2001-12

FFT-based spectrum analysis using a Digital Signal Processor.

Dorcey, Charles T.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

FFT-BASED SPECTRUM ANALYSIS USING A DIGITAL SIGNAL PROCESSOR

by

Charles T. Dorcey
December 2001

Thesis Advisor:	Herschel H. Loomis, Jr.
Second Reader:	Jon Butler

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 2001	Master's Thesis

4. TITLE AND SUBTITLE
FFT-Based Spectrum Analysis using a Digital Signal Processor

5. FUNDING NUMBERS

6. AUTHOR(S)
Dorcey, Charles T.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A spectrum analyzer based on Fast Fourier Transform (FFT) techniques was implemented using the TMS320C6201 Digital Signal Processor device manufactured by Texas Instruments. Portable C programs demonstrated optimization of the FFT algorithm for maximum speed. Previously published algorithms were adapted to the unique features of this Very-Long Instruction Word (VLIW) parallel processor and application, taking into account fixed-point arithmetic, parallel operation of functional units, and a hierarchy of memory capacities and speeds. The effectiveness of the VLIW C compiler, with automatic optimization, is compared with an explicitly-scheduled assembly-language program. The resulting program was then used to demonstrate the crucial need to keep program data in the Internal Data Memory to preserve hard-won performance gains.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Spectrum Analysis, Fast Fourier Transform, VLIW (Very Long Instruction Word), cache performance, finite-precision, signal processing	80
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

FFT-BASED SPECTRUM ANALYSIS USING A DIGITAL SIGNAL PROCESSOR

Charles T. Dorcey

Civilian, Department of Defense

BSEE, Michigan State University, 1981

Submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

December 2001

Author: Charles T. Dorcey

Charles T. Dorcey

Approved by:

Herschel H. Loomis, Jr., Thesis Advisor

Gerald M. Stum, Co-Advisor

Jon Butler, Second Reader

Jeffrey B. Knorr, Chairman

Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A spectrum analyzer based on Fast Fourier Transform (FFT) techniques was implemented using the TMS320C6201 Digital Signal Processor device manufactured by Texas Instruments. Portable C programs demonstrated optimization of the FFT algorithm for maximum speed on a general-purpose processor. Previously published algorithms were then adapted to the unique features of this Very-Long Instruction Word (VLIW) parallel processor and and performance requirements of this application, taking into account fixed-point arithmetic, parallel operation of functional units, and a hierarchy of memory capacities and speeds. The effectiveness of the VLIW C compiler, with automatic optimization, is compared with an explicitly-scheduled assembly-language program. The resulting program was then used to demonstrate the crucial need to keep program data in the Internal Data Memory to preserve hard-won performance gains.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The problem: Monitor the radio-frequency environment in a region of interest for unauthorized transmissions from unknown sources given constraints on system size and cost.

The approach: Design a spectrum analyzer based on the Fast Fourier Transform (FFT), and implement the FFT as efficiently as possible. Compare general-purpose computers (SGI R4000 and Intel Pentium-3) with a specialized Digital Signal Processor (DSP, the Texas Instruments TMS320C6201), deriving efficient algorithms for this specific application.

The major results:

1. 16-bit integer arithmetic was found to be adequate for implementation of the FFT in this application, as long as partial results are appropriately scaled to prevent arithmetic overflow. (An integer FFT algorithm which includes scaling of partial results is original work.)

2. The VonHann window function, used in the Welch method of averaged periodograms for spectrum estimation, provides adequate spectrum estimation accuracy (significantly better than the popular Hamming window), and can be derived as needed from the tabulated sine and cosine factors used in the FFT.

3. For FFT data sets which cannot fit in the cache of a general-purpose computer (or in the Internal Data RAM of the DSP) relying on automatic memory management to provide data to the FFT leads to a dramatic increase in the run time. When computing a 1048576-point transform on the RISC processor, for example, the processor is idle waiting for cache updates 80% of the time. An algorithm which factors 1048576 into 1024 transforms of 1024 points each recovers most of this idle time, running in less than one third the time of the original algorithm. While this algorithm has been published in Fortran, the ANSI-C implementation (Appendix, part C) is original work. The comparison between actual run times and run times extrapolated from small data sets sizes, to assess cache effectiveness, is also original.

4. The algorithm which factors a large transform into a sequence of smaller transforms suggests a scheme for computing a large transform on a massively parallel processor, though this was not implemented. This scheme would be particularly useful when samples are acquired at a rate which exceeds the input bandwidth of any single processor's memory.

5. The “factored FFT” algorithm relies on a matrix-transposition routine which also allows efficient management of the processor cache. This ANSI-C program (Appendix, part B) is original work.

6. Software development for the DSP environment was found to be more difficult than for the general-purpose computing environment, as described below.

Starting with a published FFT algorithm, improvements to the ANSI-C source code (Appendix, part A) reduced the run-time for a 4096-point FFT on the R4000 from 23 msec to 11 msec . After converting from floating-point to integer arithmetic, the function ran in 12 msec on the DSP with all compiler optimizations disabled. Enabling all compiler optimizations reduced the run time to 1.4 msec, yet the expert-optimized assembly language version ran in just 0.40 msec. We interpret this to mean that approaching the advertised performance on complex algorithms requires expert programming at the assembly language level. (Such algorithms may be provided in off-the-shelf libraries, though.) Modifications to the optimized assembly language to perform intermediate result scaling is original work. The use of a “financial spreadsheet” (e.g., Excel, Gnumeric) for scheduling parallel processor operations is also original work.

On the Pentium-3, portable ANSI-C code ran in 0.79 msec, and Intel’s optimized library code ran in 0.32 msec. We interpret this to mean that Intel’s C compiler comes closer to achieving peak performance than the DSP compiler does. (Note that the 100 MHz R4000 processor is at least five years older than the 733 MHz Pentium-3; this is not “a fair race” in absolute terms.)

7. Comparing general-purpose processors, we find that the Pentium-3 system is at least twice as fast as the RISC R4000 *after* compensating for the difference in clock rates, which we attribute to architectural differences. The Pentium-3 is over four times as fast for the 1048576-point transform, which reflects a faster memory system.

8. Comparing the Pentium-3 and the DSP, we find that the Pentium-3 was slightly faster in completing a 4096-point floating-point transform than the DSP was in computing the integer transform. Though the Pentium-3 processor is more expensive than the DSP, the enormous volume of systems which incorporate the Pentium-3 has driven the system price (Compaq Proliant) to a fraction of the price of the DSP system (Pentek 4290). On the other hand, specialized signal processing peripheral devices which are required to provide sampled data to the processor (e.g., a radio receiver with an 8 MegSample/sec output path) are simply unavailable for the Intel architecture.

Though the DSP has a clear advantage in high-volume markets for highly-integrated systems (e.g., modems), developers of unique systems for niche markets must carefully eval-

uate the current state of commercial products in the context of their application to get the best configuration.

Significance: The work described in this thesis has advanced our development of two spectrum analysis instruments. This thesis may provide useful guidance to others, especially to those working with FFTs of more than 65536 data points.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The Fourier Transform is one of the fundamental tools of electrical engineering. Its sampled-data version, the Discrete Fourier Transform (DFT), is crucial to several areas of digital signal processing (DSP). It is used for digital filter design and implementation, time-delay estimation, image compression coding, and spectrum analysis.

Spectrum analysis is the context of this thesis. Spectrum conflict management, signals intelligence, technical security, space probe telemetry, and the (thus far incomplete) Search for ExtraTerrestrial Intelligence (SETI) all rely on detecting the presence of radio signals of unknown frequency, power, and modulation.

Though the general topic of spectrum estimation is still an area of research, the work described in this thesis considers only Welch's method of averaged modified periodograms using the Fast Fourier Transform (FFT) [Ref. 1: p.553]. Since the popularization of the FFT algorithm for computing the DFT by J. W. Cooley and J. W. Tukey in the mid-1960s, numerous researchers have studied ways to compute it as quickly as possible with the technology of the time. Innovations in computer architecture have enabled evolution of FFT algorithms.

Digital signal processing (DSP) can be done with general purpose computers, but computer architectures optimized for digital signal processing have been implemented in microprocessor form for almost two decades. Most DSP algorithms rely heavily on multiplication and addition, so early DSP devices dedicated a substantial fraction of their chip area to single-cycle multiplication hardware. General-purpose microprocessors of that time performed multiplication through shifting and adding (sometimes implemented in microcode, but other times left as an exercise for the programmer). General purpose computers (and microprocessors) almost always fetch both programs and data from a unified memory subsystem (the Von Neumann architecture) [Ref. 2:p. 24].

The Harvard architecture, on the other hand, provides four separate buses: program address, data address, instruction, and data [Ref. 2:p. 200]. This increases the rate at which sampled signals can flow through the processor, at the expense of added complexity and

lost flexibility (a small program which manipulates a large data set may not fit into a system designed for a large program which manipulates a small data set).

Major manufacturers of DSP devices are Motorola (56800 family), Analog Devices (SHARC family), and Texas Instruments (TMS320 family), among others. The TMS320C6201 DSP device was selected for use in the development project supporting this thesis since it was readily available. We shall focus on algorithm optimizations which may be appropriate for this device.

In the development of a new signal detection system, we considered "how can DSP devices be efficiently used for spectrum analysis?" This thesis explores a variety of design issues in the development of an FFT-based spectrum analyzer. Initially, we describe variations on the theme of FFT algorithm implementation, and show how the run time of a "textbook" algorithm can be reduced by a factor of eight while preserving the portability of C language. Then we describe design choices for spectrum analysis window implementation and optimization of an FFT algorithm for the TMS320C6201 DSP device manufactured by Texas Instruments Incorporated (TI). (This processor and related products from TI are referred to below simply as the "C6x" where such usage does not cause confusion.)

Though FFT algorithms have been derived for data vectors of arbitrary length, the FFT algorithms described in this thesis are restricted to those which process data vectors of N elements, where $N=2^k$, with k an integer.

The notation "1K" refers to $1024 = 2^{10}$, and "$1K^2$" refers to $1024^2 = 2^{20} = 1048576$. Let i denote $\sqrt{-1}$.

II. THE FAST FOURIER TRANSFORM DESIGN SPACE

This chapter explores FFT algorithm design issues which apply to any implementation, whether general-purpose computer, digital signal processor, or custom hardware. Such issues include minimizing the number of arithmetic operations, trading fast addition for slow multiplication, minimizing the memory space needed, opportunities to trade memory space for execution speed, and optimizing cache memory efficiency.

A. ARITHMETIC

The DFT and Inverse DFT are defined by a pair of mathematical formulae [Ref. 3:p. 406, 407] which can be translated into arithmetic operations (multiplication and addition) in a straightforward way.

$$F_k = \sum_{j=0}^{N-1} f_j e^{i2\pi jk/N} \qquad (2.1a)$$

$$f_j = \frac{1}{N} \cdot \sum_{k=0}^{N-1} F_k \cdot e^{-i2\pi jk/N} \qquad (2.1b)$$

For the purposes of this paper, f_j can be regarded as a complex-valued sampled time series of length N, F_k as a complex output sample from the k-th bandpass filter, $e^{i2\pi jk/N}$ as a rotation in the complex plane which is proportional to time (j) and frequency (k). Equation 2.1a is the "Forward" DFT; equation 2.1b, the Inverse. Arfken notes that the equations can be made symmetrical by distributing the "1/N" factor shown in 2.1b across both equations as $1/(\sqrt{N})$ [Ref. 4:p. 789].

Unfortunately, the straightforward algorithm suggested by equation 2.1a requires N complex multiplications and additions for each of the N output values, so the number of arithmetic operations is proportional to N^2 for the complete transform. All "fast" algorithms are roughly proportional to $N*\log_2(N)$, eliminating approximately 99% of the work for a 1K-point transform, and 99.998% of the work for a $1K^2$-point transform. However, the constant of proportionality can vary significantly with implementation.

1. Minimizing Complex Multiplications

The heart of the FFT is equation 2.2, which illustrates the radix-two algorithm [Ref. 3:p. 408]. We compute two transforms of size $N/2$ (using even indexed samples for

$$F_k = F_{even,\,k} + e^{i2\pi k/N} F_{odd,\,k} \qquad (2.2)$$

one, odd indexed samples for the other), combined into one transform of size N. Computation of F_{even} and F_{odd} can be accomplished by computing four transforms of size $N/4$, and so on, until the transform size is reduced to one (which is no transform at all). Each stage, such as the one above, requires $N/2$ complex multiplications and additions (one for each odd value of k), and there will be $\log_2(N)$ such stages. Thus, the transform of size N is computed with approximately $N*\log_2(N)/2$ complex multiplications. This is illustrated below for $N = 8$. [Ref. 1:p. 300]. Each stage has four complex multiplications (the multipliers being denoted as W_N^k), and there are three stages.

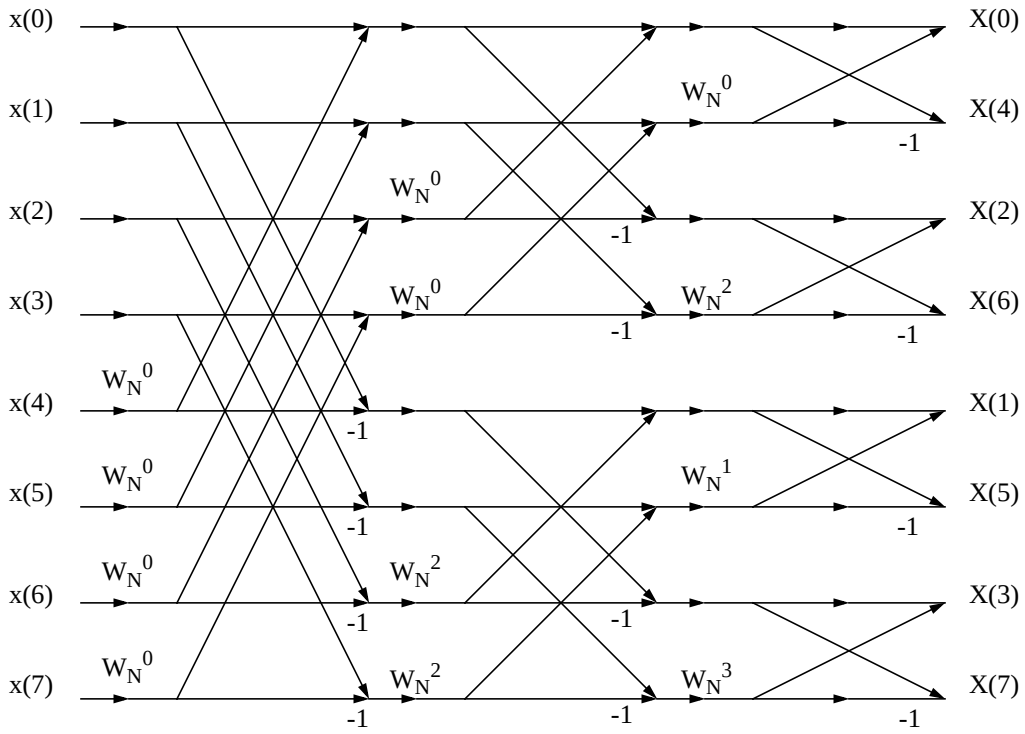

Figure 1. Eight-point FFT Flow Graph.

2. Minimizing Real Multiplications

Whether or not the programming language used for an FFT algorithm supports complex numbers, complex multiplication must eventually be implemented with real arithmetic. The term $e^{i2\pi jk/N}$ (W_N^{jk}) becomes (Cos(2 π *j k* / *N*) + *i* Sin(2 π *j k* / *N*)); x_n becomes (Re{x_n} + *i* Im{x_n}) (the sums and multiplications by *i* being complex number notation, of the form "a + *i* b", rather than actual arithmetic).

a. Alternative Expressions for Complex Multiplication

The conventional way to calculate a complex product with real arithmetic is shown in Equation 2.3.

$$(a + ib) \cdot (c + id) = (a \cdot c - b \cdot d) + i(b \cdot c + a \cdot d) \qquad (2.3)$$

However, when multiplication is more time consuming than addition, an alternative form may be advantageous.[Ref. 5:p. 430].

$$(a + ib) \cdot (c + id) = (((\underline{a + b) \cdot c}) - b \cdot (d + c)) + i(((\underline{a + b) \cdot c}) + a \cdot (d - c)) \quad (2.4)$$

The underlined common subexpression in Equation 2.4 reduces the number of multiplications from four to three, at the cost of increasing the number of add/subtract operations from two to five. This may a worthwhile change if multiplication takes more than twice as long as addition. From a parallel pipelined processing perspective, the conventional expression can complete in two stages (if four multiplication units are available), while the alternate expression requires at least three stages, but only three multiplication units. Thus, deciding which code will run more quickly requires a detailed knowledge of the processor resources.

b. Radix-four Algorithms

When $N = 4^k$, a radix-four algorithm will be slightly more efficient than the radix-two algorithm sketched above. Instead of dividing the input vector into two sub-sequences, it is divided into four sub-sequences, so only $\log_4(N)$ stages of processing are needed. This can reduce the amount of data traffic between the processor registers (or cache) and data vector memory by half, assuming that the processor has enough registers to keep intermediate results close at hand. Multiplication by *i* can be implemented by sim-

ply interchanging real and imaginary components, then negating the real component, and W_N^0 is always equal to one so the number of arithmetic multiplications is reduced. A flow-graph for a radix-four calculation is shown below [Ref 1:p. 317].

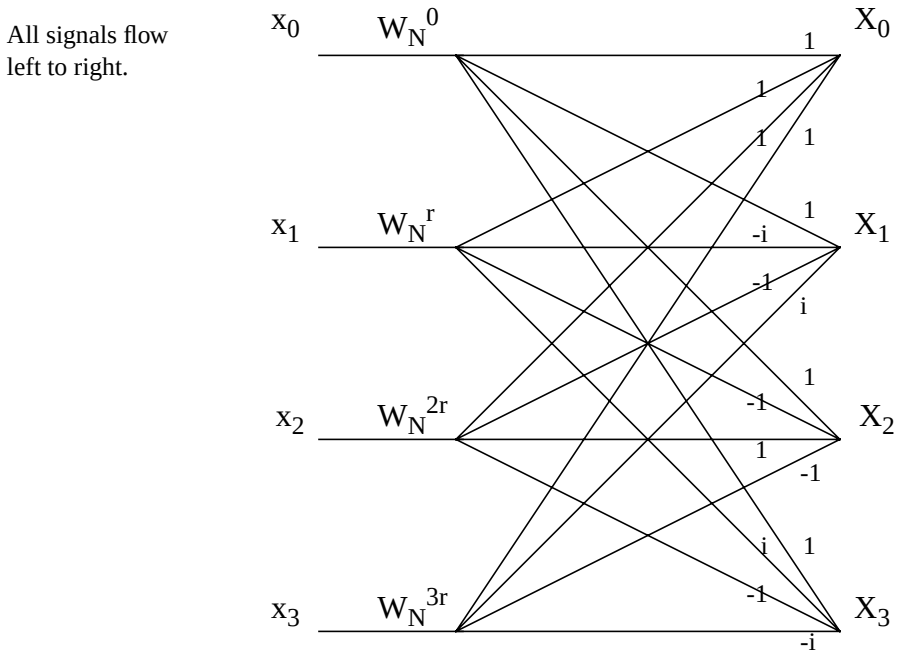

Figure 2. Four-point, Radix-four FFT.

3. Complex Exponential Constants

The "*N* complex roots of unity" constant factors $e^{i\,2\,\pi\,j\,k\,/\,N}$, typically denoted as W_N^{jk} and refered to as "twiddle factors", offer a variety of options to the algorithm designer. To call a math library function each time a constant is needed is simple and accurate, but slow. In $e^{i\,2\,\pi\,j\,k\,/\,N}$, *2*, π, *i*, and *N* are constants, and *j* and *k* are integers less than *N*, so the robust generality of a math library call is rarely needed.

One option is to note that all of these factors are W_N raised to an integer power, and can be used in ascending order. Once W_N is known (e.g., provided by a math library func-

tion), successive constants can be obtained iteratively, as in the following pseudo-code fragment [Ref. 6:p. 24]:

```
complex w_increment = exp(i 2 π / N);
complex w_jk = (1, 0);   /* w to the zeroth power is 1. */
for k = 0 to M
   (use w_jk for an FFT calculation)
   w_jk = w_jk * w_increment; /* recursively update w */
```

Figure 3. Recursive Update of Complex Exponential Constants.

Now only one library function call (or tabulated value) is needed for all *M* values. However, the recursive formula allows numerical rounding error to accumulate proportionally to the transform size (especially if fixed-point arithmetic is used).

If many transforms of some particular size *N* are to be computed (as they will be in a spectrum analyzer), we can pre-compute the whole set of constants $\text{Cos}(2 \pi j k / N)$ and $\text{Sin}(2 \pi j k / N)$ just once. This occurs when the algorithm is initialized, although the program then requires additional memory for the table. If the algorithm is being designed for a predetermined value of *N*, the constant tabulation can be done when the program is compiled.

How many values need to be tabulated? One way to answer this question would be to look at the entire algorithm to determine the maximum product of *j* and *k*, but this would give a pessimistic result. Since Sin(*x*) and Cos(*x*) are periodic functions, if we can map all products of *j* and *k* into the interval [0..*N*-1], we only need to tabulate $\text{Cos}(2 \pi k / N)$ for *k*=0 to *N*-1. Since $\text{Cos}(2\pi\text{-}x)=\text{Cos}(x)$, $\text{Sin}(x+\pi/2) = \text{Cos}(x)$, and so on, we may be able to minimize the use of memory by carefully indexing into a smaller table. On the other hand, complicated indexing logic may impose an intolerable burden on the arithmetic processor, especially if it involves time-consuming branches in the control flow.

With what precision do the values need to be tabulated (or calculated)? Floating-point arithmetic typically provides 24 or 48 bits of precision. Integer arithmetic, as used by many DSP devices, could plausibly use 8, 16, and 32 bits. Using the rough rule-of-thumb that each bit of precision provides six decibels of dynamic range, 8-bit values

would add quantization noise at the -48 dB (relative to full scale) level, 16-bit values provide -96 dB, and 32-bit values provide -192 dB. For our application, -48 dB would be excessively noisy, but -96 dB is sufficient. Thus, either 16-bit integer ("`short`" in ANSI-C) or single-precision floating-point were acceptable.

As we will see in Chapter III, using 16-bit integer Cos() and Sin() constants allows them to be interleaved in memory such that both can be loaded into a 32-bit register with a single instruction, and makes efficient use of the DSP device's 16x16 multiplication unit. Tabulating one entire cycle (*N* values) for each function allows simple (fast) data indexing, although it requires eight times as much memory as the more complex indexing scheme described above.

4. Fixed-point vs.Floating-point Data Representation

The hardware for floating-point arithmetic is inherently more complicated than that for integer arithmetic. To perform floating-point addition, the exponent terms of each value must be made equal, the mantissa shifted as appropriate, the mantissas added, then the result normalized. In multiplication, the mantissas are multiplied and the exponents added, after which limited (one or two bits) re-normalization is needed [Ref. 2:p. 296].

Analysis of data representation parallels that of coefficient precision. Though sensor data may be of only eight or twelve bits, 16-bit integers are efficiently processed and provide adequate accuracy in our application.

We can contrast the arithmetic performance of two processors from TI's C6x family. The TMS320C6201 does only integer arithmetic, while the TMS320C6701 also does floating point. These devices have otherwise identical architectures and process technologies. The 16x16 bit integer multiply completes in two clock cycles, while the 32x32 bit floating-point multiply completes in four clock cycles. In each case, a new instruction can start the multiplier pipeline on every clock cycle, but the added latency of the floating-point operation increases the likelihood that algorithm dependencies will prevent continuous utilization of the functional unit. Furthermore, the clock period (at introduction in 1997) of the 'C6201 was five nanoseconds, while that of the 'C6701 was six [Ref. 7: Mod. 1: p. 37]. (We'll discuss the 'C6x architecture in more detail in Chapter III.)

5. Integer Result Scaling

The DFT algorithm in the formula above consists of a series of complex multiplications and additions, which implies that the maximum magnitude of any output value may be as large as *N* times the maximum allowable input value. (A typical worst-case input vector has all values equal to *max + i max*. F_0 is then just *N*(max + i max)*.) If a fixed-point data representation is used, the output word may require $log_2(N)$ more bits than the input word to contain the increased magnitude. To be specific, the 4K-sample transform requires 12 more bits, the $1K^2$ transform 20 more bits, and the $4K^2$ point transform 22 more bits. Unfortunately, our 'C6x processor can most efficiently multiply 16-bit operands, so overflow avoidance would seem to require scaling its input data to just four bits for the smallest transform of interest and would make the larger sizes infeasible.

This problem can be addressed by modifying the DFT equation as follows:

$$F_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{i2\pi jk/N} \cdot f_j \qquad (2.5)$$

or

$$F_k = \frac{1}{N} \sum_{j=0}^{N-1} e^{i2\pi jk/N} \cdot f_j \qquad (2.6)$$

We can compensate for these scaling constants when interpreting the spectrum analyzer output, and they allow us to distribute the scaling of intermediate results across the stages of the FFT. (If forward transforms were followed by inverse transforms, a compensating change would be needed to the IDFT definition.) Since $N = 2^k$, and we have *k* stages of processing, we can implement the second equation above by dividing the result of each stage of a radix-two FFT's addition by two, which is simply a one-bit right shift. Alternatively, the summands to the addition can be scaled. When the summands are formed by an integer multiplication, the scaling that renormalizes the multiplication result can be modified to incorporate summand scaling with no performance penalty.

With a radix-four algorithm, the designer has more flexibility. Four terms are added together at each stage, which produces a potential word-length growth of two bits

per stage. Right-shifting by two at each stage prevents overflow (as in Eq. 2.6), while right-shifting by one (as in Eq. 2.5) allows word-length growth of log4(*N*). If the input data is acquired with eight-bit precision, the number of significant bits can be allowed to grow to fifteen (not including the sign). Without intermediate term scaling, the maximum FFT size without overflow would be 128. With single-bit intermediate scaling, the maximum size is 32K; with two-bit intermediate scaling, overflow is impossible.

Bear in mind, however, that the dynamic range of the output is still limited to at most 16 bits. Consider the fate of an input data vector containing a single non-zero sample, at index 0. Without scaling, the non-zero value is duplicated to every element in the output vector. With single-bit scaling, its value is reduced by half at each stage, so a value of 128 vanishes after seven radix-four stages (16K-point FFT). With two-bit scaling, it disappears in the fourth radix-four stage (256-point FFT).

Whether or not distributed scaling is appropriate will depend on the application. In some cases, it might be feasible to examine the result of applying conservative two-bit scaling; if no spectral lines are found which have sufficient magnitude to cause overflow, apply a transform with one-bit scaling, or no scaling at all, to the same data set to achieve greater accuracy. If the processor has an overflow flag which is updated with the status of every addition (the ‘C60 does not), responding to an overflow condition would require either time-consuming instructions to test the flag or a hardware interrupt to modify the flow of the algorithm.

B. MEMORY

As arithmetic logic technology has increased in speed and complexity, the time required to move data between memory and arithmetic units has become increasingly significant. Regardless of advancing technology, relatively fast memory is relatively expensive, and many algorithms (with the notable exception of large FFTs) have been found to require access to a small fraction of the total memory for much of the total time. Thus, most computer memory is organized in multiple levels. The processor’s register file memory provides the most rapid transfers to and from arithmetic logic. Frequently used data which cannot reside in the register file resides in a small, fast cache memory which han-

dles most of the accesses. It, in turn, is supported by a larger, slower “main memory,” which may be supported by virtual memory on a local disk, a remote “file server,” and/or archival magnetic tape [Ref. 2:p. 372]. In this section, we assess the impact of the memory hierarchy on FFT performance.

1. In-place vs. Out-of-place Algorithms

The most commonly presented FFT algorithms minimize the use of memory by operating “in place”; that is, out of *N* complex elements in the input vector, arithmetic is performed using two (or four, depending on the radix) of them, after which the modified values are written back into the same memory locations. However, this has the disadvantage of leaving the output vector “scrambled,” so most applications require an additional pass through memory data, after the transform proper is complete, to unscramble the result. Rearrangement of the FFT flow graph can produce an algorithm that returns the output elements in the proper order, although the algorithm requires two memory buffers of length *N* instead of one. Memory address calculations during the transform may require slightly more time, but the total execution time is reduced, since the unscrambling operation is unnecessary [Ref. 6:p. 49].

2. Window Function Storage

Spectrum estimation with the Welch method of averaged periodograms requires smoothing in the frequency domain, implemented by multiplying the input sequence by a “window function.” In the time domain, a window function tapers the magnitude of data elements near the ends of the array. One popular window function, the “periodic Hamming window,” was invented by R.W. Hamming: $h(k) = 0.54 - 0.46 \cos(2 \pi k / N)$, and there are a number of variations on this theme. For example, Oppenheim and Shafer [Ref. 1: p.242] give the following formula: $h(k) = 0.54 - 0.46 \cos(2 \pi k / (N-1))$. This is the “symmetric” Hamming window, used for digital filter design, which gives subtly different spectrum estimates. (As stated by the authors of *Numerical Recipes in C* in a slightly different context, “... if the difference between *N* and *N*-1 ever matters to you, then you are probably up to no good anyway....” [Ref. 3:p. 473].) Selection of a window function involves balancing the frequency-resolution of spectrum estimates, accuracy of power measurements, sup-

pression of sidelobe power which can mask weak signals, and computational complexity [Ref. 8:p.161].

For fastest processing performance, the window function should be tabulated, just as the Sine and Cosine factors used within the FFT are. The simplest scheme for tabulating the window function is to precompute all N values. The amount of memory needed, though, can be cut in half by taking advantage of the symmetry of the function. Depending on the balance between memory and arithmetic speed in the system, it may be possible to eliminate window-function storage memory by re-using the $\text{Cos}(2 \pi k / N)$ function tabulated for use within the FFT itself. If 16-bit integer arithmetic is used, the cosine table will be scaled to fill the word: $cos\_table[k] = 32767 * \text{Cos}(2 \pi k/N)$. The integer-scaled Hann window can be easily derived from this table: $h[k] = 16384 - (cos\_table[k] >> 1)$, where ">>" is the ANSI-C "right-shift" operator. The Hann window is also attractive because it provides lower spectral leakage far from a strong spectral line than the Hamming window does, at the expense of a slightly higher close-in sidelobe, as shown in Fig. 4. The signal to be analyzed consists of three sinusoids, of randomly selected frequency and unequal power. The weak signals are nearly hidden by the leakage of the Rectangular and Hamming windows.

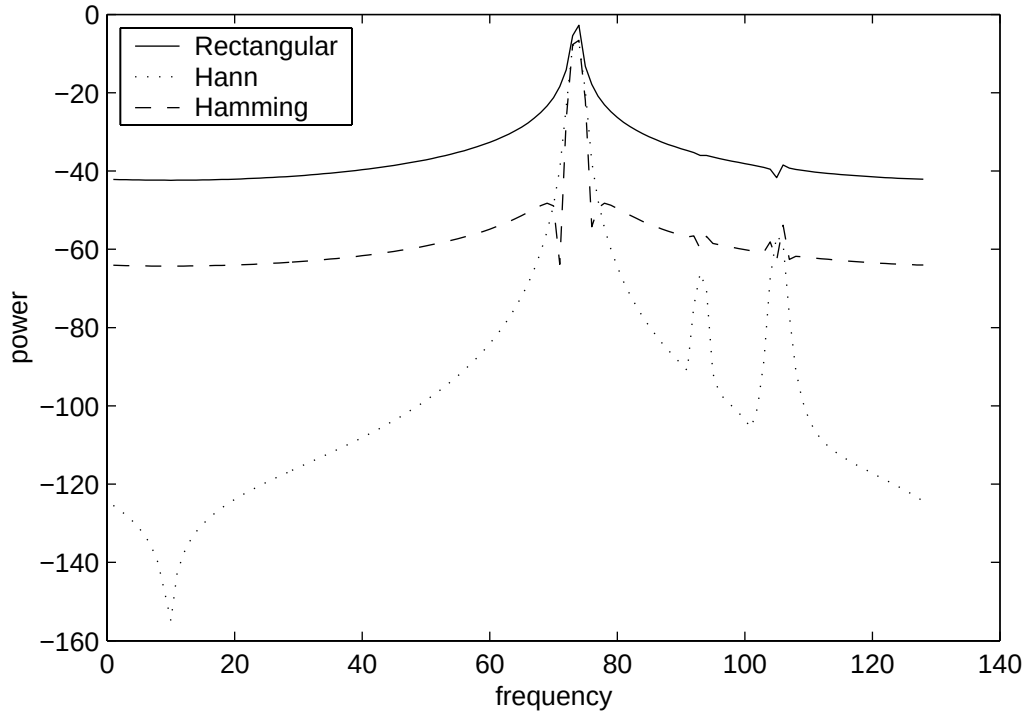

Figure 4. Spectrum Leakage for Three Window Functions.

If one of the more complex window functions was selected (Kaiser-Bessel or Dolph-Chebyshev, for example [Ref. 8:p.194]), we could conserve memory by tabulating a decimated window function, then interpolating it as needed. Since window functions are relatively smooth, we need only tabulate the even-indexed elements, and “hold” each tabulated value for the following odd data element (the simplest possible interpolation). If we do this, we’ll find that spurious signals appear within the output spectrum estimate as shown in Fig. 5, for the following reason. Consider the interpolated window function as the sum of the true window function and an error function. The error function will be zero for even elements, but will be non-zero and proportional to the first derivative of the window function for odd elements. Thus, the spectrum of the error function will contain a discrete component at one half of the sample rate. Multiplication in the time domain corresponding to convolution in the frequency domain, we find that the spectrum estimates produced with decimated window functions are convolved with the spectra of both the true

window function and the error function. Spurious spectral lines are the result, as shown at the left side of Fig. 5. (The three-sinusoid input signal is the same as for the prior figure.)

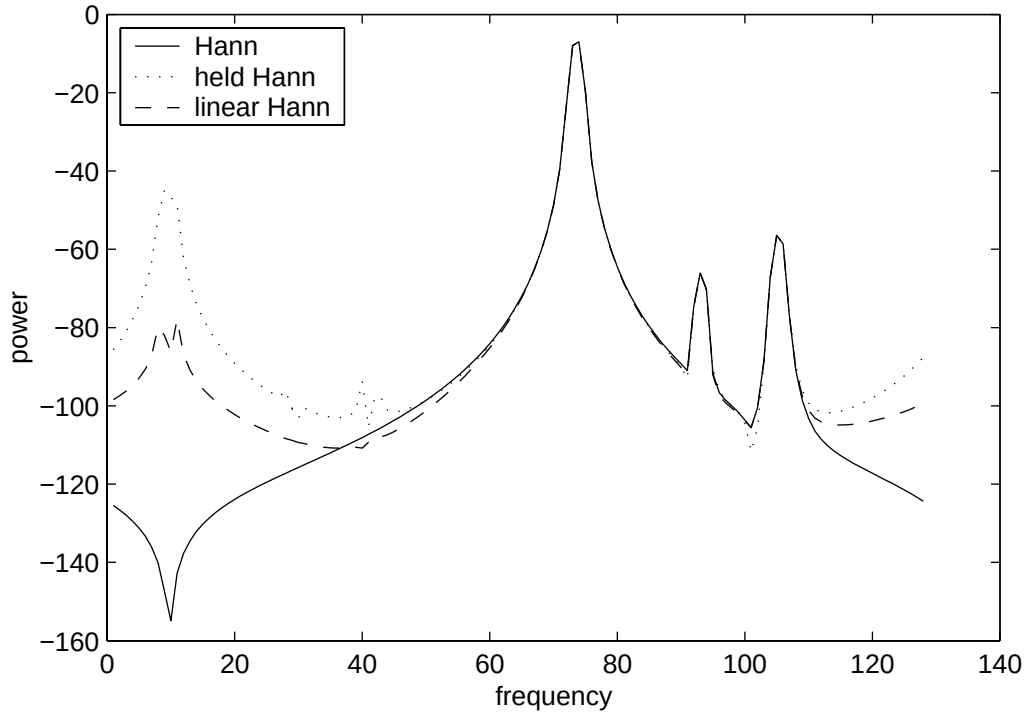

Figure 5. Decimated Window Functions Produce Spurious Spectra.

Quantitatively, the relative power of the spurious signal depends on the size of the FFT. As the size increases, the magnitude of the first derivative, and thus the amplitude of the error function, decreases. The spurious signal is 38 dB below the true signal for N=128, and it falls by 6 dB each time N is doubled.

If we use linear interpolation, instead of holding the prior value, the error function will be proportional to the second derivative of the window function. The spurious signal is then 71 dB below the true signal for N=128, and falls by 12 dB each time N is doubled. Trading off window storage space, interpolation complexity, and spurious signal levels can be done in the context of a specific application.

3. Unscrambling the Output

An in-place, decimation-in-frequency algorithm produces a scrambled output vector. That is, the transformed elements are arranged in memory as if sorted by a key which is the element index, reversed such that the most significant bit is in the least significant position. For a 256-point radix-two algorithm, element 0 will be in memory location 0, but element 1 (binary 0000 0001) will be in location 128 (1000 0000), element 2 (0000 0010) will be in location 64 (0100 0000), element 3 (0000 0011) in location 192 (1100 0000), and so on. Since we want to observe features of the signal spectrum which span more than a single FFT output value, we need an efficient way to re-order the data by frequency index.

Consider the data re-ordering algorithm for a radix-two algorithm. The entire vector can be re-ordered by incrementing the index through the vector, comparing the index with its bit-reversed value, and swapping data elements when the reversed index is larger than the original. (When the reversed value is smaller, the swap has already been done for this pair.)

Gutman [Ref. 9] describes an elegant algorithm for reversing bits, which can be coded in C for a 16-bit word ("k") as follows:

```
a = ((k & 0x00FF) << 8) | ((k >> 8) & 0x00FF);
b = ((a & 0x0F0F) << 4) | ((a >> 4) & 0x0F0F);
c = ((b & 0x3333) << 2) | ((b >> 2) & 0x3333);
d = ((c & 0x5555) << 1) | ((c >> 1) & 0x5555);
```

Figure 6. Reversing the Order of Bits in a Word.

In C, "&" is the bit-wise "and" operator, and "|" is bit-wise "or". The first line swaps the high byte with the low byte. The second swaps the four least significant bits with the four most significant bits in each byte, and so on, until the last swaps individual bits. Note that this algorithm is free of branch instructions and recursive assignments, and so a pipeline is effective in speeding up its operation.

For a radix-four algorithm, the bits of the index are swapped as two-bit digits. If we denote the eight-bit index as "$b_7b_6b_5b_4b_3b_2b_1b_0$," the scrambled index is

"$b_1b_0b_3b_2b_5b_4b_7b_6$" (not "$b_0b_1b_2b_3b_4b_5b_6b_7$," as for radix-two). To reverse digits as required for a radix-four algorithm, we can simply omit the last line of Fig. 6.

The algorithm is straightforward for index values which fill a power-of-two word size (i.e., 8 bits: N=256, 16 bits: N=64K), and can be extended for arbitrary lengths. For a N=2K FFT, the index will be 11 bits wide, and the following sequence can be used to reverse the bits:

```
a = ((k & 0x001F) << 6) | (k & 0x0020) | ((k >> 6) & 0x001F);
b = ((a & 0x00C3) << 3) | (a & 0x0124) | ((a >> 3) & 0x00C3);
c = ((b & 0x0249) << 1) | (a & 0x0124) | ((b >> 1) & 0x0249);
```

Figure 7. Reversing an 11-bit word.

Courtney, at Texas Instruments, has published a different algorithm for sorting FFT result vectors [Ref. 10], which relies on a lookup table to determine the bit-reversed index. When optimized in assembly language, it runs in approximately (N / 4) * 7 clock cycles on their VLIW processor.

4. Locality of Reference (Improving Cache Effectiveness)

The basic FFTs access memory according to a pattern which may prevent effective use of conventional computer cache memories. The signal flow graph for a 16-point, in-place, radix-four FFT is shown below in Fig. 8. There are two stages in this decimation-in-frequency algorithm, each consisting of four four-point FFTs. Each four-point FFT is referred to as a "butterfly" calculation (though perhaps they look more like spiders in this figure). The input vector is arranged sequentially in memory (m_i=x_i); the output vector is scrambled.

Input values to the first stage are read from scattered locations. In this 16-point transform, the first four samples to be processed come from locations 0, 4, 8, and 12; in a 4K-point transform, the first four samples would be 0, 1024, 2048, and 3072; for a $1K^2$-point transform, 0, 262144, 524288, and 786432. The cache management [Ref. 2:p. 344] assumption of "spatial locality," that most memory accesses tend to occur near recently used locations, is violated for all but the last few stages of a large transform. The second

cache management assumption, “temporal locality” (that memory which has recently been accessed is likely to be accessed again soon), is also violated. Once x_0, for example, has been read once and written once in stage 1, it won’t be read again until all other elements of the array have been read (once) and written (once) in stage 1. The impact of violating these assumptions will be demonstrated experimentally below. If the entire FFT data vector (and any tabulated constant factors) fits within the primary cache memory, the scattered access pattern does not affect the access time, but exceeding this size can have a significant impact on performance.

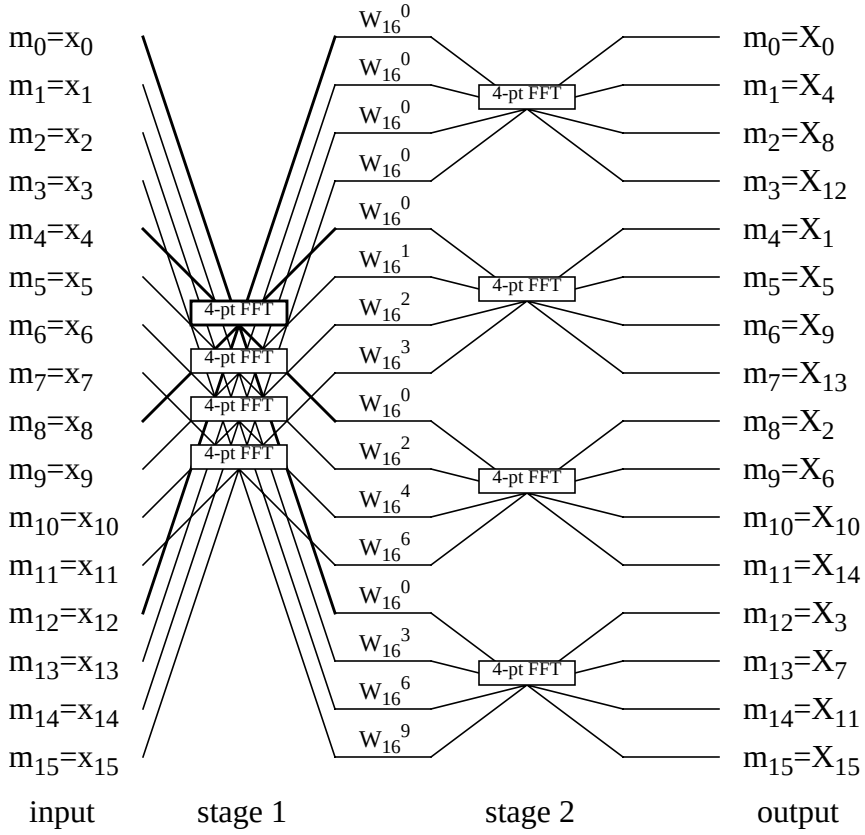

Figure 8. Memory Access Pattern, Highlighting the First Butterfly.

The solution to this problem is to decompose a large FFT into a series of smaller transforms, each of which can fit within the cache, and each of which is self-contained with respect to the rest of the data vector. To illustrate this idea, consider the 16-point data vector shown in Fig. 9, arranging the 16 input points into a four by four grid.

The first stage of the 16-point FFT in Fig. 8 applies a four-point FFT to each column of the left grid shown in Fig. 9. Then each element is multiplied by W_N^{jk} (where j and k represent the row and column indices), before a four-point FFT is applied to produce each row of the right grid in Fig. 9. Unscrambling of the data can be accomplished by reading down the columns of the right grid. This idea can be extended to much larger

sizes; a $1K^2$-point FFT can be decomposed into twenty stages of radix-two FFTs, ten stages of radix-four FFTs, or two stages of “radix-1K” FFTs.

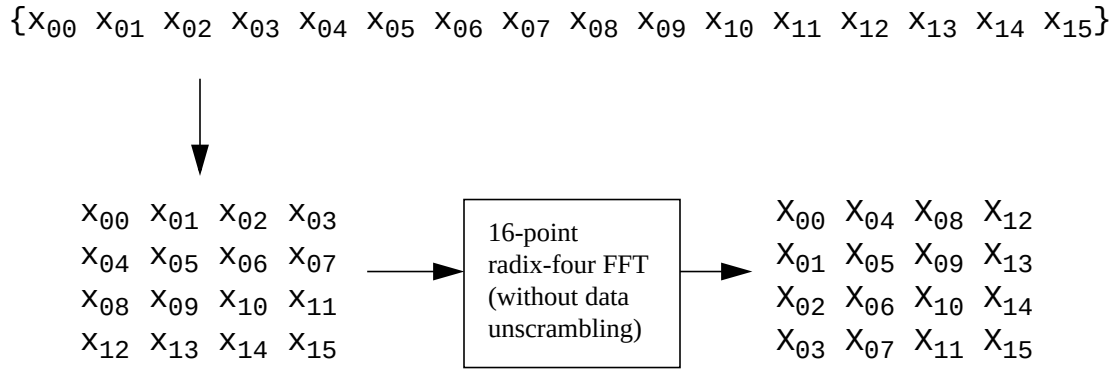

Figure 9. Reshaping the Data Vector.

Little would be gained if the small transforms operated on the widely-scattered data elements of a single column in main memory, but it is simple to “gather” these elements into a small work buffer, perform the transform, and then “scatter” the transformed elements back into their original positions (some vector processors have vector-gather and vector-scatter instructions). When all data for a single radix-1K FFT can fit within the primary cache of the processor, the radix-1K FFT can then be decomposed into five radix-four stages which run at the full processor speed.

Even greater efficiency can be achieved, however, if we transpose the entire data set before performing the small transforms [Ref. 6:p.139], as shown in Fig. 10.

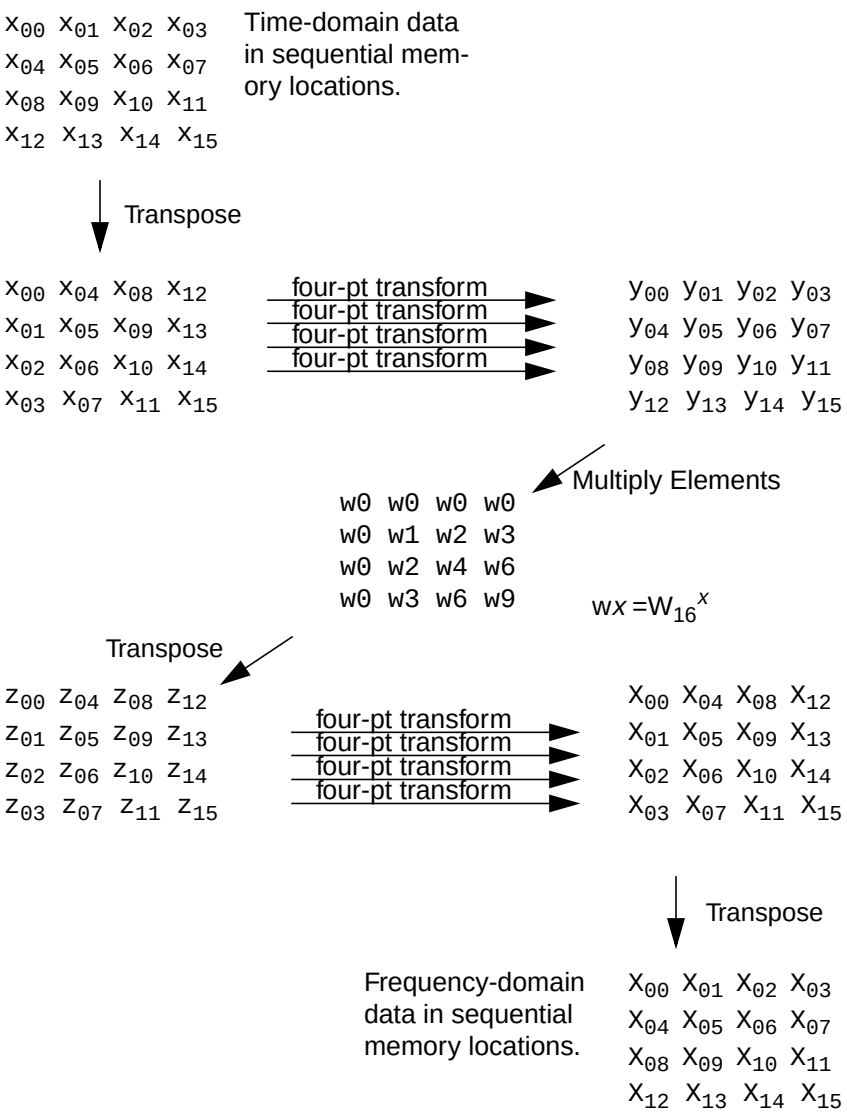

Figure 10. 16-point FFT Built from Four-point FFTs and Matrix Transposition.

5. Distributed Parallel Implementation

The algorithm described above suggests an algorithm for distributing computation on a multiprocessor architecture, since each of the row transforms can in principle be done by a separate processor. Consider a system architecture which must accept N input samples at a rate which may exceed the bandwidth of a single memory bus. High-speed logic can distribute samples to B memory buses, which implicitly performs the first matrix transpose operation. Once all N samples for a given observation interval are stored in memory, each of the B processors simultaneously performs an FFT of size N/B. Then, an interprocessor matrix transpose is followed by the twiddle-factor multiply and $N/(B^2)$ transforms of size B on each of the B processors [Ref. 6:p.173]. The result can then be read from the B memory systems in round-robin fashion to implicitly perform the final transpose.

As an approach to increase the bandwidth of a spectrum analyzer, the fast-sampling distributed FFT can be contrasted with a frequency-domain “divide and conquer” approach. Rather than distributing samples from a single analog-to-digital converter (ADC) across multiple processors, we can use a channelizing architecture to distribute subranges of the input bandwidth to independent spectrum analyzers. Channelizing can be performed with traditional analog electronics (oscillators, mixers, and filters) followed by a relatively slow ADC, or with one or more relatively fast ADCs followed by digital oscillators, mixers, and filters in hardware or software (as on the Pentek 6216 Dual Digital Receiver Module [Ref. 11]).

The channelizing architecture allows each processor to operate independently, without the synchronization and communication complexities of the distributed FFT implementation. However, the fast-sampling distributed FFT architecture may be favored if we also wish to analyze in the time domain those detected signals which may occupy a large fraction of the searched bandwidth.

6. Cache-efficient Transpose Operation

The algorithm described above transposes the full data set three times. The simplest way to perform such a transpose is with the following code, in which the two-dimensional character of the array is implicit in the read and write index calculations:

```
for (read_row = 0; read_row < read_rows; read_row++) {
  for (read_col = 0; read_col < read_cols; read_col++) {
     write_col = read_row;
     write_row = read_col;
     y[write_row * read_rows + write_col] =
        x[read_row * read_cols + read_col];
  }
}
```

Figure 11. Simple Transpose Source Code.

This algorithm allows the cache controller to load multiple elements from sequential memory locations, so the read miss rate is low. However, write operations are scattered throughout the result array, so a new cache line must be read, updated, and written back for almost every result. This inefficiency can be avoided by transposing blocks of elements [Ref. 6:p. 129]. When a block transpose of 16 by 16 complex floating-point elements can be done completely in the cache, each cache write operation will store 16 transposed elements (128 bytes), instead of one. Portable source code for this algorithm can be found in the Appendix.

C. EXPERIMENTS WITH PORTABLE PROGRAMS ON RISC

In this section, we examine some portable implementation codes for the FFT which employ some of the techniques described above.

1. Test Method and Conditions

To measure the execution time of a code, we used the UNIX “time” function, which provided three figures: the total elapsed time for the program to run, the amount of time that the processor was running the user program, and the amount of time that the processor was handling operating system functions required by the program. The “user” CPU time was used for measurements below.

The computer used for testing was a Silicon Graphics Indigo, manufactured in the mid 1990s, with a 100 MHz MIPS R4000 Reduced Instruction Set Computer (RISC) processor with floating-point coprocessor. Main memory capacity is 64 Mbytes, with a single 1MB unified secondary cache, and dual 8KB primary caches (one for data, one for instructions). The Silicon Graphics C compiler was used to compile the test programs, using -O3 (the maximum) compiler optimization flag.

2. Test Results

The following table illustrates the variation in performance between various implementations. In most cases, a large number of transforms was done to get a measurable run-time. When the run-time is given as a sum, the first part of the sum is the time needed for initialization, while the second part is the execution time for “one more” FFT.

Table 1. Execution Times of Portable FFT Algorithms.

N $\log_4 N$	four1	St	St-T	St-F	Fact-T	FFTW (estimate)	FFTW (measure)	St-F model	FFTW model
64 3	0.17m	0.15m	0.10m	0.08m	n/a	10m+ 0.056m	2.25s+ 0.048m	0.08m	0.045m
256 4	0.78m	0.65m	0.46m	0.36m	0.9m	10m+ 0.25m	3.79s+ 0.25m	0.44m	0.24m
1K 5	3.75m	5.1m	4.6m	2.2m	3.8m	0m+ 1.6m	5.3s+ 1.2m	2.2m	1.2m
4K 6	23m	20m	16m	11m	17m	20m+ 8.5m	6.2s+ 8.2m	10.6m	5.76m
16K 7	114m	88m	80m	20m+ 60m	70m+ 70m	90m+ 42m	7.2s+ 50m	49m	27m
64K 8	569m	540m	70m+ 380m	80m+ 280m	250m+ 320m	230m+ 230m	21s+ 200m	225m	123m
256K 9	8.9	5.9	0.3+ 5.7	0.3+ 4.8	1.04+ 1.68	0.6+ 2.4	52.4+ 2.0	1.01	0.55
$1K^2$ 10	54	55	3+ 25	2+ 24	3.9+ 7.7	2.4+ 12	192+ 10	4.5	2.46

Algorithm “four1” is taken from *Numerical Recipes in C* [Ref. 3:p.411], slightly modified to use only single-precision floating-point arithmetic. It is a “radix-two, decimation-in-time, Cooley-Tukey” algorithm, and the time includes the bit-reversed index sort process.

Algorithm “St” is a radix-four Stockham out-of-place transform [Ref. 6:p. 105], with in-line calls to the math library for multiplier coefficients. In each stage, data is transformed from the input/output buffer to a work buffer, then (when the stage is complete) the contents of the work buffer are copied back to the input/output buffer in preparation for the next stage. No data reordering is needed with this algorithm. It is slightly faster, in most cases, than the “four1” algorithm.

Algorithm “St-T” improves on “St” by tabulating trig functions. (For small transforms, the time required to initialize the tables is too small to measure meaningfully.)

Algorithm “St-F” improves on “St-T” by performing butterfly calculations on odd stages from the data input buffer to a work buffer, and on even stages from the work buffer back to the input buffer. For values of N which are an odd power of four, the result is returned in the work buffer.

Algorithm “Fact-T” factors N into two sets of smaller FFTs. For example, the $1K^2$-point transform is calculated by conceptually reshaping the data vector into an array of 4096 columns by 256 rows (both even powers of four, for the convenience of the St-F algorithm used for in-cache transforms). The array is transposed to 256 columns of 4096 rows, so elements which were separated by 4096 are now adjacent in memory. The 256 “St-F” FFTs of size 4K are followed by an element-by-element multiply by the complex constants, then another transpose. After 4096 FFTs of size 256 and another array transpose, the complete transformed data is properly ordered in memory. Since the array transpose operation is out-of-place, this algorithm requires a transpose work buffer of $1K^2$ elements. (An out-of-place transpose is faster than an in-place transpose, as well as being simpler for non-square arrays.) The tabulated constants also occupy $1K^2$ elements, and the St-F algorithm uses a work buffer of 4096 elements. Thus, the total memory demand for this algorithm is roughly three times greater than for any of the others.

Algorithm “Fact-B,” not listed in the table, reduces the memory demand of Fact-T by eliminating the matrix transpose (and the transpose buffer). When, for example, a 256-point FFT is to be done with elements scattered throughout the $1K^2$-point array, they are first gathered into a 256-element buffer. For the million-point transform, this algorithm takes 10 seconds. Fact-T is faster because it transposes multiple columns of the array at the same time, while Fact-B only “transposes” the one that’s currently needed.

Algorithm “FFTW” was obtained via the Internet from “The Fastest Fourier Transform in the West” project sponsored by the Massachusetts Institute of Technology [Ref. 12]. This program measures the performance of various FFT components to automatically synthesize an FFT algorithm which is “nearly optimal” for the current processor, whatever the FFT size and processor happen to be. Note that the time needed to analyze (either through estimation or actual measurement) and synthesize is shown as the first part of the sum in the FFTW column, and its units are always seconds. As we might expect, after we’ve paid the start-up penalty (which is substantial), FFTW demonstrates excellent performance for all but the largest transforms. For the 256K and $1K^2$ FFTs, though, it is much slower than Fact-T. On the other hand, it does not demand as much memory as Fact-T. Whether this was a conscious tradeoff or not is unknown, but memory is cheap and time is priceless.

To help illustrate the impact of cache inefficiency, we compare actual processing times with a simple model. Column “St-F model” assumes that memory access time is irrelevant, and so execution time is estimated using $C*N*\log4(N)$, where C is a constant of proportionality (C=4.297e-7) based on the “St-F” time for N=1024. The model seems to be reasonably accurate as N increases from 64 to 16K, but the run time exceeds the time predicted by the model as the cache miss rate increases. By the time N reaches $1K^2$, St-F is taking 5.3 times as long as predicted. Fact-T, however, only takes 1.7 times as long, indicating that the transform factoring process is effective at masking cache limitations.

Column “FFTW-model” is analogous to St-F model, using the measured optimization numbers. Again, we see that memory bandwidth limitations make FFTW take

roughly four times as long as predicted by the model. Put another way, 75% of the CPU performance is wasted just waiting for data.

Figure 12 illustrates the performance degradation imposed by memory bandwidth limitations. The plot labeled “St-F / St-F model” is the ratio between actual St-F time and the St-F model. The “FFTW / FFTW model” plot is analogous, while the “Fact-T /St-F model” plot is the ratio between the Fact-T time and the St-F model (since the Fact-T algorithm uses the St-F FFT internally). For small transforms, the transpose operations uselessly rearrange elements already in the cache, so Fact-T takes longer than St-F.

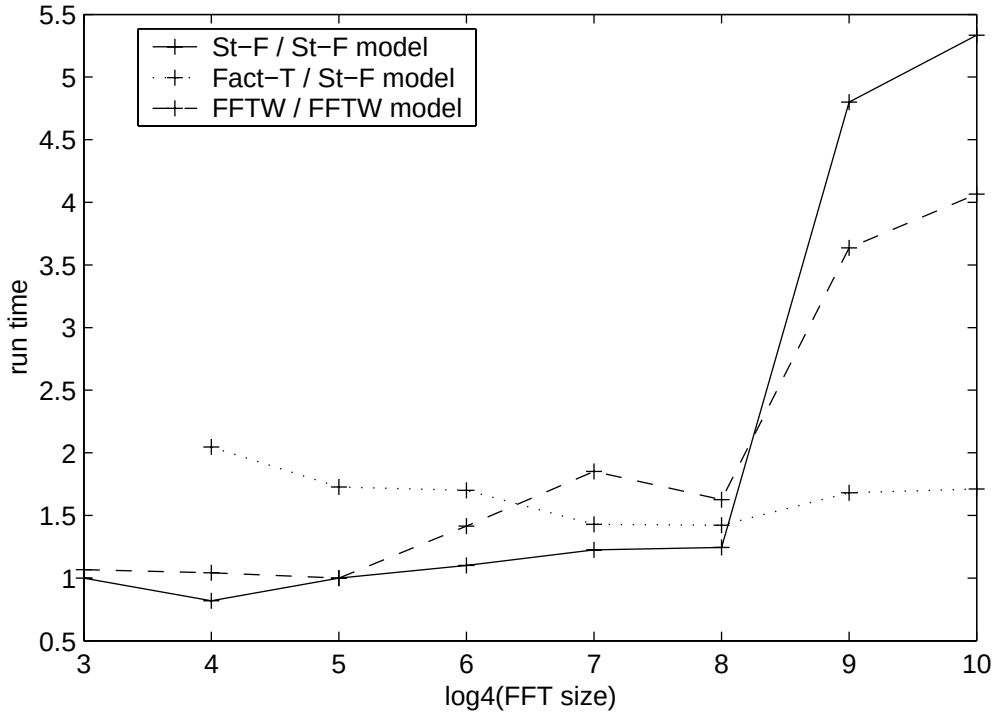

Figure 12. Cache Impact on FFT Performance.

The run time for the Fact-T algorithm can be modeled as $T_{total} = T_{init} + 3 * T_{trans-pose} + N_1 * T_{FFTN2} + N_2 * T_{FFTN1}$. For a $1K^2$ FFT, we can use $N_1=N_2=1024$, or $N_1=256$ and $N_2=4096$. We can get approximate values for T_{FFT} from the upper half of Table 1. Using values from Table 1, we expect a factoring algorithm which used FFTW, instead of

ST-F, for the small FFTs to cut another two seconds (% 26) from the time required for Fact-T to transform $1K^2$ points.

D. PORTABLE PROGRAMS ON PENTIUM-III AND RISC.

This section compares the performance of a 733 MHz Pentium-III processor and a 100 MHz MIPS R4000, using the “St-F” program described above.

1. Test Conditions

The “St-F” program was recompiled and executed on a 733 MHz Pentium-III processor with 1 GByte of main memory. The compilers (Microsoft Visual C++ V6.0 and Intel C/C++ V4.5) were configured to optimize for maximum speed of execution. We also measured the complex-float, not-in-place FFT routine found in the Intel Signal Processing Library, with the Intel C compiler. To enable comparisons with the fixed-point arithmetic implemented in the DSP device in the next chapter, the code was also modified to work with short (16-bit) integers (with internal scaling to avoid overflow). Each routine was called enough times to allow convenient measurement with a stopwatch, so each test ran for 5-30 seconds. Since each ran at least ten iterations, table initialization time was insignificant and is not listed below.

2. Test Results

The run times are shown in the table below.

Table 2. Pentium-III Run Times (all millisec).

N log4N	float Microsoft	short Microsoft	float Intel C	short Intel C	float Intel library
64 3	0.0068	0.0078	0.0059	0.0070	0.0023
256 4	0.032	0.036	0.027	0.035	0.009
1024 5	0.16	0.17	0.14	0.16	0.053
4096 6	0.75	0.85	0.66	0.79	0.32
16K 7	4.7	3.9	3.3	3.7	2.1
64K 8	41	32	38	28	30
256K 9	193	153	178	135	153
$1K^2$ 10	840	730	740	630	730

From Table 2, we see that Intel's C compiler produces code which is slightly faster than Microsoft's.

The FFT routine in Intel's Signal Processing Library is much faster, for small FFT sizes, than our portable C code. This is probably due to Intel's expert optimization of the parallel "MMX" instruction set architecture.

For small transforms, FFTs which use short integer arithmetic are slower than those which use floating-point arithmetic. Given that floating-point arithmetic is more complicated than integer, this may come as a surprise, but Intel has invested in good floating-point performance in the Pentium-III. In the floating-point version, data calculations can be executed (in floating-point hardware) while address calculations take place in integer hardware; in the integer version, the integer hardware must perform both tasks.

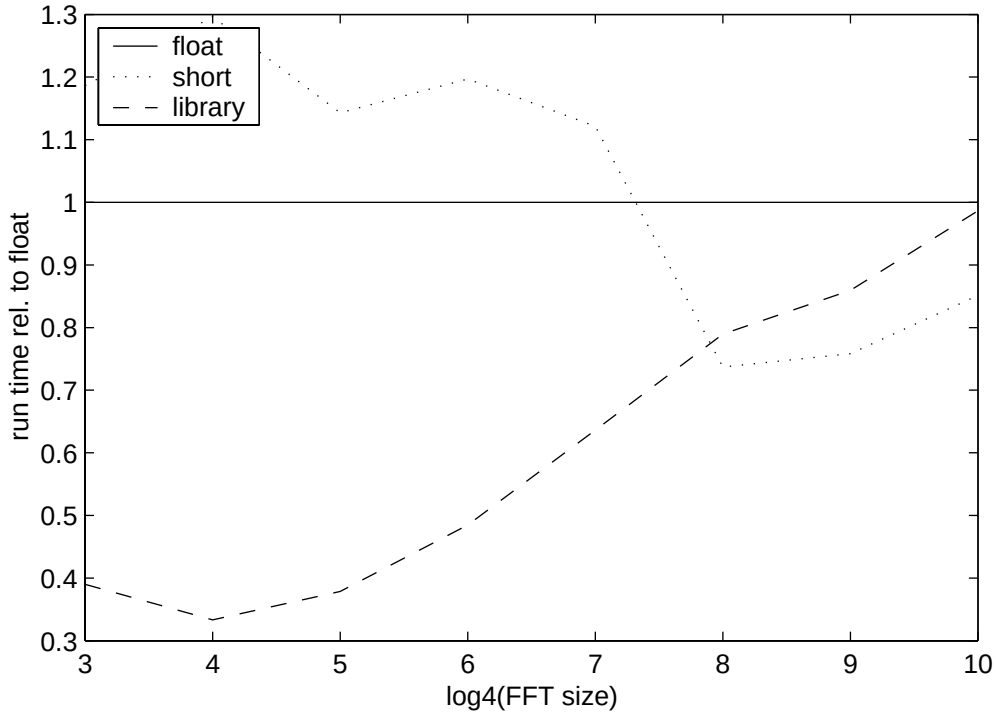

Figure 13. Library and Integer Performance, Relative to Portable Code.

For large FFTs, the integer version runs faster than the floating-point version. This is probably due to the reduced memory bandwidth required to update the cache, since each ANSI-C `float` value takes four bytes, while each `short` integer value only takes two.

While the library routine slows down the most as the FFT size becomes large, it remains slightly faster than the portable C code.

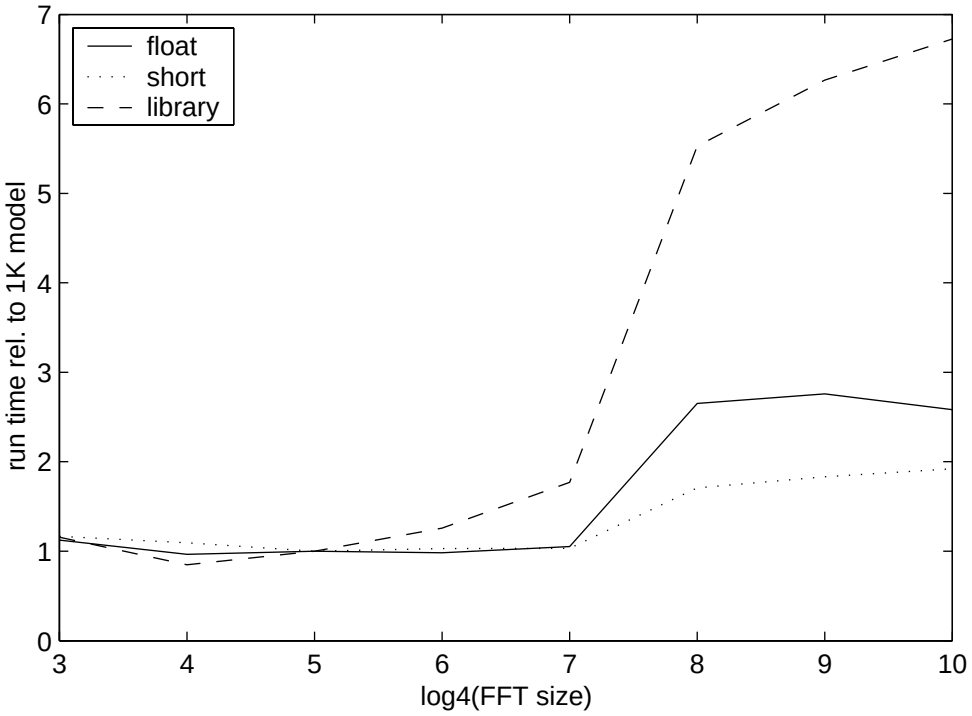

Figure 14. Cache Impact on Pentium-III Performance.

We see in Table 3 that, even after compensating for the difference in clock speeds, the Pentium-III system is at least twice as fast as the R4000 RISC system when running the floating-point portable C program. The 1K and 4K sizes illustrate the in-cache performance, while the $1K^2$ size shows that the full memory system is more efficient in the Pentium-III system than that in the R4000.

Table 3. R4000 vs. Pentium-III Performance.

FFT size	R4000	Pentium-III	ratio	clock speed comp. ratio
1K	2.2m	0.14m	15.7	2.14
4K	11m	0.66m	16.7	2.28
$1K^2$	24s	0.74s	32	4.4

This chapter illustrates several factors that affect the performance of FFT spectrum analysis on general-purpose computers: algorithm design, compiler technology, expert optimization (in the Intel Signal Processing Library), and CPU architecture. Bear in mind, though, that the R4000 predates the Pentium-III by roughly five years, so it must not be taken to represent the current state of RISC technology.

In the development of a signal processing system, the flexibility of portable programs on general-purpose computing hardware is of little importance, since the system will be dedicated to a specific process. If special-purpose hardware can accelerate this process, without imposing excessive development costs or delays, it should be included in the design. Since the FFT is a well known and important component of digital signal processing, we expect that specialized digital signal processing devices can provide such acceleration. One such device, the TMS320C6201 (by Texas Instruments), is examined in Chapter III.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DIGITAL SIGNAL PROCESSOR APPLICATION

A. OVERVIEW OF DIGITAL SIGNAL PROCESSING

Digital signal processing generally involves the algorithmic transformation of a sequence of measurements into some structure that is more useful. For example, the processor in a modem transforms a sequence of binary digits from a data terminal into a sequence of numbers representing a waveform that can pass through some communications medium, and vice versa. The processor in a digital cellular telephone transforms numbers representing the speech waveform from a microphone into bursts of numbers which satisfy the multiple-access communication protocol of the system, and reconstructs conversation from received bursts.

Though we often think of the input to a signal processing algorithm as a time-series from a single sensor, acoustic beamforming (used in SONAR) and radio direction-finding involve combining the outputs of an array of sensors.

In commercial signal processing applications, a single program may be developed to execute on thousands or even millions of processors, such as those found in cellular telephone handsets. Accordingly, economic forces tend to favor minimizing the unit hardware cost (with adequate performance) over ease of programming, since the cost of development will be shared by all buyers. Once integrated into a product, most DSP applications will execute without change for the life of the product. Both of these forces push toward a “translate slowly; run quickly” characteristic which is more tolerant of architectural innovation than the general-purpose computing market. The developer of a new DSP device need not be concerned with maintaining compatibility with past generations of application or operating system programs.

B. DESIGN FEATURES OF THE TMS320C6x FAMILY

Recent advances in computer architecture (apart from multiprocessor parallelism) have moved toward performing micro-operations in parallel. A simple pipeline can fetch instruction(*k*) while decoding instruction(*k*-1), loading operands for instruction(*k*-2), performing arithmetic for instruction(*k*-3), and storing the result from instruction(*k*-4) to memory. However, this assumes that there are no dependencies between these instructions. If an operand for instruction(*k*-3) is stored at the same memory location as the result of instruction(*k*-4), and instruction(*k*-3) loads the value before instruction(*k*-4) has stored it, then the result of instruction(*k*-3) may be incorrect. “Superscalar” processors, such as the Intel Pentium family, contain logic which detects dependencies as the program is executed, re-ordering instructions or stalling the pipeline (for example) until dependencies are satisfied. Thus, the widely used Intel x86 instruction-set legacy can be executed with increasing speed, at the expense of complex hardware and variable timing.

Programmers of Very Long Instruction Word (VLIW) and RISC architectures address the dependency problem during development (design and compilation), rather than execution, of the program. This allows the logic which is dedicated to dependency analysis to be eliminated. Additional registers, cache memory, and/or arithmetic units can be put in its place, or the overall size of the device can be reduced (lowering its cost). Instead of automatically detecting the opportunities for parallel processing implicit in a sequence of simple, short instructions (typically 32 bits long), VLIW machines employ an instruction word which has enough bits to explicitly schedule parallel operations. In the C6x series, the instruction word (“execute packet”) can be as long as 256 bits, specifying up to eight simultaneous independent arithmetic and/or memory access operations. Additional parallelism can be achieved by the use of reduced-precision arithmetic instructions, which allow, for example, two 16-bit integer additions to be performed with a 32-bit adder by blocking the carry propagation from the low half to the high half. Multiply instructions which take as operands either the high or low halves of 32-bit registers also facilitate the packing of 16-bit values into 32-bit registers. On the other hand, effective utilization of

parallel hardware assets depends on the inherent parallelism of the algorithm, its translation into instructions, and the memory access required.

Major components of the C6x processor are sketched in Fig. 15. Note that a set of four functional units is associated with each register file, with two “cross-paths” to allow one functional unit from each set access to the opposite register file. Devices in the C6x family also include various configurations of on-chip memory, serial ports, timers, direct-memory access controllers, etc., but they are of no concern here.

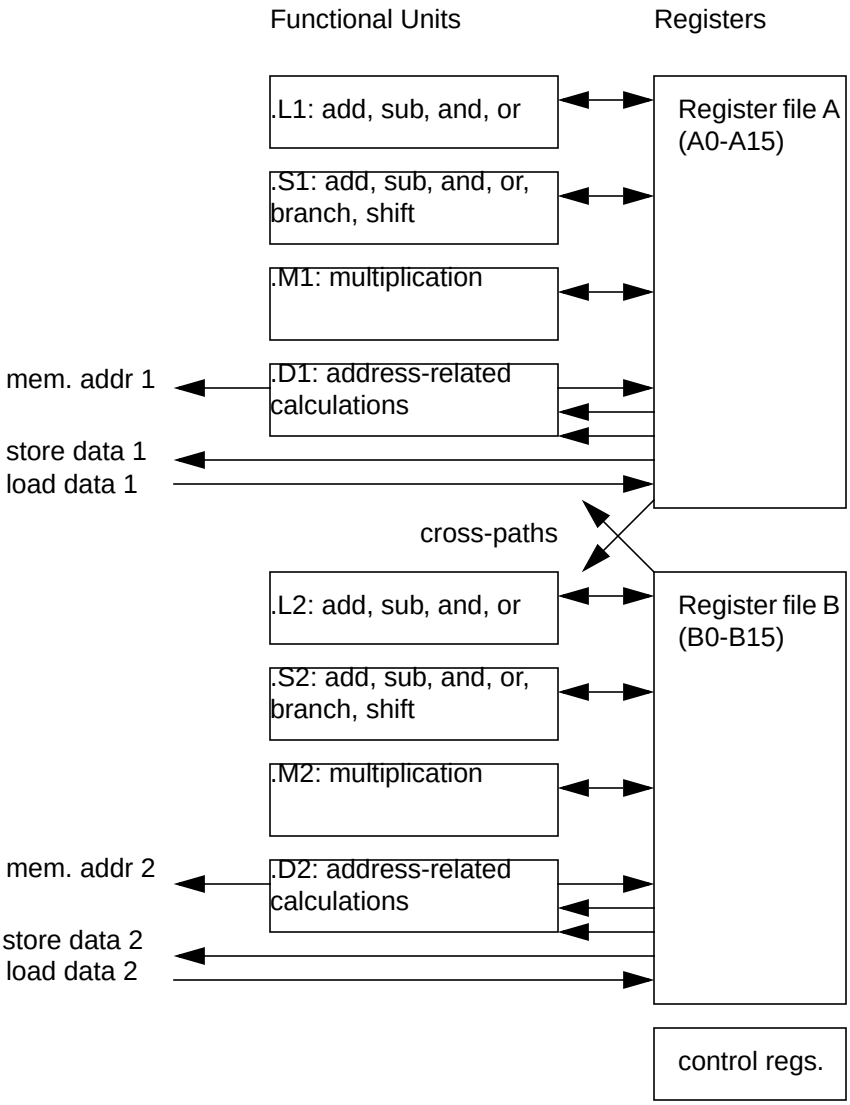

Figure 15. Simplified C6x Processor Block Diagram.

Figure 16 shows the assembly-language source code for a sample execute packet, taken from an FFT benchmark program published by Texas Instruments in C6x assembly language.

```
                                ; comments...
      sub   .L1  a1, a2, a4     ; a4 = a1 - a2, using functional
                                ; unit ".L1"
||    shr   .S1  a4, 1, a7      ; a7 = a4 >> 1 (shift right, / 2)
||    shr   .S2x a4, 1, b9      ; b9 = a4 >> 1 (but on the B-side)
||    mv    .L2  b6, b0         ; b0 = b6
||    stw   .D1  a12, *+a0[5]   ; store word A12 to memory at the
                                ; address five words above the
                                ; location in A0.
||    stw   .D2  b12, *+b1[6]
```

Figure 16. Parallelism Explicitly Coded Into an Execute Packet.

The "||" symbol indicates that the instruction which follows should be performed in the same clock cycle as the instruction on the preceding line, so all six of the instructions above execute in the same clock cycle.

The VLIW architecture simplifies the instruction processing pipeline by making parallelism explicit during execution of linear sequences of instructions. Very few programs, however, run for very long without requiring a deviation from sequential instruction fetching which invalidates the partially processed instructions in the pipeline. Conventional assembly-language programs perform a comparison, and "conditionally branch," depending on the result of the comparison. To minimize the number of branches

in a program, the C6x processors make *all* instructions conditional, not just branches. For example, the integer “absolute-value” function can be coded as shown in Fig. 17.

ANSI C

```
    if (x < 0) x = -x;
    (next statement)
```

Gnu C/assembler for Intel Pentium

```
    cmpl $0,-4(%ebp) # compare x (in memory) < 0
    jge .L1          # if x >= 0, jump (branch) to label L1
    negl -4(%ebp)    # negate x (in memory)
.L1: (next statement)
```

TI C6x assembler

```
        ldw   .d2  *+SP[0x3], B4; initiate load of x into reg.
        nop   4                 ; wait for load to complete
        cmpgt .l2 0x0, B4, B0  ; if 0 > B4, B0 = True
||      sub   .s2  0x0, B4, B4  ; (parallel) B4 = 0 - B4
  [B0]  stw   .d2  B4, *+SP[0x3]; if B0 true, store updated x
    (next statement)
```

Figure 17. C6x Branch Avoidance with Conditional Store Instruction.

The Intel code implicitly fetches the value of x from memory to set a condition flag based on the comparison, but doesn’t retain the value in a register. Then, it (conditionally) jumps to the label L1, which may disrupt the instruction processing pipeline. If the jump is not taken, x is again implicitly loaded from memory, negated, and rewritten. (Whether or not the processor actually performs this exact sequence of operations depends on how clever the processor is at interpreting the instruction stream.)

The C6x code, on the other hand, does not interrupt the sequential pipeline flow of instructions; register B0 is used as a condition flag and simply prevents storage of the negated value of x (in B4). The negation is always calculated, in parallel with the comparison, but the result may be ignored. (The “nop 4” instruction explicitly stalls the processor while it waits for the memory to respond. In other algorithms, it may be feasible to execute the “ldw” instruction earlier in the instruction stream, so the delay cycles can be occupied by useful instructions.)

The impact of branches is also reduced in the C6x by allowing the programmer to insert instructions into the “delay slots” which follow a branch instruction (not shown above).

Efficient use of fast arithmetic logic demands a ready supply of operands. At the top of the memory hierarchy are the processor’s thirty-two 32-bit registers, which provide all arithmetic operands to the arithmetic units. Register addresses are encoded in each instruction and access is immediate. A register can serve as both source and destination in the same instruction cycle, and a register can be read by as many as four simultaneous instructions.

The next level of memory hierarchy is the on-chip Internal Data Memory (IDM), storing 64K bytes (16K 32-bit words). To read from the IDM into a register requires five clock cycles, from the time the address is sent (from a register) until the data is written to a register. Such data reads are pipelined, though, so two data accesses can be started on every clock cycle. Writing to IDM has similar timing. Note that memory read/write latency can be an important factor to consider when scheduling an algorithm. If the read operation can be initiated five cycles before the value is needed, the latency is invisible. Otherwise, the program must be coded to execute “no-op” instructions before continuing.

In addition to data accesses, the processor must also be supplied with program instructions. At peak performance, it will be executing 32 bytes (*not 32 bits*) of instruction code with every 5 nanosecond clock cycle, or 6400 Megabytes per second. Practical algorithms may demand this bandwidth a substantial fraction of the time. The Internal Program Memory (IPM), containing 16384 instructions, provides this bandwidth through a 256-bit wide path to the processor.

Access to external (off-chip) memory is also supported. From the program’s point-of-view, internal and external memory both have a five clock latency. However, external memory actually takes longer to respond, so execution of all instructions is suspended as long as necessary for any data. The impact of this is shown in section E.

C. SOFTWARE TOOLS AND TECHNIQUES

Development of an application for the C6x requires translation of the algorithm into a sequence of arithmetic operations, allocation of the arithmetic to the parallel functional units of the processor, allocation of variables to registers, and allocation of data structures to the memory hierarchy. A collection of tools is needed to perform these tasks.

1. TI's Tool Set: Code Composer Suite

Implementation of the algorithm in "ANSI C" is typically the first stage of development. Though the programmer has the least amount of control over exploitation of parallelism, a C program can be portable across many different machines and operating systems. Development in C can prove, in a workstation environment, that the algorithm is theoretically sound and worthy of integration into the DSP environment, with less effort than starting development on the DSP. In TI's "Code Composer Suite" development environment includes several options for optimization of the executable program.

Compiler "pragma" directives also allow the programmer more control over the compiling process than the standard C language provides. Data structures can be assigned to specific memory sections, and the programmer can exercise fine-grained control over some of the optimization efforts of the compiler.

The C compiler for the C6x sold by TI can be used to compile ANSI C, but also supports processor-specific hardware features using "intrinsic functions." An example is shown in Fig. 18.

```
int sum_hl = 0, sum, index;
short b[100];

for (index = 0; index < 100; index += 2) {
  sum_hl = _add2( sum_hl, *(int *)b[index]);
}
sum = (sum_hl & 0xffff) + (sum_hl>>16);
```

Figure 18. Calculation Using Parallel Partial-Word Arithmetic.

This code fragment computes the sum of the `short` (16-bit) values in the array "`b`" by reading pairs from memory as `int` (32-bit integer) values, accumulating the even elements in the high 16 bits of "`sum_hl`," accumulating the odd elements in the low 16 bits of "`sum_hl`," and finally combining the high and low partial sums into a total. Only 50 loop iterations are needed to sum the 100 elements. Similarly, the "`_mpyhl()`" and "`_mpylh()`" intrinsics allow the high half of one register to be multiplied by the low half of another, without disturbing the other half of either register. (This is especially handy for the complex arithmetic used in signal processing.)

Intrinsic functions allow the C programmer access to specialized instructions of the C6x, but not to the allocation of variables to processor registers. If this additional level of control is needed to improve algorithm performance, the programmer can use "linear assembly language." Each linear assembly statement specifies the operation of and arguments to one functional unit, and the assembly optimizer tool combines statements into parallel "execution packets." Programming in linear assembly code requires the programmer to manage the assignment of variables to registers (bearing in mind that only five specific registers can be used as condition flags, and only eight can be used for certain addressing modes), the assignment of functions to functional units (e.g., though multiplication can be done only in the ".M" units, AND can be done in both .L and .S, and ADD can be done in .L, .S, and .D), and access to variables from functional units (A-side functional units can only write results to A-side registers). Linear assembly language allows the programmer to "misuse" processor control registers as scratchpad storage. Control registers cannot supply operands to arithmetic units, but can be more quickly accessed than even on-chip memory. Of course, the programmer must ensure that this does not prevent normal operation of the processor; any function which uses the Interrupt Return Pointer control register for data must never attempt to return from an interrupt service routine without somehow restoring that value!

For maximum control, the programmer can write *scheduled assembly language*, manually combining statements into parallel execution packets. Developing scheduled assembly language can be a challenging task. In addition to the complexities of linear

assembler, the developer must also be aware of constraints on operation parallelism, such as the fact that, of the eight operands which could be needed by the four A-side functional units, only one operand can be read from any of the sixteen B-side registers, and vice versa. The developer must also be aware of operation latencies. When loading a word from memory, the destination register can be used for other purposes for four more instruction cycles before the loaded word actually arrives. The internal latency of the multiply unit means that the result can't be read from the destination register until two cycles after the multiply is issued. (This means, though, that a multiply unit can be used to simply store a value that would otherwise overflow the register set. Multiplication of the value by one in cycle *k* allows use of cycle *k*+1 to save the contents of the register which will be overwritten by the multiplier result in cycle *k*+2. This saved substantial time in our FFT function.)

Regardless of the language used to express the algorithm, the resulting executable program and data structures must be allocated to specific memory addresses. Embedded processors are typically surrounded by customized memory configurations and rely on physical hardware addresses (both for memory of various types, and memory-mapped input/output device registers). The program linker performs this function.

Allocating data to a relatively slow memory can have a dramatic effect on performance. For example, our FFT code (with intermediate stage scaling) computed a 4K-point transform in 402 μsec when the data vector and constant table were stored in Internal Data RAM. The same code, with constants in Synchronous Burst Static RAM, took 671 μsec. After moving the data vector to SBSRAM, the algorithm took 3571 μsec (see details below).

2. Auxiliary Tools

a. "Financial" Spreadsheet

Optimization of an algorithm in scheduled assembly language (as defined above) can begin with scheduled assembly language code generated by a compiler, based on a portable description of the algorithm. Modifying the assembly language code can then incorporate the programmer's understanding of the precise problem to be solved. (For example, the programmer may know that a certain variable can only take on one of three

different values, which cannot be expressed to the compiler but may simplify the problem.) Or the programmer may be presented with a completely scheduled assembly-language program which needs only a slight change to perform the current application. In either case, the trick is to make an incremental change (add new behavior) without disrupting the existing program.

One way for the programmer to track the total processor state during concurrent operations is with a spreadsheet program (e.g., Gnumeric or Excel). Though none of the calculation features of the program are used, the flexible tabular format is a valuable aid. For the effort described in this Thesis, a table was created with one row for each clock cycle of the program and one column for each of the eight functional units (and the two cross paths). If a functional unit executes an instruction during the clock cycle, its box is marked as shown below. This table is most useful to identify free functional units to which inserted instructions can be allocated. For example, if we need to multiply a value by four, we could use a multiply unit (if we have a register containing the value “4” and we can wait an extra clock cycle), or we could add it to itself (in L, S, or D units) twice, or shift it left two posititions (in an S unit). The table shows which (if any) are free to use. Also, when we need to insert a new execution packet, we can quickly scan down the M unit columns to see where multiplier latency will complicate the problem.

Scheduled assembly-language source code for four clock cycles.

```
        ; execution packet 22.5
        mv     .s2x  a10, b10
||      mv     .s1x  b3,  a9
|||[!b2] addaw .d1   a5,  1,   a5

        ; 23
        shr    .s2   b3,  16,  b3
||      shr    .s1   a10, 16,  a10
||      mv     .l1   a6,  a1
||      addaw  .d1   a5,  a7,  a5

        ; 24
        add    .l2   b3,  b10, b11
||      sub    .l1   a9,  a10, a12
||      sub2   .s2x  b1,  a8,  b1
||      add2   .s1x  b1,  a8,  a8
||      addaw  .d1   a5,  a7,  a5

        ; 25
        ext    .s1   a8,  16,  18,  a8
||      shr    .s2x  a8,  18,  b10
||      sub    .l2   b3,  b10, b12
||      add    .l1   a9,  a10, a9
||      mpylh  .m1x  a12, b15, a10
||      mpy    .m2   b11, b15, b10
||      ldw    .d2   *b5++[b6],b10
```

Arithmetic unit use summarized for 23 clock cycles (including the four at left).

	S1	D1	L1	M1	S2	D2	L2	M2	X	X
22.5	x	x			x				x	x
23	x	x	x		x					
24	x	x	x		x	x	x		x	x
25	x	x	x	x	x		x	x	x	x
25.5	x			x	x	x			x	x
26	x	x		x	x	x		x	x	x
27			x	x	x	x	x	x	x	x
27.5	x	x			x	x			x	
28	x	x		x	x	x		x	x	x
29		x	x	x	x	x	x	x	x	x
30		x	x	x	x	x	x	x	x	x
31	x	x	x		x		x			
32	x	x	x		x	x	x	x	x	
33					x			x		
34	x				x		x			
35					x		x			
36							x			
37	x				x					
38					x					
39										

Figure 19. Spreadsheet Summary of Arithmetic Unit Usage.

The spreadsheet also tabulates usage of the thirty-two general-purpose registers, which is essential for minimizing the number of data transfers to and from memory. The following table illustrates the same instructions (and more) as in Fig. 19.

	b5(c)	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	LD/ST2
22.5	++	.	.	.	.	-	-	-	.	.	.	
23	.	.	.	.	.	t1	-	-	.	.	.	
24	+=4*b6	*	.	.	.	*	-	-	.	.	.	x/yi0
25	.	.	.	.	.	*	r1c *	-	.	.	*	
25.5	+=4*b6	*	.	.	.	yo0*	.	r2c	.	.	.	x/yi1
26	+=4*b6	*	.	.	.	rc1	.	.	*	.	*	x/yi2
27	+=4*b6	*	.	.	.	*	*	.	yo1	*	*	x/yi3
27.5	.	.	.	.	.	yo0 *	sc1	.	*	yo2 *	.	yo0
28	.	.	.	.	*	x/yi0	.	.	rs1	yo1 *	.	yo1
29	.	.	.	*	*	.	*	.	*	yo2 *	?s/c1	yo2
30	.	.	.	.	.	*	x/yi2 *	**	ya1	sc2	.	yo3
31	.	.	*	.	.	.	.	rs2 *	.	*	.	
32	*	.	.	.	.	*	*	rs3 *	.	ya2	.	

Figure 20. Spreadsheet Summary of Register Usage.

Each instruction cycle occupies one row of the sheet. The text in each cell was added according to the following rules:

1. All cells are initially blank; when the analysis is complete, no cells should be blank.

2. If a register receives a value (loaded from memory or the result of an arithmetic operation) during the prior cycle (such that it can be used as an operand in the current cycle), put the name of the variable into the associated cell. If the 32-bit register contains a pair of 16-bit values, separate their names with a slash (“/”). In cycle 28, a pair of 16-bit values labeled “x/yi0” appears in register B10, the result of the “LDW” instruction that is shown in Fig. 19 (cycle 25).

3. If the operation is iterative (e.g., i++), put in the expression. In the top row of Fig. 20, register B5 is incremented in cycle 22.5. In cycles 25.5 through 27.5, register B5 is incremented by four times the value in register B6.

4. If the value in a register is an instruction operand (including memory write) during the current cycle, put an asterisk in the cell. Register B6 is used to auto-increment B5 in cycles 25 through 27. If this value will not be used again, underline the cell.

5. If a register must preserve its value through the cycle, put a period in the cell. Register B11 receives a new value in cycle 27.5, preserves it through 28, and it is read in cycle 29.

6. Between the time a cell is underlined (as in rule 4), and it receives a new value (rule 3), put a dash into the cell. This cell is free to be used for a new instruction. B10, B11, and B12 are free in cycle 25.5.

7. If the status of a register is dependent on a condition flag, prefix the cell text with a question mark. Register B15 is conditionally loaded with a pair of sixteen-bit values for cycle 29.

It is also useful to have columns in the table to indicate load/store operations, as shown at the right side of Fig. 20.

Once the spreadsheet is completed for a baseline algorithm (e.g., as produced by the compiler), the impact of changes during manual optimization can be assessed. For this project, we needed to make two changes to the FFT routine provided by TI: first, TI developed for a “little-endian” memory access configuration, while our processor board is configured for “big-endian” memory access, and secondly, we wanted to prevent arithmetic overflow by implementing distributed scaling as described in section II.A.5.

Applying the rules above to the ten-cycle inner loop of the algorithm showed that there were 22 register cells (out of 320) free to receive new values, and 17 (out of 80) free functional unit cells. Each instruction inserted into the loop creates eight more free functional unit cells, and from none to six more free register cells (depending on where in the loop it is inserted). Inserting an instruction cycle creates a free register cell only when it is inserted below a row with one or more cells underlined according to rule 4, above, or already free according to rule 6.

In the end, accomplishing the required two changes resulted in an inner loop that takes 15 more nanoseconds (clock cycles designated 22.5, 25.5, and 27.5) than the original code.

b. Programs which Write Programs

Once we have decided to use tabulated values in one or more parts of a program, we are faced with the potentially tedious and error-prone task of creating the table. Our solution is to write a portable C program which generates, as its output, another C source-code file which can be compiled by the DSP development tools.

D. DESIGN FEATURES OF THE PENTEK 4290A

A processor chip by itself is useless. Pentek, Inc. integrates four C6x processors with several types of memory and input/output devices into the model 4290A VME-bus single-board computer [Ref. 13]. Each C6x on the board has exclusive access to 128k 32-bit words of Synchronous Burst Static RAM (SBSRAM). During block transfers, a new data word can be transferred on every CPU clock cycle (800 MB/sec), which is half the rate of the on-chip IDRAM. [Ref. 13: p.13]. Access latency, however, can cause a significant performance penalty, as shown in the performance measurements table below.

Each C6x also has exclusive access to 4M 32-bit words of Synchronous Dynamic RAM (SDRAM). At best, SDRAM has half the transfer rate (400 MB/sec) of SBSRAM, but data transfers stall for 60 nsec (12 clock cycles) whenever a page boundary is crossed [Ref. 13: p.14]. Again, access latency effects can be dramatic.

The 4290A also has FIFO, dual-port, and global memories for interprocessor communication, but they are irrelevant to this discussion.

E. EXPERIMENTAL RESULTS FROM DSP PROGRAMS

1. Test Method and Conditions

In this test, we observe the impact of memory latency, for each of the three types of memory (IDRAM, SBSRAM, and SDRAM) on the Pentek 4290A, for two algorithms: application of the window function and the 4K FFT. Three data structures are involved: *fifo_buf* contains complex short-integer sampled data to be processed; *fft_buf* is the working memory of the FFT; *w* contains the tabulated FFT coefficients.

As the first stage in spectrum estimation, we multiply each component of each complex element of the input data vector (in *fifo_buf*) by the corresponding element of the

tabulated window function (in the odd-indexed elements of *w*). To save space, we derive the VonHann window function “on the fly” from the tabulated FFT constants, as shown in Fig. 21.

```
fifo_ptr = &fifo_buf[0]; /* Point to first input value (from A/D).*/
win_ptr = &w[1];         /* Point to first cos() value. */
fft_ptr = &fft_buf[0];   /* Point to first output value (to FFT). */
for (ii=0; ii < 4096; ii++) {
  tmp = 16384 - (*win_ptr >> 1); /* Scale to avoid overflow. */
  win_ptr += 2;      /* Skip over sin() value to next cos(). */
  *fft_ptr++ = (*fifo_ptr++ * tmp) >> 16; /* Scale to restore ...*/
  *fft_ptr++ = (*fifo_ptr++ * tmp) >> 16; /* ..fixed-point format.*/
}
```

Figure 21. ANSI C Source Code for Window Function Algorithm.

Then the 4K FFT algorithm (not including the data unscrambling phase) is applied to data in *fft_buf*. (Since Welch’s method for spectrum analysis involves the sum of periodograms, we unscramble the final result, rather than each raw FFT output, so the unscrambling algorithm is not included in these measurements.)

2. Test Results

The execution time of the window and FFT functions were measured using TI’s Code Composer Suite performance profiling timer, with results as shown in Table 4.

.

Table 4. Pentek DSP Memory Performance.

function	clock cycles	time (μsec)	ratio to best time	conditions
win fft	10,514 83,737	53 420	1.00 1.04	all (fifo_buf, fft_buf, and w) in IDRAM
win fft	117,079 80,659	585 403	11.13 1.00	fifo_buf in SBSRAM
win fft	165,268 80,658	826 403	15.72 1.00	fifo_buf in SDRAM
win fft	63,768 134,218	319 671	6.07 1.66	w in SBSRAM
win fft	87,335 245,585	437 1228	8.31 3.04	w in SDRAM
win fft	61,795 714,198	309 3571	5.88 8.86	fft_buf in SBSRAM
win fft	78,633 1,434,998	393 7175	7.48 17.80	fft_buf in SDRAM
win fft	253,007 1,619,164	1265 8096	24.06 20.09	fft_buf and w in SDRAM
win fft	657,522 1,622,313	3288 8112	62.54 20.13	all in SDRAM

This table clearly shows the importance of putting frequently used data into the Internal Data RAM of the processor. Though the SBSRAM is described as “zero wait-state” memory, since a new value can be read on every clock cycle, attempting to use it for the FFT’s data vector wastes over 88% of the processor’s performance due to access-time latency. With all three data structures in SDRAM, the processor spends 95% of its time waiting.

The fastest configuration applied the window to 4096 complex data elements in only 10,514 cycles, or about 2.5 clock cycles per complex element. (Remember, reading from IDRAM takes five clock cycles, multiplication takes two, and branching back to the top of a loop takes six.) The C compiler exploits the parallelism of the window algorithm

(every data element is independent), but how does it perform against the complexity of a portable FFT routine?

The “Stockham algorithm, radix-four, with tabulated constants, alternating work buffers, and integer arithmetic” (algorithm St-F of Chapter II, converted to integer arithmetic with internal scaling) was evaluated with Code Composer Suite’s simulator over a set of compiler-optimization switches. The portability of the function was not impaired by using intrinsic functions or pragmas. Execution times for 4K transforms are listed in Table 5.

Table 5. TI DSP Compiler Performance.

optimization	cycles	time	optimization features
none	2380975	11.9 msec	most easily debugged, due to the clear association of source and executable code, memory is up-to-date after each C source line has completed.
-o0: register	1390231	6.95 msec	simplifies control flow, assigns variables to registers, simplifies expressions and statements, etc.
-o1: local	960201	4.80 msec	adds local copy propagation, eliminates local common expressions, etc.
-o2: function	270106	1.35 msec	adds conversion of array references to incrementing pointers, loop unrolling, loop optimizations, software pipelining., etc.
-o3: file	270106	1.35 msec	makes short functions in-line, propagates argument into function when function argument always has the same value, etc.

Comparing the best compiler-optimized portable routine (1350 μsec) against the best hand-optimized routine (403 μsec), we see that the compiled routine takes more than three times as long. Inspection of the scheduled assembly language produced by the compiler shows that at most six of the eight functional units are coded into a single execution cycle (and then, in only one of the 182 cycles in the function), and two cycles use five units. For the rest of the program, at least half of the functional units are idle. In the inner loop, 94 instructions will execute in 49 cycles, so we average less than two instructions per cycle. There is, therefore, a strong incentive to manually optimize time-critical portions of an application or incorporate a well-optimized library routine.

F. COMPARING PENTIUM-III AND C6X

In Chapter II-D we saw that the Pentium-III could compute a floating-point 4K FFT (with properly sorted output) in 320 μsec, while in section III-E, the C6201 took 403 μsec to do an integer calculation (without sorting the output). The Pentium-III was faster and easier to program. So why should we consider DSP devices?

CPU cost: the TI DSP chip currently sells for about $40 (in 1000-unit quantities) [Ref. 13], while 1 GHz Pentium-compatible AMD Athlon 4 processors cost $425 (in 1000-unit quantities) [Ref. 14].

System cost: The cost of the DSP device includes on-chip serial ports, DMA controllers, and memory controllers, which would be external to the Pentium. On the other hand, the volume of Pentium-system sales provides economies of scale which are lacking in the DSP-on-VMEbus market.

Physical size: Our DSP system puts four processors on each VME card, while our Pentium system packages just one CPUs per card.

State of the art: the clock speed of C6x family devices has increased by a factor of three since the C6201 was introduced, while Pentium speeds approach 1500 MHz, so a current DSP chip may outperform a current Pentium. However, Pentium-class processors reach the commercial marketplace much more quickly than DSP devices do (Pentek's Model 4290 C6x board is still plagued with functional and/or documentation faults), so low-volume DSP developers may always lag behind.

G. LARGE TRANSFORMS ON DSP

The Internal Data RAM of the C6201 will not be large enough to store the coefficient tables and data vectors for transforms of greater than 4K elements. To avoid the severe performance penalty of fetching data elements one at a time from external memory (as illustrated in Table 4), the C6201's internal Direct Memory Access (DMA) controllers can be used to swap work-vectors in and out of IDRAM, much as the cache controllers described in Chapter II do for general-purpose processors. The DMA controllers, however, can be programmed with arbitrary "stride" values for reading and writing, which allows them to perform the transpose operations without burdening the processor. It may be pos-

sible for DMA transfers to take place concurrently with FFT calculations, without slowing the FFTs, but the actual effectiveness of this technique remains to be determined.

IV. CONCLUSION

This thesis has explored a variety of issues involved in developing an FFT-based spectrum analyzer. We've shown how a popular "textbook" algorithm for the FFT can be modified to run faster on a general-purpose computer, and how main-memory access can become a bottleneck which limits the performance of the processor on large (greater than 64K) FFTs. An algorithm (program "Fact-T" of the appendix) was presented which recovers much of the lost performance by improving the effectiveness of the cache management. For a 4K transform, the modified routine runs in half the time of the original; for a $1K^2$ transform, one seventh.

Several options for spectrum analysis window function design and storage were assessed with results that must be considered in an application context. The Von Hann window was found to provide acceptable spectrum estimates for our application, while allowing reuse of trigonometric factors previously tabulated for the FFT itself. Decimation and interpolation of window function data was found to produce spurious features in the spectrum which diminish for large tables and higher-order interpolation algorithms.

We implemented the FFT on a fixed-point digital signal processor, the TMS320C6201, taking advantage of assembly language software development tools which allow parallel functional units to be optimally exploited. An algorithm for sorting FFT results into natural order was developed which is more memory efficient than an algorithm described in TI application notes.

Implementation of manually scheduled assembly language programs was shown to be facilitated by describing the resources and dependencies of the program in a general-purpose spreadsheet tool.

Run-time measurements on an MIPS R4000 RISC processor, Intel Pentium-3, and TI TMS320C6201 illustrate the advances in architecture which allow the Pentium-3 and 'C6201 to provide performance which goes beyond that which would be predicted based solely on their shortened clock cycles. For a portable C 4K FFT, the Pentium requires less than half as many clock cycles as the R4000; for a $1K^2$ FFT, superior cache management

gives it a four to one advantage per clock cycle. The 200 MHz C6201 running TI-optimized code is essentially equal in performance to the 733 MHz Pentium-3 running Intel-optimized code (at the 4K size), having a 3.5 to one advantage per clock cycle but a 3.7 to one handicap in clock speed. Intel has recently announced that the Pentium clock speed may double, while TI is promising 600 MHz versions of its product line, so the performance race may remain close for the foreseeable future. Considerations other than arithmetic performance, such as size, cost, power consumption, arithmetic precision, and designer familiarity outweigh their relative performance on generic benchmark programs. Determining whether or not a DSP device can satisfy a challenging performance requirement is likely to require substantial investment to implement a realistic test, and coordination with the manufacturer to determine component availability.

A 4K FFT can run without reference to off-chip memory, which is demonstrated to be essential for efficient operation. For larger transforms (e.g., $1K^2$, $4K^2$), the C6201 must rely on DMA transfer of intermediate results between internal and external memories to maintain performance anywhere close to small transform performance. The large-FFT portable program “Fact-T” illustrates the sequence of block data movements and arithmetic.

LIST OF REFERENCES

1. Oppenheim, Alan and Shafer, Ronald, *Digital Signal Processing*, Prentice-Hall, 1975.
2. Heuring, Vincent, and Jordan, Harry, *Computer Systems Design and Architecture*, Addison-Wesley, 1997.
3. Press, Flannery, Teukolsky, and Vetterling, *Numerical Recipes in C*, Cambridge University Press, 1988.
4. Arfken, George, *Mathematical Methods for Physicists*, Academic Press, 1985.
5. Aho, A.V, Hopcraft, J. E., and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
6. van Loan, Charles, *Computational Frameworks for the Fast Fourier Transform*, SIAM, 1992.
7. *TMS320C6x DSP Design Workshop: Student Guide*, Texas Instruments, 1999.
8. Gardner, William A., *Statistical Spectrum Analysis*, Prentice-Hall, 1988.
9. Gutman, Ron. "Algorithm Alley," *Dr. Dobb's Journal*, #316, September, 2000.
10. *Bit-Reverse and Digit-Reverse: Linear Time Small Lookup Table Implementation for the TMS320C6000* (SPRA440), Texas Instruments, Inc, May, 1998.
11. *Operating Manual, Pentek Model 6216 Dual A/D Converter and Digital Receiver VIM Module for Pentek Models 4290 and 4291*, Pentek, Inc., 1999.
12. *Operating Manual, Pentek 4290/4291 TMS32C6201/TMS320C6701 Digital Signal Processor for VME Bus*, Pentek, Inc., 1999.
13. Frigo, Matteo and Johnson, Steven, *The Fastest FFT in the West*, MIT-LCS-TR-728, www.fftw.org, 1997.
14. *Technology Innovations*, Texas Instruments, Inc., Vol. 7, February 2001. p. 8.
15. Ascierto, Jerry, "AMD matches Intel, rolls 1-GHz mobile processor," in *Electronic Engineering Times*, May 21, 2001. p. 48.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: PORTABLE SOURCE CODE

A. Str4flip: An out-of-place radix-four small FFT algorithm

```
/*
  Stockham Radix-four FFT (unit-stride, not in-place)
  based on Algorithm 2.4.2, Stockham Radix-four Algorithm, in
  Computational Frameworks for the Fast Fourier Transform, SIAM,
  by Charles Van Loan.

  This version uses tabulated twiddle factors, for speed optimization.
  It also flip-flops samples between buff and data, to avoid copying.
  If the base-4 log of the FFT size is odd,
    then the result is returned in the workspace “buff,”
    else the result is returned in the input array “data.”

  */
#include <stdio.h>
#include <math.h>
/* M_PI is defined in math.h */
#define MAIN 1
/*
  The Stockham version has unit (complex) stride for each pass; data is
  in order before and after transform, but the computation is not
  “in-place.”
  “buff” is a workspace equal in size to “data.”
*/
/*
  Initialize the trig-factor table. As usual, nn is the number of complex
  samples in the FFT, so the table has nn/4 complex (= nn/2 float) val-
ues. */
void init_table( float w_tab[], const int nn )
{
  int jj;
  double temp;
  for (jj=0; jj < nn / 4; jj++) {
    temp = 2.0 * M_PI * (double) jj / (double) nn;
    w_tab[2*jj    ] = (float)  cos( temp );
    w_tab[2*jj + 1] = (float) -sin( temp );
  }
}


void str4flip( float data[], float buff[], float w_tab[],
     const int nn, const int log4n)
{
  int j, k, q, r, r_star, ell, ell_star;
  float wr, wi, w2r, w2i, w3r, w3i;
  float ar, ai, br, bi, cr, ci, dr, di;
  float t0r, t0i, t1r, t1i, t2r, t2i, t3r, t3i;
  float *d_ptr, *b_ptr, *swap_ptr;
```

```
  int index;
  int w_index;

  d_ptr = data; /* Pre-swap the pointers. */
  b_ptr = buff;
  for (q = 1; q <= log4n; q++) { /* For each pass through the data */
    ell = (int) pow( 4.0, (double) q );
    r = nn / ell;
    ell_star = ell / 4;
    r_star = 4 * r;
    /* Swap data/buff pointers; effectively swapping the input and
       output buffers used below. */
    swap_ptr = d_ptr;
    d_ptr = b_ptr;
    b_ptr = swap_ptr;

    for (j=0; j < ell_star; j++) {
      w_index = 2 * j * r;
      wr = w_tab[ w_index++ ];
      wi = w_tab[ w_index   ];
      w2r = wr * wr  - wi * wi;       w2i = wr * wi  + wi * wr;
      w3r = wr * w2r - wi * w2i;      w3i = wr * w2i + wi * w2r;
      for (k=0; k < r; k++) {
        index = 2*(j * r_star + k);
              ar = *(b_ptr + index );
              ai = *(b_ptr + index + 1);
              index += 2 * r;
              br = wr * *(b_ptr + index ) - wi * *( b_ptr + index + 1 );
              bi = wi * *(b_ptr + index ) + wr * *( b_ptr + index + 1 );
              index += 2 * r;
              cr = w2r * *(b_ptr + index ) - w2i * *( b_ptr + index + 1 );
              ci = w2i * *(b_ptr + index ) + w2r * *( b_ptr + index + 1 );
              index += 2 * r;
              dr = w3r * *(b_ptr + index ) - w3i * *( b_ptr + index + 1 );
              di = w3i * *(b_ptr + index ) + w3r * *( b_ptr + index + 1 );
              t0r = ar + cr;t0i = ai + ci;
              t1r = ar - cr;t1i = ai - ci;
              t2r = br + dr;t2i = bi + di;
              t3r = br - dr;t3i = bi - di;
              index = 2 * (j * r + k);
              *(d_ptr + index )=t0r + t2r;*(d_ptr + index+1) = t0i + t2i;
              index += 2 * r * ell_star;
              *(d_ptr + index )=t1r + t3i;*(d_ptr + index+1) = t1i - t3r;
              index += 2 * r * ell_star;
              *(d_ptr + index )=t0r - t2r;*(d_ptr + index+1) = t0i - t2i;
              index += 2 * r * ell_star;
              *(d_ptr + index )=t1r - t3i;*(d_ptr + index+1) = t1i + t3r;
      }
    }
  }
}
```

```
#if (MAIN)  /* turn on for stand-alone testing. */
#define LEN (1024*16)
/* Note that the base-4 log is half of the base-2 log. */
#define LOGLEN (5+2)
#define TESTLOOPS 1

void main( void )
{
  float data[ LEN * 2 ], work[ LEN * 2 ], w_tab[ LEN / 2 ];
  int j, k;

/* Synthesize some input data. */
  for (j = 0; j < LEN; j++) {
    data[2*j]   = 0.0; /* cos( j * M_PI / 2.0 ); */
    data[2*j+1] = 0.0; /* sin( j * M_PI / 2.0 ); */
  }
  data[3] = 1.0f;
  init_table( w_tab, LEN );
  for (k = 0; k < TESTLOOPS; k++) {
    /* Since str4flip does not scale its result, recursive application of
       str4flip to a buffer quickly leads to numeric overflow processor
       exceptions. Re-create the data vector to test with many loops.
       Make sure loops are not identical, or optimizer may prevent
       looping! */
    data[2] = k;
    str4flip( data, work, w_tab, LEN, LOGLEN );
    /* Examine results (not shown). */
   }
 }
#endif /* MAIN switch, for stand-alone testing. */
```

B. Cxtranspose: Efficiently transpose an array of complex elements

```
/*  Implements an out-of-place transpose of a rectangular complex data
array. The dimensions of the array must be integer multiples of
BLOCK_DIM, below. For a 1024x1024 array, the time required to perform the
transpose is 0.6 sec (with 32x32 block size). A straightforward algorithm
took 2.1 sec.*/

#include <stdio.h>
/*  This version transposes square blocks, to improve cache performance.
    BLOCK_DIM of 16 and 32 are about equal, but 64 is worse. */
#define BLOCK_DIM 16
void cxtranspose( float in_data[], float out_data[],
          const int rows, const int cols )
{
  int ii, jj, kk, mm;/* General loop counters. */
  float *in_ptr[BLOCK_DIM];/* Array of input pointers. */
  float *out_ptr[BLOCK_DIM];/* Array of output pointers. */
       /* In_ptrs point to the first columns of the first N rows. */
  for (kk = 0; kk < BLOCK_DIM; kk++)
    in_ptr[ kk ] = &in_data[ 2 * cols * kk ];
       /* For each block-row... */
  for (jj = 0; jj < (rows / BLOCK_DIM); jj++ ) {
       /* Out_ptrs point to left edge of current block-column.*/
    out_ptr[0] = &out_data[2 * BLOCK_DIM * jj];
    for (mm = 1; mm < BLOCK_DIM; mm++)
      out_ptr[ mm ] = out_ptr[ mm-1 ] + 2 * rows;
       /* For each block in the block-row... */
    for (ii=0; ii < (cols / BLOCK_DIM); ii++) {
       /* For each row in the block... */
      for (kk=0; kk < BLOCK_DIM; kk++) {
       /* For each column in the block... */
        for (mm=0; mm < BLOCK_DIM; mm++ ) {
       /* Copy data from one row pointer to each column pointer. */
          *out_ptr[mm]++ = *in_ptr[kk]++; /* Real part. */
          *out_ptr[mm]++ = *in_ptr[kk]++; /* Imag part. */
       }
      }
       /* Done with a block; move the out_ptrs down to the next block in
           the block-column. (The in_ptrs just increment into the next
           block in the block-row.) */
      out_ptr[0] += 2 * BLOCK_DIM * (rows - 1);
      for (mm = 1; mm < BLOCK_DIM; mm++)
        out_ptr[ mm ] = out_ptr[ mm-1 ] + 2 * rows;
    }
       /* Done with a block-row; move in_ptrs to next block-row. */
    for (kk = 0; kk < BLOCK_DIM; kk++)
      in_ptr[ kk ] += 2 * cols * (BLOCK_DIM - 1);
  }
}
```

C. Fact-T: A large FFT algorithm

```
/*
  This program builds on the str4flip Stockham radix-four algorithm to
  build a large FFT, using the ideas in Section 3.3 of Van Loan.

  Note that Van Loan's terminology assumes Fortran's array organization,
  where successive rows within a column are adjacent in memory. In this
  ANSI-C program, successive elements are assumed to lie in the same row.
  */
#include <stdio.h>
#include <math.h>
/* M_PI is defined in math.h */

/* Declare the FFT and complex-transpose external routines. */
extern void str4flip( float * data, float * work,
             const int length, const int log4_length );
extern void cxtranspose( float * in_data, float * out_data,
             const int rows, const int cols );
#define LEN1 256
#define LEN2 4096
#define LEN (LEN1 * LEN2)
#define MAXLEN LEN2 /* whichever is bigger */
#define LOGLEN1 4
#define LOGLEN2 6
#define TESTLOOPS 2
/*
  Initialize the block-multiply twiddle-factor array. Note that the sign
  on the "sin" term is negative, consistent with Van Loan's text.
  */
void init_twiddle( float tw[], const int len1, const int len2 )
{
  int ii, jj, index;
  double temp;

  temp = 2.0 * M_PI / ((double) (len1 * len2));
  for (ii = 0; ii < len1; ii++ ) {
    for (jj = 0; jj < len2; jj++ ) {
      index = ii * len2 + jj;
      tw[ 2 * index    ] = +cos( temp * ii * jj );
      tw[ 2 * index + 1] = -sin( temp * ii * jj );
    }
  }
}
```

```
/* Apply the block-multiply constants. */
void use_twiddle( float data[], float tw[], const int len1, const int
len2 )
{
  int ii, jj, index;
  float temp;
  for (ii = 0; ii < len1; ii++ ) {
    for (jj = 0; jj < len2; jj++ ) {
      index = 2 * (ii * len2 + jj);
      temp             = data[ index     ] * tw[ index ] -
       data[ index + 1 ] * tw[ index + 1 ];
      data[ index + 1] = data[ index + 1 ] * tw[ index ] +
       data[ index     ] * tw[ index + 1 ];
      data[ index ] = temp;
    }
  }
}

void main( void )
{
  float data[    LEN * 2 ]; /* two floats per complex */
  float data2[   LEN * 2 ]; /* transposed */
  float twiddle[ LEN * 2 ];
  float work[ MAXLEN * 2 ]; /* The greater of LEN1 and LEN2. */
  int j, k;
  init_twiddle( twiddle, LEN1, LEN2 ); /* operates on transpose */
  for (k = 0; k < TESTLOOPS; k++) {
       /* Read (or synthesize) test data (deleted) */
       /* Now, compute the transform. */
    cxtranspose( data, data2, LEN2, LEN1 );
    for (j = 0; j < LEN1; j++) {
      str4flip( &data2[ j * 2 * LEN2 ], work, LEN2, LOGLEN2 );
    }
    use_twiddle( data2, twiddle, LEN1, LEN2 );
    cxtranspose( data2, data, LEN1, LEN2 );
    for (j = 0; j < LEN2; j++) {
      str4flip( &data[j * 2 * LEN1], work, LEN1, LOGLEN1 );
    }
    cxtranspose( data, data2, LEN2, LEN1 );
    /* Output (or check) result (deleted). */
  }
}
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Ft. Belvoir, Virginia

2. Dudley Knox Library 2
   Naval Postgraduate School
   Monterey, California

3. Loomis, Herschel H., Code EC/Lm. 1
   Naval Postgraduate School
   Monterey, California

4. Butler, Jon, Code EC/Bu. 1
   Naval Postgraduate School
   Monterey, California

5. Knorr, Jeffrey B., Code EC/Ko 1
   Naval Postgraduate School
   Monterey, California

6. National Cryptologic School. 3
   Ft. Meade, MD