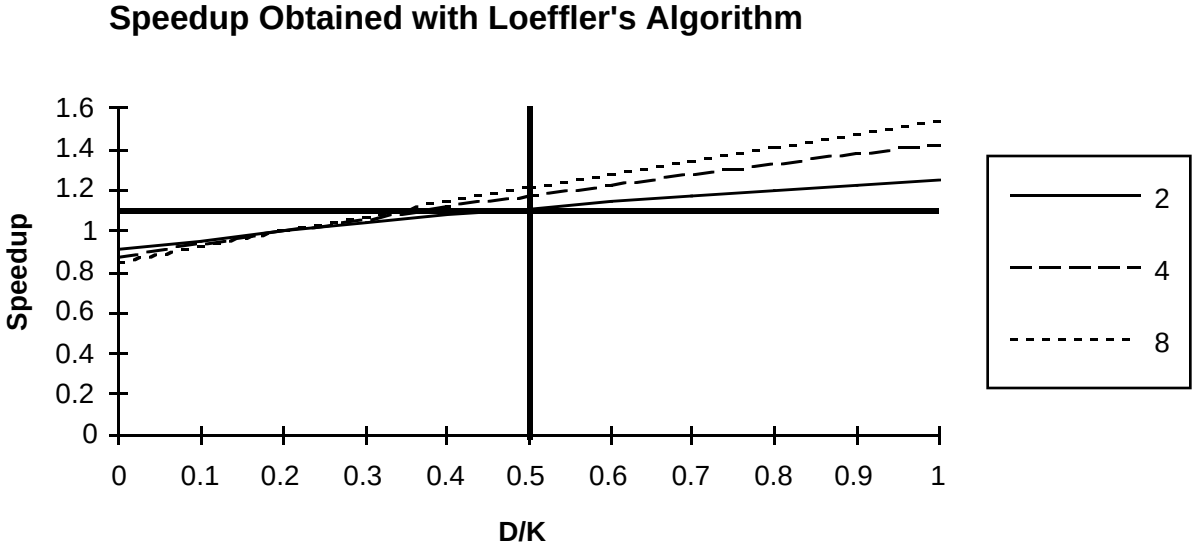


figure 12 Speedup for Loeffler's DCT

4.5 Multiple Enhanced DCT in JPEG

Using this technique to speedup the DCT in JPEG will clearly result in a fairly small speedup, since the DCT is only a fraction of the entire JPEG code. Runtime measurements of execution times show that the factor D/K is approximately 0.19 since K, or the non-parallelized part of the code, is now so large. On the other hand, this causes H/K, or the ratio of pack and unpack overhead to non-parallelized code, to be reduced to almost zero: 0.0003. Using the formula above predicts a speedup of 8.7% for the entire JPEG compression algorithm using double enhancement. The speedup improves to 13.6% and 16.2% when using parallelism of 4 and 8, respectively.

5. Conclusion

5.1 Summary

A technique to reuse high precision hardware to perform multiple low precision operations has been illustrated. This methodology requires only high level source code modifications, and will work on any machine that supports the IEEE floating point standard. This methodology is, in effect, a low cost way to get a SIMD architecture from a SISD workstation.

The above graphs seem to indicate that the dominant factor in obtainable speedup is the ratio of parallelizable work to non-parallelizable, D/K. Furthermore, once the D/K factor has been made larger, thereby shifting the operating point to the right, going to higher degrees of parallelism will have greater advantage.

Finally, it has been shown that H, the packing and unpacking overhead is not significantly large. Packing requires only a single operation in machines with a multiply-add instruction, and unpacking can be done in 3 operations using a multiply-subtract instruction. Since non-arithmetic instructions including branches, loads, and stores, as well as arithmetic operations are done in parallel, speedup results from both an increase in data bandwidth, and an increase in arithmetic speed.

5.2 Future Work

Because the major limit in performance comes from the D/K ratio, it must be made easier to perform longer stretches of code in parallel, in order to make D larger. One major obstacle to this currently is the care that must be taken in selecting the length of multiplication constants and number of multiplies in each data path. A slightly modified multiplier could ensure that the result will be the correct length, thereby allowing an unlimited number of multiplies in succession.

Another approach to the above problem might be to add a separate reset instruction. Although this would add additional instructions, it would have the advantage of not slowing down the multiplier.

Finally, by building overflow detection into the hardware, the need to know the absolute maximum values of the data would be eliminated, and perhaps more general algorithms could be implemented.

---

[1]R.B. Lee, "Precision Architecture," *IEEE Computer*, Vol. 22, No. 1, June 1989.

[2]R.K. Montoye, E. Hokenek, and S.L. Runyon, "Design of the IBM RISC system/6000 floating-point execution unit," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 59-70, Jan. 1990.

[3]K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, Applications*, Academic Press, Inc., San Diego, CA, 1990.

[4]C. Loeffler, A. Ligtenberg, G. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications," Proceedings International Conference on Acoustics, Speech, and Signal Processing, 1989.