Book 2—Cookbook

Part D:

Optimizing TriMedia Applications

TriMedia

Version 2.1

Table of Contents

Chapter 10 Porting and Optimizing Programs

Chapter 10

Porting and Optimizing Programs

Note
The examples contained in this chapter at time of publishing are obsolete.
Check release notes for additional information.

Introduction

This chapter provides guidelines for porting and optimizing performance tuning. It describes various optimization methods supported by the TriMedia Compilation System as well as techniques for exploiting the fine-grain parallelism of the TriMedia architecture. This version of the cookbook differs from previous versions because of the new version 2.0 compilation tools. The new compiler is capable of doing optimizations automatically, while in the previous version they had to be done by the programmer. These changes are summarized at the end of this chapter.

Porting Considerations

You should use ANSI Standard C when developing applications for TriMedia processors. The implementation of the TriMedia C compiler is based on the following standards:

- *American National Standard for Programming Languages—C*, ANS X3.159-1989 and ISO/IEC 9899:1990
- *Technical Corrigendum 1 (1994) to* ISO/IEC 9899:1990

Additionally, the compiler supports the concept of *restricted pointers*, as proposed by the *Numerical C Extensions Group* in X3J11/95-049, WG 14/N448

This document is available from ftp: //ftp.dmk.com.DMK/sc22wg14/c9x/aliasing. Book 4, Part A, of *Software Tools* discusses compatibility issues, C language extensions, and implementation-dependent features.

Library and System-Calls Support

The language implementation supports the standard C library, as defined in the ANSI/ISO C Standard. No other libraries are supported. For example, programs using X11 libraries or Sun-specific libraries do not compile with the TriMedia Compilation System.

The following library and system calls are implemented as traps by simulator **tmsim**; that is, **tmsim** uses the corresponding library and system call routine on the host processor to simulate the routine.

```
_close  _fstat  _isatty  _link   _lseek  _mktemp,
_open   _read   _unlink  _write  getenv  time
```

The system call names all begin with "_" because of ANSI C Standard name space requirements. Because many traditional C programs use system call names without a leading "_" (for example, **read** rather than **_read**), the C library includes stubs that perform the desired renaming (for example, defining **read**, which simply executes **_read**). You should always include the appropriate header file (<fcntl.h> for **open**, <sys/stat.h> for **fstat**, and <unistd.h> for the remaining system calls) when compiling a program that uses system calls directly.

Floating-Point Computations

All floating-point data types (**float, double,** and **long double**) are single precision and have the same range of values in the current compiler and TriMedia processor. Therefore, you should use **float** instead of **double** or **long double**.

You should be aware that the results of floating-point computations performed on a Sparc or other workstation can differ from the results of computations performed on a TriMedia processor or simulator. Many compilers automatically convert **float** to **double** during expression evaluations and function calls, especially when the compiler cannot find the function prototype.

File I/O

Applications should employ batch processing and use only file-based input. Output can be sent to the standard output stream, to the standard error stream, or to files.

Performance Tuning

Use the following techniques and tools to improve program execution times:

- Profile-driven compilation
- Loop unrolling
- Fine tuning of grafting
- Dirty float option
- Decision-tree grafting
- Restricted pointers
- Cache optimization
- Loop optimization
- Custom operators
- Unsafe alias analysis
- Cache instructions

The measure used to determine the performance of the following examples is the number of clock cycles required to perform a certain function. A general optimization goal is to minimize this number. This can be achieved by increasing the so-called instruction level parallelism of the code (later on referred to as ILP). The TriMedia processor is capable of executing up to five operations concurrently. The ILP tells how many useful operations are executed on average. If the profiling tool **tmprof** is used with the **-detail** option, the ILP is reported in the last column of the profiling result.

Some optimization techniques may result in an increased code size, which means that the overhead of a function increases, because the code needs to be loaded into the instruction cache before its execution. Several optimization trade-offs are discussed in the following examples.

The function in Figure 1 computes the convolution of two character arrays. The four drawings in the comment shows what is happening.

The **a** array has 400 and the **b** array has eight elements (first drawing). The a and b arrays are scanned in reverse order (second drawing). Conceptually, the **b** array is slid past the **a** array multiplying element by element (third drawing). As shown in the fourth drawing, it is necessary to pad the array on the left side of the **a** array to implement sliding. The **a**

array is indexed from –7 to 399. The right padding shown is needed for the optimizations that will be explained subsequently.

Although faster algorithms for computing the convolution exist, the code demonstrates the utility of profiling, grafting, loop unrolling, and custom operators. A number of different transformations of this convolution function using the listed techniques for improving the performance are presented throughout this chapter. The full program, including the different versions of the convolution function, is included in the software release directory *examples*.

We start by making optimal use of the processor's computing resources. In particular, we increase the level of parallelism by enlarging the number of operations in decision trees and by removing irrelevant dependencies between these operations. The required techniques are grafting, loop unrolling, and improving the compiler's alias analysis with restricted pointers.

When you use these techniques, you might reach the stage at which the processor is saturated. The processor's computing resources—the number and configuration of the available functional units—limit application performance.

This performance limit applies to the application only as it is formulated by you and compiled by the compiler. To further improve performance, you must either find another implementation (change the formulation of the algorithm) or invoke the global optimizer (change the way the application is compiled).

```
/*                            fir1.c -- (part)
 *
 * convolution of 2 8-bit integer arrays (a & b) of length 400 & 8, resp.
 * Rough pictorial description of the process.
 *
 *              |--------|--------|--------|--------|  a[400]
 *
 *              |--------|                           b[8]
 *
 *              |--------|--------|--------|--------|  a
 *
 *     |--------|                                     time reversed b
 *
 *              |--------|--------|--------|--------|  a
 *
 *                 |--------|                        time reversed
 *                                                   and sliding b
 *
 * Increase the length of array a so that vector of length 8 could be
 * prepended and appended. With this additional zeros, separate handling
 * of beginning and end of data is avoided.
 *
 *   |00000000|---------|---------|---------|---------|00000000|
 *
 *            |<—   Original length 400 array a  ——>|
 *
 * These arrays hold the result of convolutions. Actual required output
 * array length = 400 + 8 - 1 = 407, but our modified algorithm calculates
 * one unnecessary element. To handle this, output array length has been
 * increased by 1 */
```



```
#define NROF_SAMPLES                400

void
direct_convolution( char *a, char *b, int *c ){
   int        k, j;

   for( k = 0; k < NROF_SAMPLES; k++ ){
      c[k] = 0;
      for(j = 0; j < 8; j++)
         c[k] += b[j] * a[k - j];   /* a is shifted 8 in the call*/
   }
}
```

Figure 1 Convolution Example (Part of Example firl.c)

Profile-Driven Compilation

The TriMedia Compilation and Simulation system facilitates the *compile-profile-recompile* cycle of performance tuning. First, you compile the program using the compiler driver **tmcc** with the **-p** option (write profiling information to file dtprof.out). Next, you simulate program execution using the machine-level simulator **tmsim**. Then, you recompile the program with **tmcc**, using either the **-r** option (profile-driven compilation) or the **-G** option (profile-driven compilation with grafting).

Note
Profiling also works on the hardware.

The first optimization step is to obtain profile information about the program and identify the most frequently executed parts of the program. After the most frequently executed parts of the program are identified, you can perform grafting and loop unrolling and can use restricted pointers to remove spurious dependencies. (Function inlining is not currently supported.)

The following procedure illustrates both how to perform profiling, and how to use **tmprof** to summarize execution statistics:

1. Compile the source modules using **tmcc** with the **-p** option. Do not use the **-G** and **-r** options at this stage. (With the **-p** option, the compiler instruments the user program with code to determine decision-tree execution counts and branch probabilities.)
2. Use **tmsim** to simulate the instrumented program, which generates the profile information in file dtprof.out**.** Use the option **-nomm** to switch off the simulation of the memory model and save execution time.
3. Recompile the source modules *with* the **-r** option and *without* the **-p** option. This causes the generated decision trees to be free of profiling code. Assuming the input was representative, recompilation based on profiling adds branch probabilities in the new decision-trees file. It is important not to change the source code because the profiling information is based on the control-flowgraph of the program. When the source code changes during the generate-profile and read-profile compilations, the profiles do not match and the profile is ignored.

4. Run **tmsim** with the **-statfile** option to save the execution statistics and since -nomm is not used, memory mode is simulated.
5. Run **tmprof** with the **-func** option to generate a report for each function in the program. The **-scale 1** option tells **tmprof** to report the cycle count without scaling.

The following commands generate a summary report for the program fir1.c:

```
tmcc -p fir1.c -o fir1           /* Generate program with profiling on.*/
tmsim -nomm fir1                 /* Simulate interm code & gen. dtprof.out.*/
tmcc -r fir1.c -o fir1           /* Recompile using profile information.*/
tmsim -statfile fir1.stat fir1   /* Simulate & collect accurate cycle info.*/
tmprof -scale 1 -func fir1.stat  /* Output is sent to stdout.*/
```

The report produced by this sequence of commands is as follows. Note that the values printed differ depending on the version of the TCS:

Function	Executions	Total Cycles	MIPS	I-Cycles	D-Cycles
_direct_convolution	1	29322	67.89	58	460
_main	1	102	0.24	87	0
total/average	226	43188	100.00	10112	1483

The report shows only the functions that belong to the program code shown. The real tmprof output contains several more functions including the startup and library functions. Their contribution is about 14000 clock cycles in this case. They are omitted for because they are not subject to optimization. The tmprof output shows the total number of cycles executed for each function, and the stall cycle contribution of both the instruction-cache and data-cache.

Because the function **direct_convolution** in the source module fir1.c takes about 68% of the total cycles, it is the one to be optimized. The next sections show how to get a further performance gain.

If one compares this simulation result with the corresponding one from the version 1.1 compiler, it can be seen that the function direct_convolution is 14% faster now. The reason for this is that the new compiler performs automated loop unrolling. The resulting code consists of larger basic blocks and imposes less overhead due to jumps.

Grafting Based on Profile Information

Grafting increases parallelism within decision trees. This technique replaces any jump with a copy of the destination tree and thus “grows” larger decision trees. As a result, the program size increases.

The core compiler **tmccom** generates an intermediate representation of a program known as a decision-tree representation[1]. Decision trees are derived from basic blocks. A basic block is a sequence of instructions with no jumps into it, except to the first instruc-

1. You can generate an example decision tree by compiling a program using **tmcc** with the -t option. **tmcc** -t foo.c produces a file foo.t with machine-like operations, see Chapter 4, *Using the Instruction Scheduler*, of *Software Tools*.

tion and no jumps out except at the last instruction. Basic blocks are connected to one another by conditional or unconditional jumps. It is well known that basic blocks derived from typical C code are small and not much parallelism within basic blocks exists to be exploited. Furthermore, frequent branching behavior would result in underutilization of processor resources.

A decision tree is similar to a basic block in that the decision tree can be entered only at the beginning. However, a decision tree can have multiple exits. (Chapter 4, *Using the Instruction Scheduler,* of *Software Tools* defines the syntax and semantics of decision trees.) Decision trees are larger than basic blocks and potentially have more fine-grain parallelism that can be exploited during optimization.

Figure 2 shows a decision tree ending in a branch. The actual operations in the tree are not important for this example. The decision tree **__ip_DT_1** has two exits, one leading back to itself (**gotree {__ip_DT_1}**) and the other leading to another decision tree (**gotree {__ip_DT_2}**).

```
{__ip_DT_1:}
tree (50)
        2 rdreg (12);
        1 ld32 2;
        4 rdreg (11);
        6 rdreg (10);
        7 ld32x 6 4;
        9 rdreg (9);
        10 ld32x 9 4;
        11 imul 7 10;
        12 iaddi(1) 11
        13 st32 2 12
                after 10 7 1;
        14 iaddi (1) 4;
        15 wrreg (11) 14
                after 4;
        16 ilesi (50) 14;
        if 16 (0.980000) then
                gotree {__ip_DT_1}
        else (16)
                gotree {__ip_DT_2}
        end (16)
endtree (*__ip_DT_1*)
```

Figure 2 Example of a Decision Tree Ending in a Branch

Notice the back edge from **__ip_DT_1** to itself has a probability of 0.98. This statistic is derived from a profiling run. The compiler can do a better job of grafting if it has information about decision-tree execution counts and branch probabilities. In this case, the decision tree **__ip_DT_1** has an execution count of 50 (the first number after the label). If grafting is enabled, the compiler replaces the instruction "**gotree {__ip_DT_1}**" with a copy of the tree **__ip_DT_1**, doubling the size of the decision tree **__ip_DT_1**.

shows a schematic of the same tree after it is grafted. The scheduler can decide where to place code and when to use guarded execution if it has information about branch proba-

bilities. You can also guide the compiler in its grafting decisions, discussed later in the section.

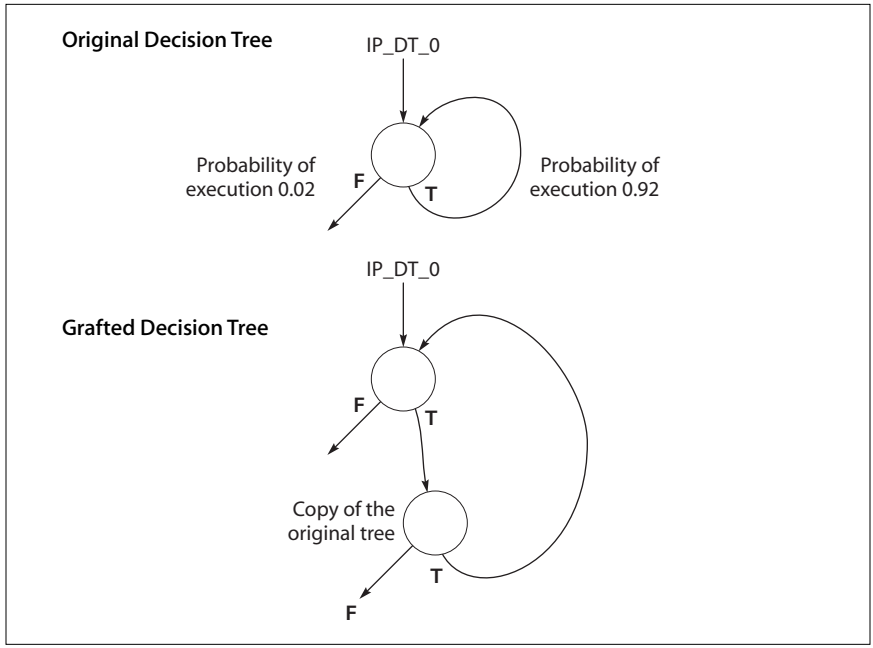

Figure 3 Decision Tree After it is Grafted

It is important to note that grafting is a code-replication technique that eliminates branches but increases the code size. It is a technique similar to loop unrolling, but does not reduce the overhead of the loop as manual loop unrolling can. This is shown later on.

You can improve performance of program fir1.c by grafting after profiling. The following procedure performs the compile-profile-recompile cycle with grafting enabled after profiling.

```
tmcc -p fir1.c -o fir1   /* Generate program with profiling on */
tmsim -nomm fir1         /* Simulate interm code and produce dtprof.out */
tmcc -G fir1.c -o fir1   /* Recompile using profile & perform grafting */
tmsim -statfile fir1.stat fir1   /* Simulate & collect cycle accurate info */
tmprof -scale 1 -func fir1.stat  /* Output is sent to stdout */
```

The report produced is as follows:

Function	Executions	Total Cycles	MIPS	I-Cycles	D-Cycles
_direct_convolution	1	28028	66.52	377	447
_main	1	102	0.24	87	0
total/average	226	42133	100.00	10605	1498

We discover that the execution time of the grafted version of direct_convolution improved by 4.5% over the ungrafted version. However, the number of stall cycles in the instruction cache increased, which is due to the increase in code size. We discover this when we list the size information of fir1.o with **tmsize** the text size increased from 712 bytes to 1855 bytes. It appears that grafting is a valuable tool for easy performance increments, but that a trade-off has to be made between performance gain and code size increase. *Graft-Tuning Parameters* on page 37 describes how grafting can be customized with a grafting parameter file. Note that in the previous versions of the documentation a performance gain of 18% was achieved by applying grafting. The version 2.0 compiler performs so called automated loop unrolling which compensates the advantages of grafting. In many cases it makes grafting unnecessary.

Loop Optimization

You perform loop optimization by moving critical code off the control flow path so that the inner loops of the program can be reduced to a single decision tree. This section describes several techniques to achieve loop optimization, such as loop nesting, using **goto**s, and **dtree** breaks. Most of the techniques described here are automatically performed by the compiler. This section gives you the information you need to give you control over the number of decision trees when required.

Remove If Statements and Conditional Expressions

Frequently, you can transform code to eliminate **if** statements. For example

```
if( p->data < v ) cnt = cnt + 1;
```

can be replaced by

```
cnt = cnt + (p->data < v);
```

There is a conditional expression in the following preprocessor macro:

```
#define abs(v) ((v) < 0 ? -(v) : (v))
```

You can eliminate it by using a TriMedia custom operation as follows:

```
#include <ops/custom_defs.h>
...
#define abs(v) IABS(v)
```

The following preprocessor macro clips a floating point value between 10^{-38} and 10^{38}:

```
#define THRESHLO   1e-38
#define THRESHHI   1e38
#define MINFLOAT(x, y) ((x) < (y) ? (x) : (y))
#define MAXFLOAT(x, y) ((x) > (y) ? (x) : (y))
#define CLIP(x) MAXFLOAT(MINFLOAT(x, THRESHI), THRESHLO)
```

Assuming the values are positive, you can eliminate the conditional expressions as follows:

```
#include <ops/custom_defs.h>
...
#define MINFLOAT(x, y) FMIN(x, y)
#define MAXFLOAT(x, y) FMAX(x, y)
#define CLIP(x) MAXFLOAT(MINFLOAT(x, THRESHI), THRESHLO)
```

Table 1 provides a list of such transformations. Most transformations are applied automatically by the compiler. See notes 9 and 10.

Table 1 Code Transformation

Original Code	Transformed code	Notes
if (e_1) v += e_2	v += INONZERO(e_1, e_2)	1,2,4,9
if ($e_1 < e_2$) v += 1	v += $e_1 < e_2$	1,2,5,6,9
if (e_1) v = 0;	v = INONZERO(e_1, v)	1,2,9
(e_1 ? e_2 : 0)	INONZERO(e_1, e_2)	9
if (e_1) v = e_2	v = INONZERO(e_1, v–(e_2)) + (e_2)	1,2,9
(e_1 ? e_2 : e_3)	(e_2 + INONZERO(e_1, (e_3)–(e_2)))	1,2,9
if (e_1) v = –v	v = IFLIP(e_1, v)	9
(e_1 != 0 ? –e_2 : e_2)	IFLIP(e_1, e_2)	9
if (v < 0) v = –v	v = IABS(v) v = FABS(v)	10
(e_1<0 ? –e_1 : e_1)	IABS(e_1) FABS(e_1)	10
(e_1<e_2 ? e_1 : e_2)	IMIN(e_1, e_2) FMIN(e_1, e_2)	**FMIN** values must be positive
(e_1>e_2 ? e_1 : e_2)	IMAX(e_1, e_2) FMAX(e_1, e_2)	**FMAX** values must be positive
	UMAX(e_1, e_2)	7
max(min(e_1,e_2), ~e_2)	ICLIPI(e_1, e_2)	8

Notes:

1. e2 should not contain side effects.
2. v must contain no side effects. If it is an indirection, the address must be valid.
3. float e1, e2. Values must be non-negative.
4. idem –=, *=, &=, |=, <<=, >>=, ...
5. idem –=, *=, <<=, >>=, ++, --.
6. idem >, >=, <=, ==, !=, &&, ||.

7. unsigned e1, e2.
8. Clips to ~e2 .. e2.
9. This transformation is performed automatically by the compiler's **if**-conversion algorithm.
10. The compiler performs this automatically except when the argument types are floating point.

Parallel Reduction Loops

The loop of Figure 4 is called a *reduction* because it reduces the dimension of a vector. Many loops in multimedia and DSP applications are reductions. Operations such as computing a vector sum or product are reductions. Scalar product computations are reductions also.

```
#include <stdio.h>
#include <ops/custom_defs.h>
float vecmax(float *a, int size) {
   float max = a[0];
   int i;
   for (i=1; i<size; i++) {
      max = fmux(a[i] > max, a[i], max);
   }
   return max;
}
```

Figure 4 Maximum with FMUX

The program is using the floating point pseudo operation fmux which returns either the second or third argument, depending on whether the first argument is true. Its functionality can be compared to the C language A ? B : C operator. Use mux(A, B, C) for non floating point types.

Unrolling is of limited effectiveness on reduction loops because of the loop-carried dependence on the scalar variable (for example, max). Unrolling the loop of Figure 4 four times produces less improvement in performance than using grafting (384 versus 338 cycles). You can optimize reductions by using the mathematical laws of commutativity and associativity to reorganize the order of computation.

In the program of Figure 5, four copies have been introduced for the reduction variable. Four independent maximums are computed on four slices of the vector. You can reduce the four results to a single operation using three FMUX operations. Table 2 compares the performance of the two loops.

```
#include <ops/custom_defs.h>
float vecmax(float *a, int size) {
   float max0 = a[0], max1 = a[1], max2 = a[2], max3 = a[3];
   int i;
   for (i=4; i<size; i+=4) {
      max0 = fmux(a[i] > max0, a[i], max0);
      max1 = fmux(a[i+1] > max1, a[i+1], max1);
      max2 = fmux(a[i+2] > max2, a[i+2], max2);
```

```
      max3 = fmux(a[i+3] > max3, a[i+3], max3);
   }
   max0 = fmux(max0 > max1, max0, max1);
   max2 = fmux(max2 > max3, max2, max3);
   return fmux(max0 > max2, max0, max2);
}
```

Figure 5 Reduction Variable Copying

If you know the vector values to be non-negative use **FMIN**, **FMAX** and **IMIN**, **IMAX** instead of **FMUX** and **MUX;** this saves instructions. Calculate integer minimum and maximum using **IMIN** and **IMAX**.

Floating point addition is not associative. You should not optimize reductions using floating point addition if the result of the transformation must be bit exact.

Table 2 Time to Calculate Maximum of 100 (1000) Element Vector

	Total Cycles	**I$ Cycles**	**Per Element**
Reduction Using one **FMUX**	469 (2733)	203 (203)	4.69 (2.73)
Reduction Using four **FMUX**	313 (2338)	87 (87)	3.13 (2.34)

Another interesting information that can be obtained from Table 2 is the influence of the instruction cache. If a 100-element vector is used more cycles per element are required than for a larger vector because the relative contribution of the initial load of the code into the cache is higher.

Use MUX on Variable Length Loops

The program of Figure 5 was unrolled four times. If the vector length n is not a multiple of the loop step m, the program does not work. There are a number of variable elements equal to the remainder, $n \bmod m$.

You can deal with the variable elements by exploiting the mathematical properties of a group. The identity i of a group is such that $x \; op \; i = x$ for all elements. The intent is to round up the number of elements to the loop step by appending values equal to the identity element. For example, if a vector sum is being computed we round up by appending trailing zeroes ($x + 0 = x$). If a vector product is being computed, we round up by appending trailing ones ($x \times 1 = x$). To round up the vector, $(n \bmod m)$ elements need to be added.

```
#include <ops/custom_defs.h>
#include <float.h>
#define STEP 4
float vecmax(float *a, int size) {
   int i, adj;
   float max0, max1, max2, max3;
   adj = size & (STEP-1);
   size &= ~ (STEP-1);
   max0 = fmux(adj>0, a[size],   -FLT_MAX);
   max1 = fmux(adj>1, a[size+1], -FLT_MAX);
   max2 = fmux(adj>2, a[size+2], -FLT_MAX);
```



```
    max3 = -FLT_MAX;
    for (i=0; i<size; i+=STEP) {
       max0 = fmux(a[i] > max0, a[i], max0);
       max1 = fmux(a[i+1] > max1, a[i+1], max1);
       max2 = fmux(a[i+2] > max2, a[i+2], max2);
       max3 = fmux(a[i+3] > max3, a[i+3], max3);
    }
    max0 = fmux(max0 > max1, max0, max1);
    max2 = fmux(max2 > max3, max2, max3);
    return fmux(max0 > max2, max0, max2);
}
```

Figure 6 Unrolling Variable-Length Loops

For a loop step of four, this corresponds to one to four elements. Figure 6 shows code for calculating the maximum of a vector of arbitrary length. For each of the reduction variables (**max0**, **max1**, **max2**, **max3**), you need to make a selection between an element at the end of the vector and the identity element. You can do this with an **FMUX**. The initialization code before the **for** does this. The identity for the operation being calculated (the maximum) is negative infinity ($max(x,-\infty) = x$). The identity for the minimum is ($+\infty$).

In Figure 6, the loop step must be a power of two. This allows a bitwise and (**&**) to replace a modulus operation. The **&** has a single cycle latency. The number of iterations needs to be rounded down to a multiple of the loop step. You can do this with an **&** also. Integer division and modulus are 50 times slower.

Many DSP kernels are sum reductions for which the identity is zero. In such cases, you should use the TriMedia custom operation **IZERO** to initialize the reduction variables (**max0**, **max1**, **max2**, **max3**) instead of **MUX** and **FMUX**. It selects between zero and a value in one instruction. Use **FZERO** for floating point types.

Table 3 compares performances of a 100-element (1000-element) vector maximum for a four-way unrolled loop. Thirty-one cycles of overhead are necessary to deal with the remaining elements.

Table 3 Time to calculate Maximum of 100 (1000) Element Vector

	Total Cycles	I-Cache Cycles	Per Element
Length a Multiple of Four	313 (2338)	87 (87)	3.13 (2.34)
Arbitrary Length	344 (2369)	116 (116)	3.44 (2.37)

Apply Strength Reduction

Figure 7 following shows two procedures to normalize a vector. In the program to the left, the individual elements are divided by the sum of values. For 100-vector elements, 2964 instruction cycles are necessary for the program with global optimization. Floating

point division requires 17 cycles on TriMedia. The divisions correspond to 1700 of the cycles.

```
norm( float *a, int size ){
   int i;
   float sum = 0.0;
   for( i=0; i<size; i++ )
       sum = sum + a[i];
   for( i=0; i<size; i++ )
      a[i] = a[i] / sum;
}
```

```
norm( float *a, int size ){
   int i;
   float sum = 0.0, invsum;
   for( i=0; i<size; i++ )
      sum = sum + a[i];
   invsum = 1.0 / sum;
   for( i=0; i<size; i++ )
       a[i] = a[i] * invsum;
}
```

Figure 7 Vector Normalization Procedures

Floating point multiplication instead requires only three cycles and a new multiplication can be started in every clock cycle. In the program to the right, the division is replaced by a multiplication by the reciprocal. Doing so saves 14 cycles (17–3) per division. You can calculate the reciprocal using a single division outside the loop. 99 of the 100 divisions can be replaced by a multiplication. This corresponds to a reduction in the total execution time of 1386 cycles (99×14). An optimization such as this, which replaces a costly operator by a less expensive one, is called *strength reduction*. The result of the reciprocal followed by the multiplication can vary from the division in the low order bit.

Table 4 compares the performance of vector normalization with division and with multiplication. It reflects the overhead of the code loading into the instruction cache by comparing the numbers of cycles required to normalize a 100 and a 1000 elements vector.

Table 4 Time to Normalize 100 (1000) Elements (Floating Point)

	Total Cycles	**Per Element**
Normalization Using Division	2964 (24981)	29.6 (25.0)
Normalization by Multiplication by the Reciprocal	1489 (10906)	14.9 (10.9)

Figure 8 shows two programs to calculate the greatest common divisor (g.c.d.). The program on the left calculates the g.c.d. using integer arithmetic.

```
int gcd( int u, int v ){
   unsigned t;
   if( u > v ){
      t = u; u = v; v = t;
   }
   while( u > 0 ){
      t = u;
      u = v % u;
      v = t;
   }
   return v;
}
main(){
   (void)gcd( 12381203,41231207 );
}
```

```
int gcd( int u, int v ){
   int t;
   if( u > v ){
      t = u; u = v; v = t;
   }
   while( u > 0 ){
      t = u;
      u = v-(int)((float)v/u)*u;
      v = t;
   }
   return v;
}
main(){
   (void)gcd( 12381203,41231207 );
}
```

Figure 8 Two programs to calculate greatest common divisor

There are 791 cycles necessary to calculate the g.c.d. Most of the time is spent in the subroutine **_rt_imod**. This subroutine calculates the remainder for signed integers. 605 cycles are spent for 10 executions of **rt_imod**, including 145 instruction cache cycles. The **rt_umod** subroutine calculates the remainder for unsigned integers.

The program to the right uses floating point arithmetic. Calculating the remainder in floating point requires 24 cycles. Seventeen cycles are required for the division, three are required for the floating-point-to integer conversion, three are required for the multiplication, and one is required for the subtraction. You need 385 cycles (including 58 instruction cache cycles) to calculate the g.c.d. There is a saving of 406 (406=791-385) cycles for ten remainders (40.6 cycles per remainder) using floating point.

The g.c.d. of 12,381,203 and 41,231,207 is one. Both algorithms give the correct value. The g.c.d. of 268,435,454 and 268,435,582 is two. The algorithm to the right of Figure 8 calculates a g.c.d. of 128. The value is incorrect because the values are outside the range $[-2^{24}, 2^{24}]$ (floating-point numbers have a precision of 24 bits).

Figure 9 shows two ways to subsample a vector with a 3:4 ratio. In the program to the left, the array index is calculated uses an integer division and multiplication. The **tmprof** output for subsampling a 100-element vector using the program on the left is shown below.

Function	Executions	Total Cycles	MIPS	I-Cycles	D-Cycles
__rt_idiv	100	4445	24.06	145	0
_subsample	1	954	5.16	435	0
(...)					

```
subsample(char *a, char *b, int n){
   int i;
   for (i=0; i<n; i++ )
      a[i] = b[i*4/3];
}
```

```
subsample(char *a, char *b, int n){
   int i;
   for (i=0; i<n; i++ )
      a[i] = b[(int)(i*(4.0/3))];
}
```

Figure 9 Two ways to subsample a vector with a 3:4 ratio

Most of the time is spent in the subroutine **rt_idiv**. This subroutine implements division for signed integers. 43 cycles are needed per call, not including call overhead. **rt_udiv** implements division for unsigned integers. Note that for the algorithm to the right side

-tmccom -dirty_float 1 --

has to be used for the compilation to replace the quotient (4.0/3) by a constant.

Using the algorithm to the right, 449 cycles are necessary to subsample a 100-element vector and 2922 for a 1000-element vector. The two algorithms compute the same value for $n < 2^{24}$.

In both cases, the use of grafting was recommended in earlier versions of the cookbook. With the version 2.0 compiler, this has changed. The automated loop unrolling makes the grafting superfluous.

Table 5 Time to Subsample 100-Element (1000-Element) Vector

	Total Cycles	**Per Element**
Subsampling with Integer Divide and Multiply	5399 (48677)	54.0 (48.7)
Subsampling Using Floating Point	449 (2922)	4.5 (2.9)

For a variable of **signed** type, replacing a division by 2^n by a shift (>>) eliminates three operations from the program and saves three cycles. The result differs by one from that of the division operator (/) if it is negative and there is a remainder. Replacing $x \% 2^n$ by $(x \,\&\, (2^n - 1))$ eliminates three instructions and saves three cycles. The result is positive or zero. Integer remainder produces a negative or zero result if the result of the division is negative. If the variable is known to be non-negative, changing the type to unsigned obtains the same effect automatically. The results of >> and **&** correspond to the mathematical definitions of division and remainder.

Move Externals and Reference Parameters to Locals

You cannot allocate external variables to registers and require memory references. Accesses have a latency of three cycles. Copying an external variable to a local variable can improve performance substantially in time-critical parts of the code. Figure 10 shows two ways of writing a program to push an array onto a stack. The stack pointer is con-

tained in the external variable **stackp**. 1534 instruction cycles are necessary to push 100 words. Only 753 cycles are necessary if **stackp** is copied to a local variable.

```
int *pusha(int nargs, int *p){
   int *oldstkp;
   extern int *stackp,stack[NSTACK];

/* save the old stack pointer */
   oldstkp = stackp;

/* save each node pointer */
   while ( nargs-- ) {
      if (stackp <= stack)
      abort("stack overflow");
      *--stackp = *p++;
   }
   return oldstkp;
}
```

```
int *pusha(int nargs, int *p){
   int *newstkp, *oldstkp;
   extern int *stackp,stack[NSTACK];

/* save the old stack pointer */
   oldstkp = stackp;
   newstkp = stackp;
/* save each node pointer */
   while (nargs--){
      if (newstkp <= stack)
      abort("stack overflow");
      *--newstkp = *p++;
   }
   stackp = newstkp;
   return oldstkp;
}
```

Figure 10 Pushing an Array Onto a Stack

Table 6 summarizes the performances of both programs shown in Figure 10.

Table 6 Effect of Copying Externals to Locals

	Total Cycles
External Variable in Loop	1534
Local Copy in Loop	755

Remove Function Calls

Figure 11 shows two programs that calculate the square of the distance from the origin for a collection of points, (x_i, y_i). For the program to the left, 1928 cycles are necessary with global optimization alone (**-O3**). 1911 instruction cycles are necessary with global optimization and grafting. The execution time is reduced only by 1 percent with grafting. This is because of the function call. Grafting does not cross function call boundaries. Each invocation of a function also adds a decision tree. In the program on the right, the function call has been inlined using the C front end processor, **tmcfe**. Automated function inlining can be performance by using **-O5** as an argument for the compilation. If just a particular function is to be inlined, this can be achieved with the following line in the code

#pragma TCS_inline=functionName

or

–Xinline=functionName

on the compiler command line. 1199 cycles are necessary without grafting and 1600 are necessary with grafting (an increase of 33%). This example shows that one must be careful with using grafting in combination with function inlining.

An even better result can be obtained by the hand-inlining of the function **hypot**, as shown in Figure 11. Without grafting, it takes 695 cycles to calculate the distance vector and with grafting 669 which is a reduction of 4%. The main reason for the difference between the automated function inlining and the hand-inlining is that the compiler performs automated loop unrolling before the function inlining. Therefore, automated loop unrolling is done when a macro is used, but it is not done if the loop unrolling algorithm detects a function call within the loop.

In the program of Figure 10, the function call to abort corresponds to an error (stack overflow). This occurs very rarely. Although the function does not return, the compiler does not know this. In Figure 12 the function call has been moved outside the loop. This removes a join node, allowing the loop to be represented by one decision tree. 700 instruction cycles are necessary with global optimization and without grafting, compared to 755 previously. The 968 instruction cycles are necessary with grafting. Using grafting makes the code more efficient, but also longer.

The ILP is still limited because of the stores through pointers. Adding the restrict qualifier to the definitions of **newstkp** and **p** reduces the execution time to 618 cycles with grafting. This corresponds to a reduction of 12%. The performances are summarized in Table 7.

Table 7 Performance Summaries

Operation	Without Grafting (I-Cache Cycles)	With Grafting (I-Cache Cycles)
Call + Local Copy in Loop (Table 6)	755 (162)	694 (232)
Local Copy in Loop + Call Outside (Figure 10)	700 (120)	653 (147)
Local Copy in Loop + Call Outside + Restrict	700 (120)	618 (157)
Call in Loop (Figure 11, Left)	1928	1911
Automated Function Inlining (-O5 or #pragma ...)	1199	1500
Inlining with define (Figure 11, Right)	695	669

Table 7 shows that grafting can result in performance decreases if the code size increase is not compensated by the number of cycles saved due to the better IPL of the code.

```
float hypot(float x, float y){
   return x*x + y*y;
}
main() {
   float x[100], y[100], rad[100];
   int i;
   for (i=0; i<20; i++)
      rad[i] = hypot(x[i],y[i]);
}
```

```
#define hypot(x,y) (x)*(x)+(y)*(y)

main() {
   float x[100], y[100], rad[100];
   int i;
   for( i=0; i<20; i++ )
      rad[i] = hypot(x[i],y[i]);
}
```

Figure 11 Program to Calculate Distance Vector


```
int *pusha(int nargs, int *p){
   int *newstkp, *oldstkp;
   extern int *stackp, stack[NSTACK];

/* save the old stack pointer */
   oldstkp = stackp;
   newstkp = stackp;

/* save each node pointer */
   while (nargs-- && newstkp>stack){
      *--newstkp = *p++;
   }
   if (newstkp <= stack) abort("evaluation stack overflow");
   stackp = newstkp;
   return oldstkp;
}
```

Figure 12 Program to Push Arguments on the Stack

Pay Attention to Compile Time

Figure 13 shows a program that multiples a 40 x 10 matrix by a 10 x 20 matrix, giving a 40 x 20 result (**c = a*b**). The source program is 29 lines long. Six minutes, ten seconds are necessary to compile the program.

```
int a[40][10], b[10][20], c[40][20];
main() {
   int i;
   for( i=0; i<40; i++ ){
     c[i][0] = a[i][0]*b[0][0]   + a[i][1]*b[1][0]  + ... + a[i][9]*b[9][0];
     c[i][1] = a[i][0]*b[0][1]   + a[i][1]*b[1][1]  + ... + a[i][9]*b[9][1];
     c[i][2] = a[i][0]*b[0][2]   + a[i][1]*b[1][2]  + ... + a[i][9]*b[9][2];
     c[i][3] = a[i][0]*b[0][3]   + a[i][1]*b[1][3]  + ... + a[i][9]*b[9][3];
                          (...)
     c[i][18] = a[i][0]*b[0][18] + a[i][1]*b[1][18] + ... + a[i][9]*b[9][18];
     c[i][19] = a[i][0]*b[0][19] + a[i][1]*b[1][19] + ... + a[i][9]*b[9][19];
   }
}
```

Figure 13 Matrix Multiply Program matmul_1.c

The **-vtimes** option of **tmcc** reports on the execution times of the individual phases. The **-K** option tells **tmcc** to keep intermediate output files around (that is, matmul_1.t, matmul_1.s, and matmul_1.o). In this case, almost all the time can be observed to be spent in the TriMedia scheduler **tmsched**. On WindowsNT and Windows 95, the program running is shown in the menu bar.

```
$ tmcc -vtimes -K matmul_1.c
cpp:    0.033
tmccom: 0.967
tmsched:360.719
        ...
total:  363.035
```

Almost all the execution time in the program is spent in the decision tree corresponding to the for-loop, $main_1$. You can determine the number of operations in the decision tree by examining the trees output file tmsim.t produced by the **tmcc** command. Scheduling time is nonlinear with respect to the number of operations. There are 1018 operations in $main_1$.

Unusually long scheduling times are typically the consequence of feeding **tmsched** a decision tree with too many operations. In the program below, the two inner loops of the multiplication have been completely unrolled. Decision trees that are too long result in reduced performance due to scheduler spilling. Spilling means that the scheduler runs out of temporary registers and it leads to extra loads and stores. Compile time is a performance indicator.

In the program, each iteration reads a row of matrix a (10 accesses). The matrix b is read entirely (10 x 20 accesses) and a single element $c_{i,j}$ is computed for each column (20 accesses). There are 40 (200 + 10) reads and 20 writes for each of the 40 iterations. The 9200 memory accesses are necessary in total (8400 reads and 800 writes). You can determine the actual number of memory accesses by using **tmsim** with the **-v** option.

```
$ tmsim -v a.out
              (...)
data cache statistics: size 16 kB, blocksize 64 b, associativity 8
nr of accesses:  21946
```

There are 12746 (21946–9200) more operations than expected. Almost all the accesses are 32-bit accesses in the decision tree corresponding to the **for**. You can differentiate spills from nonspills as follows:

```
$ grep ld32 matmul_1.s | wc -l
     324
$ grep st32 matmul_1.s | wc -l
     224
$ grep "ld32.*-- SPILL" matmul_1.s | wc -l
     114
$ grep "st32.*-- SPILL" matmul_1.s | wc -l
     204
```

The number of nonspill accesses is given by subtracting the spill accesses from the number of total accesses. There are 210 (324–114) nonspill loads, and 20 (224–204) nonspill stores per iteration. This corresponds to 9200 accesses in total, as expected. Spilling is responsible for 318 (114+204) memory accesses per iteration. 12720 (318×40) accesses correspond to scheduler spills.

Use #pragma TCS_break_dtree

The interrupt mechanism of TriMedia is discussed in the TriMedia data books. Interrupts only occur when control passes from one decision tree to another. Decision-tree breaks (**#pragma TCS_break_dtree**) limit the length of a decision tree, allowing control over interrupt latency.

They can also be used to improve the performance of a program. Spilling in case of program in Figure 13 is a result of excessive register pressure due to the high degree of

unrolling. In Figure 14, the compiler pragma **TCS_break_dtree** is used to remove spilling. The loop is split into two decision trees.

The performances are summarized in Table 8. Introducing a decision tree break completely removes the spills. The extra nonspill loads are because a row (ten elements) needs to be reread.

```
int a[40][10], b[10][20], c[40][20];
main() {
   int i;
   for (i=0; i<40; i++) {
     c[i][0] = a[i][0]*b[0][0]   + a[i][1]*b[1][0]  + ... + a[i][9]*b[9][0];
     c[i][1] = a[i][0]*b[0][1]   + a[i][1]*b[1][1]  + ... + a[i][9]*b[9][1];
     ...
     c[i][8] = a[i][0]*b[0][8]   + a[i][1]*b[1][8]  + ... + a[i][9]*b[9][8];
     #pragma TCS_break_dtree
     c[i][9] = a[i][0]*b[0][9]   + a[i][1]*b[1][9]  + ... + a[i][9]*b[9][9];
                       (...)
     c[i][18] = a[i][0]*b[0][18] + a[i][1]*b[1][18] + ... + a[i][9]*b[9][18];
     c[i][19] = a[i][0]*b[0][19] + a[i][1]*b[1][19] + ... + a[i][9]*b[9][19];
   }
}
```

Figure 14 Matrix Multiply Program mtmul_1.c Compiler Pragma TCS_break_dtree added

You can also use decision tree breaks to prune infrequently executed branches from a decision tree. In the program of Figure 15, the trailing return statement is included in the decision-tree for the loop. The computations for the return increase the length of the critical path. Inserting a decision-tree break at the end of the loop reduces the execution time per iteration from 7.7 to 6.7 cycles. If grafting is applied, the execution time drops to 5 cycles per iteration regardless if the pragma is used or not.

Table 8 Effect of Decision Tree Break on Unrolled Matrix Multiply (per iteration)

	Non-Spill Loads	Non-Spill Stores	Spill Loads	Spill Stores
without **TCS_break_dtree**	210	20	114	204
with **TCS_break_dtree**	220	20	0	0

```
void compare(int *a, int *b, int size, int *pcount){
   int i;
   int count;
   count = 0;
   i = 1;
   do {
      count = count + (a[i]==b[i]);
   }while (i++ < size);
#pragma TCS_break_dtree
   *pcount = count;
}
```

Figure 15 Loop Pruning

Loop Fusion

The program of Figure 16 calculates the mean and variance for an array of n elements. The mean is equal to the sum of array values divided by the array size. The variance is equal to the sum of squares divided by the array size minus the square of the mean. The first loop calculates the sum of values. The second loop calculates the sum of squares. Loop fusion merges two loops that are executed the same number of times into a single loop. Loop fusion eliminates half the overhead. The program on the right illustrates the application of fusion. Table 9 compares performances for an array of 100 elements. Again, grafting does not improve the performance of the function because the compiler generates better code with its automated loop unrolling algorithm.

```
float meanvar(
   float *a,
   int n,
   float *var
){
   float sum = 0, sumsq = 0;
   float mean, ninv;
   int i;
   ninv = 1.0/n;
   for( i=0; i<n; i++ )
      sum = sum + a[i];
   for( i=0; i<n; i++ )
      sumsq = sumsq + a[i]*a[i];
   mean = sum * ninv;
   *var = sumsq*ninv - mean*mean;
   return mean;
}
```

```
float meanvar(
   float *a,
   int n,
   float *var
){
   float sum = 0, sumsq = 0;
   float mean, ninv;
   int i;
   ninv = 1.0/n;
   for( i=0; i<n; i++ ){
      sum = sum + a[i]
      sumsq = sumsq + a[i]*a[i];
   }
   mean = sum * ninv;
   *var = sumsq*ninv - mean*mean;
   return mean;
}
```

Figure 16 Loop Fusion

Table 9 Effect of Loop Fusion on Calculation of Mean and Variance (Instructions/Element)

	Without Grafting	With Grafting
Separate Loops to Calculate Mean and Variance	11.62	11.82
Fusion of Two Loops	6.00	6.80

Replace || by |

C has a number of constructs, including the **&&**, **||**, and **?:** operators, that were designed for efficient execution on a sequential processor. Their operands cannot be evaluated in parallel. Use of these operators can increase the number of decision trees.

You can replace the expression (E_1 || E_2) by (E_1 | E_2) if two conditions are satisfied. It must be evaluated for its effects on control flow. E_1 and E_2 must have no side effects. If the expression is being evaluated for its value, it can be replaced by (E_1 | E_2) **!= 0**. Boolean or (||) operators add a decision tree to the program both when used in a control statement (**if**, **while**, **for**, or **do while**) and inside a **?:** expression.

Replace && by & or IZERO

The **IZERO** custom operation has a value **0** if its first operand is zero; otherwise, it has the value of its second operand. You can replace a boolean and operator (**&&**) by **IZERO** if the expression has no side effects and it is being evaluated for its effect on control flow.

Boolean and operators add a decision tree when used in a **?:** expression or a two-sided if statement but not inside an else less **if** statement or as the condition of a for or while.

If the value of the expression is needed, it can be replaced by **IZERO(**E_1**,** E_2**!= 0)** or **IZERO(**E_1**,** E_2**) != 0**, depending on which has the better critical path.

The program of Figure 17 counts the number of alphabetic characters and underscores in a string. If the operands of a **&&** operation are constrained to a boolean (**0/1**) value, you can use a bitwise "and" (**&**) operator. The operands in the figure are relationals with a boolean value. Replacing the **&&** and || operators by **&** and | reduces the execution time from 1347 to 556 cycles.

Table 10 Effect of Eliminating && and || Operators (Instruction Cycles)

	Without grafting	With grafting
Program of Figure 17 with && and || Operators	1003	783
Program with & and | Operators	635	486

```
#define alpha(c) (((c) >= 'a' && c<='z') || (c>='A' && c<='Z') || c=='_')
int alphacount(char *s){
   int c, count;
   count = 0;

   while( c = *s++ ) count = count + alpha(c);
   return count;
}
main(){
   alphacount("The quick brown fox jumps over the lazy dog");
}
```

Figure 17 Character Count

Using Software Pipelining

Typically, the three instructions after a jump operation are mostly unused. Data that is needed in the next iteration can be preloaded in these slots. This is called software pipelining. Figure 18 shows the C string comparison routine **strcmp**, before and after software pipelining. Eight cycles are needed for the loop for the program on the left with global optimization. Software pipelining allows the loop to execute in six cycles. You can also schedule long-latency operations (for example, floating point) in these slots.

A simple form of software pipelining is implemented by the global optimizer if grafting is not enabled.

```
strcmp(char *p, char *q){
   int c;

   while( (c=*p++) == *q++ && c ){}

  return c - q[-1];
}
```

```
strcmp(char *p, char *q){
   int c, c1, cont;
   c = *p++;
   c1 = *q++;
   cont = (c == c1) & (c != 0);
loop:
   if( cont ){
      c  = *p++;
      c1 = *q++;
      cont = (c == c1) & (c != 0);
      goto loop;
   }
   return c - c1;
}
```

Figure 18 Example of Software Pipelining

Use TriMedia Style Booleans in Critical Parts of the Code

In C, true and false are represented by nonzero and zero, respectively. TriMedia uses the low-order (guard) bit of a register to determine whether a condition is true or false. Even if the expression being tested is compared against zero, a comparison operator is required. Typically, five cycles are required for a conditional jump.

If the value of the expression is known to depend only on the low order bit, the comparison is not necessary. The TriMedia C compiler recognizes expressions of the form **(E$_1$ & 1)** or **!(E$_1$ & 1)**. Using these instead of **(E$_1$!= 0)** or **(E$_1$ == 0)** generates better code. Expressions such as (**E$_1$ & 2^n**) and **!(E$_1$ & 2^n)** are optimized also.

You can use TriMedia style booleans with **MUX** and **FMUX**. If the guard is known to depend on the first order bit, you can use the machine level pseudo operations mux and fmux instead. For example, if the variable v is constrained to a **0/1** value, you can replace **MUX(v != 0, E$_1$, E$_2$)** by **mux(v, E$_1$, E$_2$)**, saving a cycle.

Manual Loop Unrolling

You can perform loop unrolling manually. The loop in Figure 1 is shown unrolled in Figure 19, where the inner for loop is completely unrolled—that is, replaced with eight assignment statements. The outer for loop is unrolled four times. Replacing the convolution function with **unrolled_direct_convolution** gives us the new program fir2.c.

Note that loop unrolling is a specialized version of grafting. In loop unrolling, a conditional jump from a decision tree exit back to itself is replaced with the code for the decision tree. The main difference is that grafting replaces the jump part of the conditional jump with the destination decision tree but leaves the condition in place, which causes control dependence between one iteration of the loop to the next.

For example, the grafting shown in is essentially loop unrolling, and it can be seen that the grafted code is still governed by a condition. Without data-flow analysis, such condi-

tions cannot be removed and thus result in a lower performance, compared to manual unrolling.

```
void
unrolled_direct_convolution( char *a, char *b, int *c ){
   int   k, j;

   for(k = 0; k < NROF_SAMPLES; k += 4) {
      c[0] = b[0]*a[ 0] + b[1]*a[-1] + b[2]*a[-2] + b[3]*a[-3]
           + b[4]*a[-4] + b[5]*a[-5] + b[6]*a[-6] + b[7]*a[-7];
      c[1] = b[0]*a[ 1] + b[1]*a[ 0] + b[2]*a[-1] + b[3]*a[-2]
           + b[4]*a[-3] + b[5]*a[-4] + b[6]*a[-5] + b[7]*a[-6];
      c[2] = b[0]*a[ 2] + b[1]*a[ 1] + b[2]*a[ 0] + b[3]*a[-1]
           + b[4]*a[-2] + b[5]*a[-3] + b[6]*a[-4] + b[7]*a[-5];
      c[3] = b[0]*a[ 3] + b[1]*a[ 2] + b[2]*a[ 1] + b[3]*a[ 0]
           + b[4]*a[-1] + b[5]*a[-2] + b[6]*a[-3] + b[7]*a[-4];
      a += 4;
      c += 4;
   }
}
```

Figure 19 Convolution Example – Loop Unrolled (Example fir2.c)

The following commands compile fir2.c with profiling but without grafting.

```
tmcc -p fir2.c -o fir2     /* Generate program with profiling on */
tmsim -nomm fir2           /* Simulate interm code and produce dtprof.out. */
tmcc -r fir2.c -o fir2     /* Recompile using profile information. */
tmsim -statfile fir2.stat fir2   /* Simulate & collect cycle-accurate info */
tmprof -scale 1 -func fir2       /* stat   Output is sent to stdout. */
```

The performance of the unrolled loops is as shown in the following **tmprof** report below. The size of **text** section of fir2.o is 1223 bytes, which is somewhat more than the ungrafted, unrolled program fir1.c. The execution time of unrolled_direct_convolution is about 3.9 times faster than that of direct_convolution (in fir1.c) and about 3.7 times faster than the grafted version of direct_convolution. From this we can see that grafting is not the solution to all performance problems. It helps on large parts of the code that are not very critical but still interesting, but the most critical parts can better be optimized by hand.

Function	Executions	Total Cycles	MIPS	I-Cycles	D-Cycles
_unrolled_direct_convolut	1	7509	35.13	261	444
_main	1	102	0.48	87	0
total/average	226	21375	100.00	10315	1467

Manual Loop Unrolling Versus Grafting

You should apply grafting at the last stage of optimization because it is impossible to understand what is happening after grafting. You should apply manual loop unrolling only if it produces better performance then grafting. Consider, for example, the program in Figure 20, which initializes a symbol table with the list of C keywords. When compiled and run using the global optimizer (**-O3**), 1367 instruction cycles are necessary for

30 calls to **definesym** (45 cycles per call). 40 cycles per call are spent in the character copy loop. This corresponds to five cycles per copied byte.

```
#define NSYM 8
#define NFREE 100
struct symbol {
   struct symbol *next;
   char name[NSYM];
   int value;
} *avail, *symlist;

#define freesym(sym)  {struct symbol *t = (sym); t->next = avail; avail = t;}

char *keywords[] = {
"void", "char", "short", "int", "long", "float", "double", "struct",
"union", "enum", "unsigned", "auto", "extern", "static", "register",
"goto", "switch", "case", "default", "return", "if", "else", "while",
"do", "break", "continue", "for", "typedef", "sizeof" "const","volatile", 0};

struct symbol *definesym( char *str, int value ){
   int i;
   struct symbol *res = avail;
   avail = avail->next;
   for( i=0; i<NSYM; i++ ) res->name[i] = str[i];
   res->value = value;
   res->next = symlist;
   symlist = res;
   return res;
}
main() {
   int i;
   for( i=0; i<NFREE; i++ )
      freesym( (struct symbol*)malloc(sizeof(struct symbol)) );
   for( i=0; keywords[i]; i++ )
      definesym( keywords[i], i );
}
```

Figure 20 Symbol Table Initialization

The number of times the for loop in **definesym** is executed is known in advance. Grafting replicates the condition (**i<8**). For this reason, it is better to use loop unrolling.

Figure 21 shows code for the unrolled loop. Five cycles are necessary per byte using a loop. You can estimate the time corresponding to the unrolled loop as 3.25 cycles.

```
struct symbol *definesym(char *str, int value){
   struct symbol *res = avail;
   avail = avail->next;

   res->name[0] = str[0]; res->name[1] = str[1];
   res->name[2] = str[2]; res->name[3] = str[3];
   res->name[4] = str[4]; res->name[5] = str[5];
   res->name[6] = str[6]; res->name[7] = str[7];

   res->value = value;
   res->next  = symlist;
   symlist = res;
   return res;
}
```

Figure 21 Unrolled Loop

Using Restricted Pointers

C programs make heavy use of loads and stores through pointers. The C language does not allow the compiler to make any assumptions about pointers. Consider the following two assignment statements in the **for** loop of program fir2.c:

```
c[0] = b[0]*a[ 0] + b[1]*a[-1] + b[2]*a[-2] + b[3]*a[-3] + b[4]*a[-4] +
       b[5]*a[-5] + b[6]*a[-6] + b[7]*a[-7];
c[1] = b[0]*a[ 1] + b[1]*a[ 0] + b[2]*a[-1] + b[3]*a[-2] + b[4]*a[-3] +
       b[5]*a[-4] + b[6]*a[-5] + b[7]*a[-6];
```

Since a, b, and c are pointer parameters to the function **unrolled_direct_convolution**, in the absence of inter-procedural analysis the compiler assumes that they might refer to the same or overlapping memory locations; that is, be aliased to each other. In other words, the second assignment statement might be data dependent on the first statement. This implies that the operations of the two statements cannot be executed in parallel. However, you know that a, b, and c always are distinct arrays and thus never alias. You can convey this information to the compiler by declaring these pointers to be *restricted*.

Declaring pointers as restricted is a hint to the compiler that these pointers point to separate objects in memory that do not overlap with any known variable in the current context or with such an object related to any other restricted pointer. Based on this information, the compiler decides that different variables and/or restricted pointers do not alias. Note that it is your responsibility to verify that the assertion is true: proper use of restricted pointers reduces the amount of dependencies and, therefore, increases potential parallelism. However, declaring pointers to overlapping memory regions as

restricted results in an incorrect program. In our running example, we assign the type qualifier **restrict** to the declarations of a, b, and c, as shown in Figure 22.

```
restrict_direct_convolution(
   char * restrict a,
   char * restrict b,
   int  * restrict c
){
   int k, j;

   for(k = 0; k < NROF_SAMPLES; k += 4) {
      c[0] = b[0]*a[ 0] + b[1]*a[-1] + b[2]*a[-2] + b[3]*a[-3]
           + b[4]*a[-4] + b[5]*a[-5] + b[6]*a[-6] + b[7]*a[-7];
      c[1] = b[0]*a[ 1] + b[1]*a[ 0] + b[2]*a[-1] + b[3]*a[-2]
           + b[4]*a[-3] + b[5]*a[-4] + b[6]*a[-5] + b[7]*a[-6];
      c[2] = b[0]*a[ 2] + b[1]*a[ 1] + b[2]*a[ 0] + b[3]*a[-1]
           + b[4]*a[-2] + b[5]*a[-3] + b[6]*a[-4] + b[7]*a[-5];
      c[3] = b[0]*a[ 3] + b[1]*a[ 2] + b[2]*a[ 1] + b[3]*a[ 0]
           + b[4]*a[-1] + b[5]*a[-2] + b[6]*a[-3] + b[7]*a[-4];
      a += 4;
      c += 4;
   }
}
```

Figure 22 Convolution Example - Restricted Pointers (example fir3.c)

The following sequence of commands compiles, profiles, and recompiles the program, and then produces a report.

```
tmcc -p fir3.c -o fir3      /* Generate program with profiling on.*/
tmsim -nomm fir3            /* Simulate interm code and produce dtprof.out.*/
tmcc -r fir3.c -o fir3      /* Recompile using profile information. */
tmsim -statfile fir3.stat fir3   /* Simulate & collect cycle-accurate info */
tmprof -scale 1 -func fir3.stat  /* Output is sent to stdout.*/
```

The output of **tmprof** is as follows:

```
Function                Executions  Total Cycles   MIPS   I-Cycles   D-Cycles
_____________________   __________  ____________   ____   ________   ________
_restrict_direct_convolut        1          2711  16.35        145         60
_main                            1           102   0.62         87          0
_____________________________________________________________________________
total/average                  226         16577 100.00      10199       1083
```

Notice that the execution speed of the unrolled loop improves by about 64% when restricted pointers are used, compared to program fir2.c. This latest version of the loop executes about 10.8 times faster than the original version in program fir1.c.

As one more improvement, we use grafting along with unrolling and restricted pointers. The following commands are used for compiling program fir3.c.

```
tmcc -p fir3.c -o fir3  /* Generate program with profiling on.*/
tmsim -nomm fir3        /* Simulate interm code & produce dtprof.out.*/
tmcc -G fir3.c -o fir3  /* Recompile using profile infor, perform grafting.*/
tmsim -statfile fir3.stat fir3   /* Simulate & collect cycle-accurate info */
tmprof -scale 1 -func fir3.stat  /* Output is sent to stdout.*/
```


Grafting again gains performance, mainly due to the effect on other functions. The text section of the object code fir3.o has a size of 2736 bytes, which is about 3.8 times the original code size.

Function	Executions	Total Cycles	MIPS	I-Cycles	D-Cycles
_restrict_direct_convolut	1	2574	15.41	724	60
_main	1	102	0.61	87	0
total/average	226	16708	100.00	10981	1111

The pointers **str**, **avail**, and **symlist** in Figure 21 could have identical values or differing values. They could even overlap. Because of this, the compiler must order stores through pointers with respect to other loads and it stores in strict program order. There are 11 loads and 13 stores in the procedure **definesym**. Issuing these operations in strict program order limits the ILP.

The **str** points to an array of characters and **avail** points to a symbol table entry. It seems clear they differ given the types. **avail** points to a list of available nodes and **symlist** points to a list of nodes on the symbol table. These two sets should be independent. All but two (the uses of **symlist**) of the 24 memory references can be shown to differ. The other accesses can be performed in parallel.

Figure 23 shows how to modify the program to use restricted pointers. 516 instruction cycles are needed for **definesym,** as compared to 1367 before and 1096 after unrolling. This corresponds to 12 instructions per call to the function. Table 11 summarizes the performances with restricted pointers and loop unrolling and before optimization.

```
struct symbol *definesym( char * restrict str, int value ){
   struct symbol * restrict res = avail;
   avail = avail->next;
   res->name[0] = str[0]; res->name[1] = str[1];
   res->name[2] = str[2]; res->name[3] = str[3];
   res->name[4] = str[4]; res->name[5] = str[5];
   res->name[6] = str[6]; res->name[7] = str[7];
   res->value = value;
   res->next = symlist;
   symlist = res;
   return res;
}
```

Figure 23 Using Restricted Pointers

Table 11 Instruction Cycles Per Procedure Call

	Call To definesym
No Optimization	45.6
Loop Unrolling	36.5
Loop Unrolling + Restricted Pointers	17.2

Be aware that unwarranted use of restricted pointers can introduce subtle bugs.

The implementation of the **restrict** keyword is based on the paper *Restricted Pointers in C* by the Numerical C Extensions Group of ANSI X3J11. This paper (X3J11/94-019, Draft 2) can be found at http://www.lysator.liu.se/c/restrict.html. To the best of our knowledge, restricted pointers will be in the future ANSI C standard.

Using Custom Operators

The TriMedia hardware architecture provides special operations for DSP applications. They are made available through the **custom_op** declaration. In fact, all machine operations are available through the **custom_op** mechanism, but not all of them are of use to you. The most important ones are defined in the include file custom_defs.h[1]. We recommend that you use only the custom operators defined in custom_defs.h. By using only these, you can develop and execute on the host platform because a special library with the implementation of the custom operators is provided.

```
void custom_ops_direct_convolution(
   char * restrict a, char * restrict b, int  * restrict c
){
   int i, ib0, ib1, i0, i1, i2;
   int * restrict ia;

   ia = (int *) a;

/* Copy b, in a new array called rev_b in time reversed order. ib points to
 * this array as an integer pointer. Let A = |abcd| and B = |pqrs| where
 * a,b,c,d,p,q,r, and s are all 8 bit integers. Then PACKBYTES(A,B) = |ds|
 * and PACK16LSB(A,B) = |cdrs| */
   ib0 = PACK16LSB(PACKBYTES(b[7], b[6]), PACKBYTES(b[5], b[4]));
   ib1 = PACK16LSB(PACKBYTES(b[3], b[2]), PACKBYTES(b[1], b[0]));

   for( i = 0; i < NROF_SAMPLES/4; i++ ){
   /* Let A = |abcd| and B = |pqrs| where a,b,c,d,p,q,r, and s are all 8-bit
    * integers. Then
    * FUNSHIFT1(A,B) = |bcdp|
    * FUNSHIFT2(A,B) = |cdpq|
    * FUNSHIFT3(A,B) = |dpqr|
    * IFIR8II(A,B)   = a*p + b*q + c*r + d*s */

      i0 = ia[i - 2];
      i1 = ia[i - 1];
      i2 = ia[i    ];

      c[0] = IFIR8II(ib0,FUNSHIFT1(i0,i1)) + IFIR8II(ib1,FUNSHIFT1(i1,i2));
      .....
      c[3] = IFIR8II(ib0, i1) + IFIR8II(ib1, i2);
      c += 4;
   }
}
```

Figure 24 Convolution Example - Custom Operators

1. The real declaration of the custom operators is done in include file custom_ops.h. The file custom_defs.h is an abstraction from the custom operators to enable you to develop and execute on the host platform with use of the TriMedia custom_ops.

Of special interest for the example are the custom_ops **FUNSHIFT** and **IFIR8II.** Let A = |abcd| and B = |pqrs| where a, b, c, d, p, q, r, s, are all 8-bit integers. Then,

```
FUNSHIFT1(A,B) = |bcdp|
FUNSHIFT2(A,B) = |cdpq|
FUNSHIFT3(A,B) = |dpqr|
IFIR8II(A,B) = a*p + b*q + c*r + d*s
```

Four multiplications and three additions in the inner loop of program fir3.c are replaced by one **FUNSHIFT** and one **IFIR8II** operation. Other usage of custom operators in the fir program are selecting bytes or half words and merging them into one word (PACKBYTES and PACK16LSB). To use these custom operators, the program must include the header file custom_defs.h. You can find this include file in the directory $TCS/include/ops. This directory is in the default include path for the compiler driver. Most custom operators directly map to hardware operations. Access via the include file custom_defs.h ensures that your program can still run on the host system, because a library of custom operator implementation for the host system is provided.

Figure 24 shows the modified function, still using restricted pointers. Compiling the program fir4.c while grafting using the following commands and running **tmprof** shows the performance gain due to custom operators:

```
tmcc -p fir4.c -o fir4   /* Generate program with profiling on. */
tmsim -nomm fir4         /* Simulate interm code and produce dtprof.out. */
tmcc -G fir4.c -o fir4   /* Recompile using profile info, perform grafting */
tmsim -statfile fir4.stat fir4   /* Simulate & collect cycle-accurate info.*/
tmprof -scale 1 -func fir4.stat  /* Output is sent to stdout.*/
```

The output of **tmprof** shows

Function	Executions	Total Cycles	MIPS	I-Cycles	D-Cycles
_custom_ops_direct_convol	1	1340	8.68	673	82
_main	1	102	0.66	87	0
total/average	226	15445	100.00	10901	1133

The unrolled version of the loop now is already 21.9 times as fast as the original version, because of the use of custom operators, grafting, and restricted pointers.

Graft-Tuning Parameters

You can guide the grafting decision of the compiler through a grafting parameters file. The graft-tuning file allows specifying a number of conditions on decision tree grafting on a function by function basis. You can specify graft parameters for some functions and use a default applicable to all functions that were not explicitly listed in the graft parameters file. The parameters that govern grafting decisions are as follows.

- *Graft Enable*. This boolean flag enables or disables grafting for a particular function.
- *Maximum Code Replication Factor*. This limits the factor by which the code size for a function can be expanded due to grafting. Although grafting might initially increase code size, many optimizations are performed after grafting that reduce code size.

- *Maximum Graft Depth*. This limits how many times grafting is performed along a particular execution path in the current decision tree, restricting how much grafting is allowed on a tree.
- *Minimum Probability Threshold*. This specifies the minimum probability of execution of a branch to allow grafting for that branch.
- *Minimum Execution Count Threshold*. A decision tree is not a candidate for grafting if its execution count is below this threshold.

A graft-tuning file can contain different parameters for different functions and at most one default set of parameters for all functions that are not listed explicitly in the file. The default values for the graft tuning parameters are as follows:

Function	Enabled	Codesize	Depth	Prob. Threshold	Exec. Count Threshold
<default>	1	20.0	20	0.4	10.0

The **tmprof** output shows that the stall cycles from the cache misses increased during the optimization stages. First loop unrolling was done, followed by grafting. These are similar techniques, and possibly the code expansion of the two was too much for the default grafting parameters. When limiting the grafting for the already unrolled function, a better cache characteristic can be obtained. The **graftfile** with the following parameters is used on the example:

Function	Enabled	Codesize	Depth	Prob. Threshold	Exec. Count Threshold
<default>	1	4.0	2	0.4	10.0

The performance improvement by tuning the graft file is mainly due to the reduction in the instruction cache stall cycles.

Tuning the graftfile should be applied only to applications of a significant size. The performance improvement is due to a reduction in code size. This produces a reduction in instruction cache cycles.

Using Profiling and Grafting

As explained in "Grafting Based on Profile Information" starting on 12, the TriMedia compiler can generate more parallel code by being trained about the behavior of the program. First, it is necessary to compile the program with the **-p** (**profile** option):

```
$ tmcc -p insertion_sort.c
```

This tells the compiler to add code to generate statistics to the program. Running it using **tmsim** produces a file dtprof.out containing more decision tree information. It can be read using **tmdtprof**:

```
$ tmdtprof dtprof.out
insertion_sort.c:main()        calls = 1      operations = 1241
insertion_sort.c:insertion()   calls = 1      operations = 76345
Function count = 2
```



```
path: insertion_sort.c main() 16/1 (dtree count = 5)
            (...)
path: insertion_sort.c insertion() 4/6 (dtree count = 7)
   dt(0)4/1 ops(11) exits(2)
      0 -> dt(1) exec count(1)
      1 -> dt(1) exec count(0)
         (...)
   dt(3)11/17 ops(15) exits(3)
      0 -> dt(3) exec count(4851)
      1 -> dt(4) exec count(0)
      2 -> dt(4) exec count(99)
...
```

The underlined information gives the correspondence between a decision tree and the file and line in the source program. For example, the procedure **insertion** starts at column six of line four of "insertion_sort.c." The **insertion$_3$** function begins at line nine, column six. The **–>** lines are the exit paths (jumps) out of the decision tree. For example, on path zero, **insertion$_3$** loops to itself 4,851 times. On the second path, it goes 99 times to **insertion$_4$**, and path one is never taken.

You can compile with the **-G** option which tells the compiler to use grafting. The program can be compiled and run with grafting as follows:

```
$ tmcc -G -O3 -O ins.O3.graft insertion_sort.c
$ tmsim -v -statfile ins.O3.graft.stat ins.O3.graft
   ...
nr of cycles:                 20866
nr of executed instructions:  19142
CPI:                          1.090

instruction cache statistics: size 32 kB, blocksize 64 b, associativity 8
nr of accesses:  19143
nr of hits:      19091
hitrate:         100 %
CPI:             0.083
data cache statistics: size 16 kB, blocksize 64 b, associativity 8
nr of accesses:  10388
nr of hits:      10373
   ...
$ select insertion_DT_2 ins.O3.graft.stat
tree name          execs  instc istallc dstall  cpback cnflct  isoper  exoper
-------------------------------------------------------------------------------
__insertion_DT_2    1128  16638     87      0       0      0     56068   55718
__clearregs_DT_1      32    192     58     16       0      6       512     511
__clearregs_DT_1      32    928     58     24       0      1       832     831
```

Table 12 compares the inner loop behavior at **-O3** with and without grafting. 78% of the branches (1128 versus 4950) are eliminated due to grafting (the loop is grafted four

times). The ILP increases from 1.60 to 3.36. The issued and executed operations correspond to the **isopers** and **exopers** fields of the statfile (see above).

Table 12 Inner Loop Behavior at -O3 with and without Grafting

	Executions of Decision Tree 2	Instruction Cycles	Issued Operations	Executed Operations	Ops/Instr (ILP)
No Grafting	4950	49104	79002	79002	1.60
Grafting	1128	16638	56068	55718	3.36

Table 13 compares the overall behavior. The number of memory accesses is about the same with and without global optimization. However, there is a reduction of one third when using both global optimization and grafting.

Table 13 Overall Behavior At Different Levels of Optimization

Optimizations	Instructions	Memory Accesses	Total Cycles
Local Optimization	71237	15180	72534
Global Optimization	51332	15279	52573
Grafting and Global Optimization	19142	10373	20855

Using Unsafe Alias Analysis

The earlier section *Using Restricted Pointers* on page 33 describes how you can use restricted pointers to help the compiler in alias analysis[1]. Good alias analysis is one of the key techniques for obtaining parallelism and the best optimization. The alias analyzer weakens the ordering of memory operations (all assignments and uses of values in C terms). A weaker ordering allows more operations to go in parallel. In C, it is important not to use pointers unless it is necessary because an unknown pointer aliases with all nonlocal nonexposed variables[2]. Also, the use of global variables limits the abilities of the alias analyzer to disambiguate two memory locations.

The compiler currently has three levels of alias analysis. Level zero is perfectly safe, that is, no assumptions are made on any use of the ANSI C language. The two higher levels do make assumptions on the use of the language and are safe in most programs. *However, when using unsafe alias analysis, it is very important to understand the details of the program and the use of all memory references*.

You can specify unsafe alias analysis with the option **-A[012]** to the compiler. The default level is 1. You can use the pragmas to change the alias level function per function.

1. Alias analysis is the technique used in the compiler to determine whether two memory locations are the same or whether they overlap.
2. Local variables are variables declared within a function scope. Nonexposed variables are variables of which the address is never taken. The compiler knows that if the address is never taken that it cannot be stored to any pointer variable and, thus, does not alias with any pointer indirection. In the absence of inter-procedural analysis, the nonexposed property can only be determined for local variables.

Level 1 makes the following two assumptions:

1. The memory object referred to by a certain pointer does not contain the same pointer. This means that when **p** points to a struct, it is assumed that there is no field **f** in the struct such that **p–>f==p**. Also, when **p** points to an array of pointers, it is assumed that there is no index **i** such that **p[i]==p**. Similar assumptions are made for arbitrary nesting of arrays and structs within the memory objects that **p** refers to. See Figure 25, for example.
2. It is assumed that it is senseless to address outside any variable, and that it is impossible to 'reach' a variable through a pointer that does not already 'point' somewhere in the same variable. This means that no assumptions are made on the relative positions at which the variables are mapped in memory, and that no attempt is made during execution to determine these relative positions.

These assumptions are used by the alias analyzer when trying to determine possible aliasing in case of a memory reference through a pointer (with a certain offset) and to a variable: if, given that the pointer points 'somewhere' in the variable, the memory reference via the pointer and with the given offset would result in addressing (partially) outside that variable, no aliasing is assumed. In mathematical terms: if **p** is a pointer and **a** is a variable the memory region [***p + sizeof(a), *p + sizeof(*p)**] does not alias with **a**. See Figure 26.

Level 2 assumes everything from level 1. Globals are modified only by accessing the global itself; that is, the address of a global is never taken. See Figure 27 for an example.

Note
Level two is only implemented with global optimizations on, that is, optimization level three (**-O3**).

```
int *p, *q;
program fragment:
   *p = 3;
    q = p;
/* At unsafe alias analysis level -A1, we assume that *p != p, so the value
   of p does not have to be reloaded after the assignment *p = 3.*/
```

Figure 25 Example of Point One for Unsafe Alias Analysis Level One

```
typedef struct some_type{
   char    c;
   short   s;
   int     i;
};
int a;
struct some_type *p;
```

Figure 26 Example of Point Two for Unsafe Alias Level One

At any program point, **p–>c** and **p–>s** will alias with **a** (for instance when we initialize **p = (struct some_type)&a**, but **p–>i** will not alias with any load or store to **a** at unsafe alias **-A1**. This is because **a** can not be packed into a larger object (like ***p**). However, due to the

casting possibility in C, the compiler still lets both objects alias at the start (or at the address) of the object.

Note that the assumption that **p->i** and **a** do not alias is not valid when we initialize **p = (some_type *)(&a–1**). But then a store to ***b** would arbitrarily overwrite something in memory. Program constructs like this do occur, but rarely.

```
int *p;
int a;
```

Figure 27 Example of Point Three for Unsafe Alias Level Two

a and ***p** will not alias at unsafe alias level **-A2** (in **-O3**) because the compiler assumes that the address of a is never taken and can thus never have been assigned to **p**.

Figure 28 shows a program that initializes four arrays. Decision tree statistics for the loop at **-O3** are shown.

tree name	execs	instc	istallc	dstallc	cpbacks	cnflctc	isopers	exopers
__clearregs_DT_2	32	896	58	12	0	0	640	606

```
#define NREG 32
typedef struct rtl *rtx;
int *qty_first_reg, *qty_last_reg,   max_qty = NREG;
rtx *qty_const,     *qty_const_insn;

main(){
   qty_first_reg  = (int *)malloc(NREG * sizeof(int));
   qty_last_reg   = (int *)malloc(NREG * sizeof(int));
   qty_const      = (rtx *)malloc(NREG * sizeof(rtx));
   qty_const_insn = (rtx *)malloc(NREG * sizeof(rtx));
   clearregs();
}
clearregs(){
   int i;
   for( i = 0; i < max_qty; i++ ){
      qty_first_reg  [i] = i;
      qty_last_reg   [i] = i;
      qty_const      [i] = 0;
      qty_const_insn [i] = 0;
   }
}
```

Figure 28 Initializing Four Arrays

The loop is executed 32 times and there are four stores per iteration. This corresponds to 128 memory accesses. The compiler does not know whether the addresses of the external variables (**max_qty**, **qty_first_reg**, **qty_last_reg**, **qty_const**, **qty_const_insn**) have been assigned to a pointer. For example, the loop is executed 32 times. However, **qty_first_reg** could point to **max_qty** at the start of the loop. If this is true, the loop should only be executed once. Five load accesses are necessary per loop iteration because of the pointer aliasing. 160 memory accesses are added to the program.

The **-A2** option of **tmcc** relaxes the rules for alias analysis. The compiler can assume that the accesses to **extern** and **static** variables do not alias with stores through pointers. The option **-O3** must be specified if the **-A2** option is to have effect. The decision tree statistics with **-A2** are shown below:

tree name	execs	instc	istallc	dstallc	cpbacks	cnflctc	isopers	exopers
__clearregs_DT_2	32	193	29	19	0	0	480	446

Table 14 compares the performance with and without relaxed aliasing. Six instruction cycles (192/32) are necessary per loop iteration with the **-A2** option. 28 instruction cycles (896/32) are necessary without the **-A2** option. There are five variables and a load of each requires two operations. This corresponds to the ten (5×2) fewer operations in the loop using relaxed aliasing. Only 279 cycles, as opposed to 1004 including cache overhead, are necessary with relaxed aliasing.

Table 14 Performance With And Without Relaxed Aliasing

	Instruction Cycles per Loop	Operations	Memory Accesses	ILP	Total Cycles
Without Relaxed Aliasing (no **-A2** option)	28	20	288	0.73	1004
With Relaxed Aliasing (**-A2** option)	6	14	128	2.29	279

Take care when using relaxed aliasing. Do not use it if a global variable's address is taken. Figure 29 shows how to use relaxed alias analysis for an individual procedure or function.

```
clearregs(){
   int i;
#pragma TCS_A2
   for( i = 0; i < max_qty; i++ ){
      qty_first_reg  [i] = i;
      qty_last_reg   [i] = i;
      qty_const      [i] = 0;
      qty_const_insn [i] = 0;
   }
}
```

Figure 29 Locally Relaxed Aliasing

Using a Dirty Float

Usually compiler optimizations on floating point expressions are illegal. This is because all commutative and associative properties that hold for integer operations like addition and multiply do not hold for floating point operations. You can give the compiler more freedom in expression optimizations and program transformations by using the **dirty_float** option.

You can give the option on the command line with **dirty_float <nn>** or with pragma's **TCS_dirty_float0**, ..., **TCS_dirty_float2**. There are three levels with the following meaning:

- At level zero there are no optimizations performed on floating point expressions.
- At level one the compiler folds constant floating point expressions and introduces conversions for **if** statements containing floating point expressions. Expressions remain ordered against read and write to PCSW
- At level two, besides the operations performed at level one, the compiler performs tree height reduction and reordering of floating point expressions to increase parallelism. Also, otherwise illegal optimizations like rewriting the expression **d != d** (check for **NaN**) to false are performed.

Note
This option only has an effect at optimization level three, and *might cause incorrect results.*

Using Cache Optimization

Several of the techniques discussed in the preceding sections, including the use of grafting, loop unrolling, and inlining, result in an increase in the size of the program code, which in turn, increases the number of instruction cache stalls. You must pay attention to the code size because the I-cache stalls can become an important factor. This section addresses techniques to enhance data cache utilization, thereby improving the overall program performance.

Vary the Right-Most Array Index in the Inner Loop

The program on the left of Figure 30 zeroes a byte array. **tmprof** output from running it is shown below:

Treename	Executions	Total Cycles	MIPS	I-Cycles	D-Cycles
__main_DT_2	1280	669877	96.01	58	658169
__memset_DT_1	621	6053	0.87	29	3540
___cache_copyback_DT_1	103	4698	0.67	48	4032
__main_DT_1	64	3833	0.55	29	3548
total/average	2453	697725	100.00	10041	670399

Most of the execution time (670399 out of 697725 cycles) is lost in data cache stalls. Almost all the stall cycles are in the decision tree **__main_DT_2**. This corresponds to the inner **for** loop. It is executed 1280 (64×20) times. The inner loop is unrolled 15 times by the compiler.

The stalls in the program of Figure 30 are data cache write miss stalls. Figure 31 shows code that you can use to instrument the program. The list of events is in the TriMedia

data book. You must add two lines before the first **for** to generate and count data cache write missed events:

```
Int timer;
UInt32 events;
monitor(&timer);
```

You must also add instrumentation code after execution of the relevant section of the program:

```
timGetTimerValue( timer, &events);
printf("cache misses = %d\n", events());
```

The instrumentation reports 19199 data cache write misses. There is a cache miss for every access. In C, the rightmost subscript of a multidimensional array varies fastest as elements are accessed in storage order. Each execution of the inner loop clears a single byte. Consecutive accesses by the program of Figure 30 are spaced by 64 bytes. This is the size of a cache line. Each cache miss corresponds to 64 bytes, only one of which is used.

The program on the right is equivalent to the program on the left, with the order of the for loops interchanged. 132702 cycles are necessary to execute the program on the right. Only 5271 cycles are lost in data cache stalls. Instrumentation allows the number of data cache stalls in the program on the right to be measured also. Measurement shows 300 data cache stalls. 300 data cache lines correspond to 19200 (300×64) bytes of data. This corresponds to all the data in the array. Each miss costs 17.57 (5271/300) cycles, compared to 34.92 cycles previously. This is because the misses can be overlapped with the execution of the program. The program on the left of Figure 30 generates too many stalls, so no overlap is possible after the first miss. Table 15 summarizes the effects of interchanging the loops.

```
#include <stdio.h>
#include <tm1/mmio.h>

char a[300][64];
main(){
   int k, l;
   for( l=0; l<64; l++ )
      for( k=0; k<300; k++ )
         a[k][l] = 0;
}
```

```
#include <stdio.h>
#include <tm1/mmio.h>

char a[300][64];
main(){
   int k, l;
   for( k=0; k<300; k++ )
      for( l=0; l<64; l++ )
         a[k][l] = 0;
}
```

Figure 30 Loop Interchange

```
#include <tm1/mmio.h>
#include <tm1/tmTimers.h>

void monitor( Int *timer) {
   timInstanceSetup_t   setup;
   MMIO(MEM_EVENTS)= 5;
   setup.source   = timCACHE1;
   setup.prescale = 1;
   setup.modulus  = 0;
   setup.handler  = Null;
   setup.priority = intPRIO_4;
```

```
    setup.running  = True;
    timOpen( timer);
    timInstanceSetup(*timer, &setup );
    timSetTimerValue(*timer,0);
}
```

Figure 31 Instrumentation Code

Table 15 Data Cache Write Misses Clearing a 64 x 300 Array of Characters

	Write Misses	Write Miss Stall Cycles
First Index Varies in Inner Loop	19199	658169
Second Index Varies in Inner Loop	300	4142

Pack Data as Tightly as Possible

Figure 32 shows a procedure to look up the name of the city closest to a point. A city is represented by a data structure containing the **x** and **y** coordinates (2×4 bytes) and the name of the city (64 bytes).

```
#define distance(x1, y1, x2, y2)  ((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1))
#define NCITY  256
typedef struct city { int x; int y; char name[64]; } CITY;
CITY cities[NCITY];

char * closest( int x, int y ){
   int max, dist, here = 0, i; CITY *ap;

   max = distance(x, y, cities->x, cities->y);
   for (i = 1, ap = &cities[1]; ap<&cities[NCITY]; ap++, i++) {
      dist = distance(x, y, ap->x, ap->y);
      if (max > dist) { max = dist;  here = i; }
   }
   return cities[here].name;
}
```

Figure 32 Linear Search

The distance needs to be computed from each city. The key fields (**x**, **y**) of the data structure are referenced 256 times for each call. The name of the city is consecutive with those in memory. The cache brings both into memory. However, the name is referenced only once at the end of the procedure. Each access during the search accesses 64 bytes, only eight (2×4) of which are used. 4082 cycles are necessary in the procedure, of which 1179 correspond to data cache miss stalls.

In Figure 33, the data structure has been modified so that the fields not accessed during the search are stored apart. The key fields (**x,y**) have also been packed into shorts (2×2

bytes). The 3350 cycles are necessary after data restructuring. The 448 cycles are data cache stalls. Table 16 summarizes the effect.

Table 16 Data Cache Performance for a 256-Element Linear Search

	Data Cache Stall Cycles
Key and Value Stored Together (8 Bytes of Key)	1179
Key and Value Stored Separately (4 Bytes of Key)	448

```
struct city { short x; short y;  } CITY;
CITY cities[NCITY];
char city_names[NCITY][64];
```

Figure 33 Modified Data Structure

Trade CPU Cycles for Cache Cycles

Figure 34 shows two programs to calculate the sieve of Eratosthenes. The program on the left represents the sieve by an array of bytes. The program on the right represents the sieve by a bit vector. Using a bit vector saves space but requires more operations to set and test an element.

Table 17 compares performance for calculating the 6542 primes between one and 65536. The figures shown correspond to the number of instruction cycles and cache stall cycles for the inner loop. Even though the program on the right is more complex, the number of instructions is identical. This is because there is spare processing power available. The store to sieve in the program on the right is more complex, but it can be executed partially in parallel. At the end of most decision trees there are available slots. Part of the store also executes in these slots.

Only 8K bytes are necessary to represent 65536 primes using a bit vector. Represented this way, the sieve fits in the cache. Represented as an array of bytes, it does not fit in the cache. This explains the difference in performance. Five opportunities are lost to issue operations for every stall cycle. It is worth increasing the number of instructions if the working set fits in the cache as a result.

Table 17 Sieve of Eratosthenes (Primes from one to 65536)

	Instruction Cycles	Data Cache Cycles
Sieve Represented as a Byte Vector	884931	1966538
Sieve Represented as a Bit Vector	1330858	1792

```
#define MAXPRIME 1000000
char sieve[MAXPRIME+1];
main(int argc, char *argv[]){
  int i, j, maxprime;

  maxprime = atoi(argv[1]);
  for(i=2; i<=maxprime; i++)
    sieve[i] = 1;
  sieve[0] = sieve[1] = 0;
  for(i=2; i <= maxprime>>1; i++){
    if (sieve[i]) {
      for(j=2*i;j<=maxprime;j+=i){
        sieve[j] = 0;
      }
    }
  }
}
```

```
#define MAXPRIME 1000000
char sieve[(MAXPRIME+7)/8];
main(int argc, char *argv[]){
  int i, j, maxprime;

  maxprime = atoi(argv[1]);
  for(i=0; i<=(maxprime+7)/8; i++)
    sieve[i] = -1;
/* 0 and 1 aren't prime */
  sieve[0] &= ~3;
  for(i=2; i <= maxprime>>1; i++){
    if((sieve[i>>3] >> (i&7)) & 1){
      for(j=2*i; j<=maxprime; j+=i){
        sieve[j>>3] &= ~(1<<(j&7));
      }
    }
  }
}
```

Figure 34 Sieve of Eratosthenes

Watch for Cache Set Hotspots

Figure 35 shows a procedure that sums up a column of an $n \times 128$-element matrix. Performance figures for different values of n are given in Table 18. They correspond to 16 consecutive columns.

```
int colsum(int col, int step) {
   int i, sum = 0;
   int *pcol;

   pcol = &matrix[0][col];
   for (i=0; i<128; i++) {
      sum  += *pcol;
      pcol +=  step;
   }
   return sum;
}
```

Figure 35 Column of n×128-Element Matrix

Table 18 Performance Figure for Values of N

Matrix Dimensions	Stride	Data Cache Stall Cycles	% Total Cycles
128 x 64	256	2051	89
128 x 65	260	1309	83
128 x 80	320	1441	85
128 x 512	2048	1975	89
128 x 513	2052	1988	89
128 x 1040	4160	1897	88

The percentage of data cache stall cycles varies depending on the row dimension in a ratio from one to fifteen. Accesses to an array in row order, address consecutive bytes. In column order the accesses are separated by a *stride* equal to the size of the element multiplied by the row length. For example, for a 256×1040 array of integers, column accesses are separated by 4160 bytes.

There are 256 lines in the cache of 64 bytes each. These are organized into 32 sets capable of holding eight elements each. The set number is given by address bits six through eleven. The byte offset inside a line is given by bits zero to five.

The accesses in Table 18 are separated by a stride of more than 64 bytes. Each references a different line. The contents are reused only after an entire column has been traversed. Satisfactory performance for this program requires that 128 lines be held in the cache.

For a 256×64 matrix, imagine that the first access hits a particular set (say 29). The stride is an exact multiple of the line size (256 = 4×64). The next access hits the set number + 4 modulo 32 (say 1). After referencing eight elements the accesses wrap around to set 29. The 128 accesses only use 64 (8×8) of the 256 lines of the cache. The working set of the program is 128 lines. This explains the poor performance. The performance is the same regardless of the starting set.

For a 256 x 1040 matrix, imagine that the first access also hits set 29. The stride is also an exact multiple of the line size. (4160 = 65 x 64). The next access hits set 30 (94 modulo 32). The next access hits set 31. The 128 accesses can fully use the cache. Again, the performance does not depend on the number of the first set.

For a 256 x 65 matrix, the stride (260) is not an exact multiple of 64 bytes. Accesses are made to set numbers separated by four (260/64). However, every 16 accesses (64 / (260 mod 64)), the set number is also incremented. This allows the 128 accesses to be distributed among all the sets.

For a 256 x 513 matrix, every 16 accesses the set number is also incremented by one (260 mod 64 = 2052 mod 64). However, accesses are made to set numbers separated by 32 (2052/64), so the 16 accesses all hit the same set. The 128 accesses are distributed among only eight of the 256 lines of the cache.

```
float a[96][96], b[96][96], c[96][96];
main(){
   int i, j;

   for (i=0; i<96; i++)
     for (j=0; j<96; j++)
        c[i,j] = a[i,0]*b[0,j] + a[i,1]*b[1,j] + a[i,2]*b[2,j] +  ...   + a[i,94]*b[94,j]+ a[i,95]*b[95,j];
}
```

Figure 36 Dot Product Matrix Multiply

Blocking

Figure 36 and Figure 37 show different algorithms to multiply two 96×96 square matrices. $a_{i,j}$ is used as shorthand for **a[i][j]** in the figures. The algorithm of Figure 36 uses an

unrolled dot product. This gives a high degree of parallelism. However, the blocked algorithm has better register and cache reuse.

A 96-element row of the array **a** is brought into memory to be multiplied with a column of the array **b**. The values cannot be reused until the entire dot product has been calculated. The blocked algorithm works using 6×6 pieces of the two matrices. 72 (2×6×6) values need to be brought in from memory.

There are 216 (6^3) product terms with a total of 432 (2×216) operands. Each input value can be reused six (432/72) times. Blocking allows 5/6 of the load instructions to be eliminated.

This also gives better cache locality. This is because the dot product reads 96 elements of **b** in column order. The effect of this is limited here because the elements can fit in the cache.

```
void
block(float(*restrict a)[96],float(*restrict b)[96],(float(*restrict c)[95]){
   float (*restrict d)[96];

   d = c;
   c0,0 = c0,0+a0,0*b0,0+a0,1*b1,0+a0,2*b2,0+a0,3*b3,0+a0,4*b4,0+a0,5*b5,0;
   d0,1 = d0,1+a0,0*b0,1+a0,1*b1,1+a0,2*b2,1+a0,3*b3,1+a0,4*b4,1+a0,5*b5,1;
          ...
   d0,5 = d0,5+a0,0*b0,5+a0,1*b1,5+a0,2*b2,5+a0,3*b3,5+a0,4*b4,5+a0,5*b5,5;
   c1,0 = c1,0+a1,0*b0,0+a1,1*b1,0+a1,2*b2,0+a1,3*b3,0+a1,4*b4,0+a1,5*b5,0;
          ...
   d5,5 = d5,5+a5,0*b0,5+a5,1*b1,5+a5,2*b2,5+a5,3*b3,5+a5,4*b4,5+a5,5*b5,5;
}
float a[96][96], b[96][96], c[96][96];
main(){
   int i, j, k;
   memset( c, 0, sizeof(c) );
   for( i=0; i<96; i+=6 )
      for( j=0; j<96; j+=6 )
         for( k=0; k<96; k+=6 )
            block( &a[i][k], &b[k][j], &c[i][j] );
}
```

Figure 37 Matrix Multiply with Blocking

Table 19 compares the performance of the blocking and the dot product algorithms.

Table 19 Performance of Blocking and Dot Product Algorithms

	Instruction Cycles	Memory Accesses	Misses (Data Cache)	Miss Cycles (Data Cache)	ILP (Inner Loop)
Dot Product	966710	1778714	56549	10024263	4.74
Blocking	613710	589878	19053	306674	4.63

Two-Level Blocking

The blocking algorithm of Figure 37 brings in 144 bytes (6×6×4) of the **a** and **b** arrays into memory. Blocks of the **b** array are read in column order. By adding a second level of

blocking, you can improve the cache locality. Figure 38 gives the algorithm. You can use the **block** procedure of Figure 37 (just change the dimension).

Three levels of loops, corresponding to the iteration variables (**i**, **j**, **k**) have been added inside the loop. The innermost (**k**) loop has been unrolled to reduce overhead. The three outer loops process 30×30 square blocks. The inner loops process 6×6 subsquares. The order (**i, j**, **k**) of the inner loops should be the same as the outer loops so that the **c** result is accumulated in the same order. Floating point addition is not associative. Comparative performance for single-and two-level blocking is given in Table 20. Although the extra loop levels increase the number of instructions, the overall performance is nearly doubled.

Table 20 Performance for Single and Two-Level Blocking

	Instruction Cycles	Memory Accesses	Misses (Data Cache)	Miss Cycles (Data Cache)	ILP	CPI
One Level Blocking	9555784	9216056	635049	9576916	4.39	2.002
Two Level Blocking	11029758	11061102	54423	1095367	4.52	1.099

```
void block
   (float(*restrict a)[262],float(*restrict b)[262],(float(*restrict c)[262);
float a[262][262], b[262][262], c[262][262];

main(){
  int i, j, k,, ii, jj;

  memset( 0, c, sizeof(c) );
  for( i=0; i<240; i+=30 )
    for( j=0; j<240; j+=30 )
      for( k=0; k<240; k+=30 )
        for( ii=i;   ii<i+30; ii+=6 )
          for( jj=j; jj<j+30; jj+=6 ){
            block( &a[ii][k  ], &b[k  ][jj], &c[ii][jj], &c[ii][jj] );
            block( &a[ii][k+1], &b[k+1][jj], &c[ii][jj], &c[ii][jj] );
            block( &a[ii][k+2], &b[k+2][jj], &c[ii][jj], &c[ii][jj] );
            block( &a[ii][k+3], &b[k+3][jj], &c[ii][jj], &c[ii][jj] );
            block( &a[ii][k+4], &b[k+4][jj], &c[ii][jj], &c[ii][jj] );
            block( &a[ii][k+5], &b[k+5][jj], &c[ii][jj], &c[ii][jj] );
          }
}
```

Figure 38 Blocking Matrix Multiplication

Watch for Data Cache Bank Conflicts

The parameters to the program in Figure 39 are sets represented as bit vectors. The program tests whether one vector is included in another. 5027 cycles are necessary to test

for inclusion of a 1024-word vector in another. 1024 of these are data cache stalls. The **tmsim** statfile line corresponding to the loop is given below:

tree name	execs	instc	istallc	dstallc	cpbacks	cnflctc	isopers	exopers
__subset_DT_1	128	2176	64	275	0	896	5632	5374

The **cnflctc** column of the statfile is for data cache bank conflicts. All the stalls are bank conflicts.

There is a bank conflict whenever two memory accesses are made in the same cycle and bits two to four of the address are identical. This is the case in the program of Figure 39. The procedure **malloc** returns a pointer whose value is 4 mod 2^n, 2^n being the power of two immediately greater than or equal to the size. The pointers are, therefore, equal, mod 2^{12}. The same index is used inside subset to reference both arrays. The two loads are scheduled in the same instruction because of scheduling latency constraints. A cycle is added for every access to the two arrays as a result. Adjusting the addresses by allocating an extra word and incrementing one of them so that bits two to four differ eliminates the conflicts.

```
int subset( int *b, int *a, int size ){
   int i, result = 1;

   for (i=0; i<size; i+=8, a+=8, b+=8)
       result &= !(b[0] & ~a[0]) & !(b[1] & ~a[1])
              & !(b[2] & ~a[2]) & !(b[3] & ~a[3])
              & !(b[4] & ~a[4]) & !(b[5] & ~a[5])
              & !(b[6] & ~a[6]) & !(b[7] & ~a[7]);
   return result;
}
main(){
   (void)subset( (int*)malloc(1024*sizeof(int)),
                 (int*)malloc(1024*sizeof(int)),
                 1024 );
}
```

Figure 39 Set Inclusion

Summary

The TriMedia Compilation System is geared toward profile-based program optimization methods. Some methods, such as grafting, are done automatically by the compiler while taking into account user directives in the form of grafting parameters.

The version 2.0 compiler also performs improved if-conversion, function inlining, and automated loop unrolling. The automated loop unrolling can interfere with grafting, so it should be checked if grafting offers any advantages on top of the automated loop unrolling. As the case studies have shown, in some cases it does and in other cases it does not.

Other techniques, such as manual loop unrolling, the use of restricted pointers, and custom operators, currently need user intervention. It is expected that future versions of the compiler will include interprocedural optimization, source-to-source transformations, and better alias analysis. However, mechanisms like restricted pointers to pass specific user knowledge to the compiler, and the use of custom operators to exploit the TriMedia architecture to the maximum extent are likely to remain in the application programmer's domain.

To get started in optimization, you should first compile the program and run it with the **-O3** and **-p** (profile) option to make the decision tree frequencies and probabilities appear in the tree code. It is a good idea to look at the assembly code and, because cache misses are a significant performance factor, in most applications. You can then find the hot spots using **tmprof**. The C source and the tree file (.t) and assembler (.s) output corresponding to the hotspots should be examined together.

An important number of **after** keywords in the tree code for loads and stores usually indicates a need to use restricted pointers. The shape of the decision trees (number of leaves, branch structure, probabilities) provides important information about program restructuring. Frequently executed decision trees containing only a few operations indicate a control flow problem.

You must also pay attention to the memory behavior of the program. Unfortunately, memory statistics are only provided on a global basis by the current version of the SDE. You can recognize problems in the critical path, including aliasing, from sequences of **nops** in the assembler code.

You should restructure the C code, compile it with profiling, run it again with **tmprof**, and recompile and analyze it as many times as necessary. You need to apply grafting last because it is impossible to understand what is happening after grafting. You should apply loop unrolling only if it produces better performance than grafting. To determine this, measurement is necessary. It is a good idea to prepare a sample input smaller than the full set so as not to lose time running **tmsim**. On a Sparcstation 20, the ratio of real time to simulated time is about 36000 to 1. Understanding the instruction set helps in optimization. The ILP factors reported in the statfile and by **tmprof** can include operations that become redundant after optimization. These figures should be taken as estimates.

There are tools for performance analysis not mentioned in this chapter. For instance, there are tools to investigate produced schedules in detail by report options to **tmsched** and use of the tool **tmcritpath** to investigate the critical path of a schedule.

Besides the support for optimizing programs, the TriMedia Compilation System offers support for system level programming. Interrupt service routines can be programmed in C and support is added for using the most interesting cache instructions.

Chapter 11

System Programming Support

Programming Support

The TriMedia Compilation System offers system level programming at the C level. For instance, interrupt service routines and fine control of the data cache are supported. The toolset comprises interrupt latency inspection and offers support for interrupt latency control. This section describes what the toolset offers to programmers needing one of these features.

Interrupt Service Routines and Exception Handlers

The TriMedia C compiler allows you to create interrupt service routines (*handlers*, for short) and exception handlers entirely in C. The distinction between interrupt handlers and exception handlers is made clear in the next section, *User View*. First we discuss the general mechanism and do not distinguish between the two types of handlers.

The compiler allows maximal flexibility in handlers, and transparently generates code that uses the appropriate return address and does the additional register saving that is required for certain types of handlers. As for normal functions, the compiler attempts to minimize the calling overhead of handlers. Because handlers are nonstandard, this section contains some implementation detail to explain what you can expect from them.

User View

For your purpose, the only difference between handlers and functions is the way in which they are activated. You must explicitly call functions, whereas you must activate handlers upon an interrupt. Note that the compiler checks the type and parameter list of a handler but does not check erroneous calling of interrupt handlers. Normal functions which attempt to mimic a handler cause failures under certain conditions, because handlers have different register saving requirements.

Except from the fact that handlers are not allowed to return a result, there are no further differences between functions and handlers. Any legal resultless function with the specified number of parameters can be declared as a handler and therefore, the handler's body can range from simple updates to some flag in shared memory to complex control flow using conditionals, loops, and calls to other functions. However, as is the case for any C function, the calling overhead is strongly dependent on the complexity of the handler. See also the description of the calling sequences generated by the compiler for functions and handlers in *Declaring Interrupt Service Routines* on page 59.

Handlers come in three varieties: *interruptible*, *non-interruptible,* and *exception*. The first two are interrupt handlers that you can use for any of the vectored interrupts specified for the TriMedia processor. You can use an exception handler for any type of exception, such as misaligned store exception, floating point exceptions, and so on. The interrupt handlers have no parameters and come in a noninterruptible and an interruptible form. The difference is that interruptible handlers allow service of new interrupts of any kind

during their invocation (that is, *nested* interrupts), while noninterruptible handlers clear the interrupt enable bit (**IEN**) in the processor status word during their invocation and, therefore, can only be interrupted by *nonmaskable interrupts* (NMIs). This simple distinction between interrupt handlers is useful in many cases. However, sometimes you might require a finer level of interrupt masking. You must explicitly code such finer level masking using saving, modifying and restoring of the **IMASK**. For details on this, see the appropriate TriMedia data book.

Exception handlers are interruptible and get one parameter, the value of **spc** (saved program counter). You should install exception handlers with care because they might interfere with the exception handler installed by the debugger. The debugger uses an exception handler to single-step (dtree steps) through a program. So any user program should be certain not to destroy the debugger's handler.

When pSOS is not running, handlers make use of the stack of the process that was active at the moment of the interrupt. As of the 2.0f release, pSOS provides a system stack and interrupts can be written to run on the system stack. This leads to a more efficient and predictable use of the various stacks. The size of the pSOS system stack is set in the sys_conf.h file using the macro **KC_SYSSTK**. Interrupt service routines must switch explictly to the system stack using the call **AppModel_run_on_sstack(func,param)**, where **func** is a pointer to a normal function, and **param** is a **void** pointer that will be passed to that parameter.

For example:

```
static void isrFunc( param ){
   /* not declared as handler */
}
static void ISR( void ){
#pragma TCS_handler
   AppModel_suspend_scheduling();      /* OS-independent version of ienter */
   AppModel_run_on_sstack( (AppModel_Fun)isrFunc, (Pointer)&param );
/* param pointer is dereferenced before passing to isrFunc */
   AppModel_resume_scheduling();      /* OS-independent version of ireturn */
}
```

Here, **ISR** is the actual handler called by the hardware, and **isrFunc** is called on the system stack. You can see that the use of this mechanism is optional.

As of July 1999, the TSSA renderers and digitizers use the system stack.

```
#include <tm1/MMIO.h>

volatile int s;

void handler1(void){
   #pragma TCS_handler
   s++;
}

void handler2(void){
   #pragma TCS_handler
   int i;

   for (i=0; i<100; i++)
      s += i;
   }

void handler3 (void){
   #pragma TCS_interruptible_handler
   do_the_work_while_allowing_interrupts();
}

/* ---------------------------------------- */

void install_handler(
   int           nr,
   handler_type  handler
){
   base_of_mmio[ INT_VECS + nr ] = handler;
}

/* ---------------------------------------- */

main(){
   install_handler(1,handler1);
   install_handler(2,handler2);
   install_handler(3,handler3);
}
```

```
{_handler1:}
{__handler1_DT_0:}
   3 uimm (_s);
   2 ld32 3;
   4 iaddi (1) 2;
   5 st32 3 4
               after 2;
   6 readdpc;
   cgoto 6;
endtree (*__handler1_DT_0*)
```

Figure 40 Sample Interrupt Handler

Saving/Restoring Behavior

Similar to functions, handlers save and restore all the callee-saved registers that they use (including the frame pointer). Contrary to functions, handlers obtain their return address from the processor's destination program counter (**DPC**). In case a nested interrupt is possible during the execution of the handler, the **DPC** is also saved at entry.

Unlike normal functions, caller-saved registers that might be modified by executing the handler are also saved and restored at the handler's entry and exit. For handlers that call other functions, this means that the entire set of caller-saved registers is saved and restored. For handlers that do not call other functions, this means that caller- saved registers are treated as callee-saved registers, that is, only saved and restored when used by

the handler itself. Note that argument registers and the return pointer register are special cases of caller-saved registers.

Additionally, the code generated for noninterruptible handlers can save and restore the interrupt enable bit (**IEN**) in the processor status word. Any other change to the processor state during the handler's invocation remains visible after termination of the handler. This especially holds for the *source program counter* (**SPC**), which is used during exception processing to determine the decision tree in which the exception occurred.

Declaring Interrupt Service Routines

Interrupt handlers must be defined as parameterless, resultless functions, with either pragma **#pragma TCS_handler**, or **#pragma TCS_interruptible_handler** (as appropriate) in the beginning of the function body. By default (that is, declared as **TCS_handler**), handlers are noninterruptible.

You must define exception handlers as a resultless function with a single void . parameter, using the pragma **#pragma TCS_exception_handler**. Remember the warning that any explicit installation of an exception handler might cause the debugger to malfunction.

Figure 40 shows some examples of handlers, and of a simple generic function that is used to install them in the interrupt vector region of the MMIO space. See the data book and *Using MMIO Locations* on page 63.

Included in the figure is a sample translation to decision-tree intermediate code of a very simple handler.

```
int s;

void raise_s( void ){
   #pragma TCS_handler
   s = 1;
}

main(){
   s = 0;

   while( !s ){}

   printf("terminated \n");
}
```

DOES NOT WORK!

Figure 41 Use Volatile Shared Variables

Usage Notes

You must be aware that interruptible handlers, allowing nested interrupts, require some additional care. First, you must make reentrant, a nested invocation of a same handler does not corrupt the invocation state of the interrupted one. Second, with the possibility of nested interrupts, it has become essential to guarantee that the occurrence rate of interrupts does not exceed the system's capacity to service them for any longer amount

of time. Where such a situation only results in slow response or the loss of events in systems that do not allow nesting of interrupts, it might cause a crash in systems that do.

Any data that is shared between handlers and the mainstream program, or between handlers and other handlers, must be declared volatile to prevent surprising effects caused by optimizations. The optimizer might decide to replace loads from nonvolatile variables by earlier results, which might not be desired for shared data. An example is shown in Figure 41, which shows an illegal way for synchronization on an event: because the shared variable **s** is nonvolatile, the optimizer propagates the earlier assigned value **0** to the loop test condition. This results in a never ending loop, even when the handler is triggered.

Interrupt-Latency Support

Real-time system programmers have to be sure that the interrupts that occur during execution are handled within a certain number of cycles. Because on TriMedia decision trees are executed as large chunks of critical sections (noninterruptible code), special care is taken in the hardware as well as in the toolset. This section discusses how you can find out interrupt latencies for your particular program. No automatic support to guarantee a certain interrupt latency is given[1].

This section also addresses how you can find out whether the interrupt latency is more than a given threshold and how you can modify the code to reduce the interrupt latency. The support offered is threefold. First, there is a means to inspect violations of a certain threshold of cycles executed between interruptible jumps. Second, there are statistics from the simulator that produce a raw data histogram describing how many dtrees are executed with a certain number of cycles between interruptible jump. Also, the last dtree that executed that many cycles is shown. Third, there is a pragma, **TCS_break_dtree**, honored by the compiler, with which you can force the compiler to create smaller critical sections.

```
tmsim -il -mm a.out
interrupt latency distribution
000004 002644 __vfprintf_DT_16 + 1c
000005 002592 ____sinit_DT_0 + 1b84
000006 000799 ____swrite_DT_2 + 28
000007 000173 __fflush_DT_3 + 26
000008 002584 __vfprintf_DT_10 + 39
000009 000462 __vfprintf_DT_34 + 5c
*** several lines deleted
000223 000001 ___fwalk_DT_0 + 106
000289 000001 __latency_isr_DT_0 + 10b
```

1. The rationale for this is that automatic support has to be based on worst-case assumptions for all instructions executed. This is a *very* unrealistic situation, especially when assuming each load/store leads to cache miss, on top of losing the maximum number cycles for arbitration and the maximum number of requests serviced before getting the bus. Our experiments with a major application like MPEG-1 + RTOS showed that the number of cycles executed between two interruptible jumps was worst case 30µs, while the TriMedia hardware survives 300 µs. The TriMedia DMA-based peripherals have no short-term real-time constraints (order of several milliseconds) and the most critical peripheral is the synchronous serial interface SSI.


```
000411 000001 ___fwalk_DT_4 + 17b
000419 000001 __memchr_DT_2 + 304
000758 000001 __foo_DT_0 + 457
```

The first column in the report is the number of cycles between two interruptible jumps. The second column is the number of times a dtree was executed with that number of cycles. The last column names the dtree and address of the exit from the dtree last executed for the given number of cycles between interruptible jumps. For example, the last line means that the execution of the tree **__foo_DT_0** required 758 cycles and the number of cycles between two interruptible jumps of 758 was seen only once during the execution of the program. Similarly, during the execution of the entire program there were 462 instances where the number of cycles between two interruptible jumps was equal to 9. Among these, the last time this happened was while executing the dtree at **__vfprintf_DT_34**.

Supporting Cache Control

You can use the cache operations as specified in the data book (**dcb**, **dinvalid**, **iclr**, **rdstatus**, and **rdtag**) through the **custom_op** mechanism discussed in *Using Custom Operators* on page 36. But when you use these instructions directly (through a **custom_op** declaration), the ordering of memory accesses and the cache custom operations does not work.

```
void _cache_invalidate( void *address, int size );
void _cache_copyback  ( void *address, int size );
void _cache_allocate  ( void *address, int size );
```

The TriMedia C library has two entry points **cache_copyback** and **cache_invalidate** that allow you to maintain software coherency between the SDRAM and the data cache.

However, the TriMedia C library supports a more 'user view' model for using the most interesting custom operations at the C level, for example, copying back, allocating, or invalidating a piece of memory using functions declared in tmlib/tmlibc.h.

As an example, take the entry point **_cache_invalidate**. The semantics are: invalidate the piece of memory [address, address + **size**[1]). *The entire contents of the cache blocks in the range will disappear.* Any dirty data will be lost. Calls to **_cache_invalidate** are translated to issues of the appropriate number of **dinvalids**. The object referenced by the pointer should be cache aligned with respect to its upper and lower bounds.

The **_cache_copyback** entry point flushes dirty data back to the cache. This can be used prior to starting DMA or before a cache_invalidate if required. Unlike invalidates, copybacks are not destructive and the range does not need to be aligned. The **_cache_allocate** entry point resets the dirty bit in all data in the range. The memory range should be cache aligned.

In addition to the functions **_cache_allocate**, **_cache_copyback** and **_cache_invalidate**, the user can perform cache allocation, cache copyback and cache invalidation directly, using the custom operators **ALLOCATE**, **COPYBACK** and **INVALIDATE** defined in ops/

1. Currently, the cache block size is 64 bytes.

custom_defs.h. As noted above, the user should *not* use the underlying hardware cache operations directly as custom_ops in a program, because the resulting ordering of operations in the generated program may be incorrect. The C compiler translates the **ALLOCATE**, **COPYBACK** and **INVALIDATE** custom operators into the required series of hardware cache operations as specified in the data book, and assures the appropriate ordering of all memory operations which alias with the specified memory region. In general, these custom operators should be used only for extremely time-critical code; in most situations, the user should call the corresponding function instead.

The example from Figure 42 compiled with **tmcc -O3 -t example.c** translates into the trees code in Figure 43. In the example, note that ***c = 3;** aliases with all other memory locations because the address of **c** is unknown. The **INVALIDATE** call at the C level is translated into two **dinvalid** operations, which are ordered among all aliasing memory operations. The store to value **a[1]** is ordered only against the first **dinvalid** operation, because the compiler assumes[1] it overlaps only with the first cache line. In general, the input ordering of all cache operations is maintained. Note that the assignment **b[10] = 3;** is free to move across the **dinvalid** operations.

```
#include <custom_defs.h>
char   a[1000];
char   b[1000];
char   *c;

foo(){
   *c = 3;  b[10] = 4;
   INVALIDATE(a, 2);
   a[1] = 1;
   return a[1];
}
```

Figure 42 Example of Use of Cache custom_ops

The other cache operations are completely analogous to the **invalidate custom_op** and are not discussed any further.

```
{_foo:}
{__foo_DT_0:}
entree (0)
   2 uimm (_c);
   1 ld32 2;
   4 iimm (0x3);
   3 st8 1 4               (* *c = 3; *)
      after 1;
   6 uimm (_b);
   8 iimm (0x4);
   7 st8d (10) 6 8         (* b[10] = 4; *)
      after 3;
   10 uimm (_a);
   11 uimm (0x40);
   9 dinvalid(0) 10        (* invalidate [a, a+64) *)
      after 3;
   12 uimm (64 + _a);
```

1. Note that the assumption might not be true when the address (the first parameter to the invalidate call) is not cache-block aligned, as in this case!


```
   13 dinvalid(0) 12       (* invalidate [a+64, a+128) *)
      after 9 3;
   16 rdreg (1);
   15 st8d (1) 10 16       (* a[1] = 1; *)
      after 9;
   18 wrreg (5) 16;
   19 rdreg (2);
   cgoto 19
endtree (*__foo_DT_0*)
```

Figure 43 Intermediate Representation for Cache custom_op Example

Using MMIO Locations

Writes to MMIO locations do not take effect immediately. For example, if there is a write to the **IPENDING** location in cycle i that generates an interrupt, the interrupt is not triggered if an **ijmpi** operation is executed in cycle i+1. The interrupt is taken if the **ijmpi** operation was executed in cycle i+2. The amount of delay required for a write to an MMIO location is dependent on the location. Fore more information, refer to the databook.

Chapter 12

Case Studies

Introduction

Figure 44 shows the source code for a block FIR filter with floating-point arithmetic. The filter has been structured as a general-purpose library routine. The array of filter coefficients are supplied in an argument. The filter components are computed element by element. A separate function **dotap** is used to compute an element.

```
void blkfir( float *input, float *state, float *coeff,
             float *output, int npoints, int ntaps
){
   int i;

   for( i=0; i<npoints; i++ ){
      output[i] = dotap(input[i], state, coeff, ntaps);
   }
}
float dotap( float input, float *state, float *coeff, ntaps ){
   int i;
   float sum = 0.0;

   state[0] = input;
   for( i = ntaps; i>0; i-- ){
      state[i] = state[i-1];             /* slide window */
      sum = sum + state[i] * coeff[i];
   }
   return sum;
}
```

Figure 44 General-Purpose Block Filter

Special-Purpose Block Filter

Speed is critical when writing a routine that is specialized for a particular purpose. Implementation of a filter requires memorization of state information. You must use an array to represent the state if the size is arbitrary. Fixing the length allows it to be stored in scalars. These can be allocated to one of TriMedia's 128 general-purpose registers. If the length is variable, a loop is needed to evaluate the output value. This adds a control dependence that limits ILP. By fixing the state length beforehand, you can use a closed form expression. This has more ILP. In the example of Figure 44, the program is divided into two functions. This interferes with the optimizing ability of the compiler.

In the program of Figure 44, the array **coeff** is provided as a parameter. Only two accesses to memory can be made per instruction on TriMedia. If the routine is specialized for a particular set of coefficients, these can be placed as constants in the instruction stream. This reduces memory accesses and eliminates latency.

```
void blkfir( float *input, float *output, npoints ){
  int i, j;
  float state1, state2, state3, state4, state5, state6, state7, state8;

  state2 = state3 = state4 = state5 = state6 = state7 = state8 = 0.0;
  for( i=0; i < npoints; i++ ){
```



```
    state1 = input[i];
    output[i] = state1*0.5 + state2*0.25 + state3*0.125 + state4*0.0625 +
    state5*0.03125 + state6*0.015625 + state7*0.0078125 + state8*0.00390625;
    state8 = state7;  state7 = state6;  state6 = state5;  state5 = state4;
    state4 = state3;  state3 = state2;  state2 = state1;
  }
}
```

Figure 45 Specialized Filter

Figure 45 shows a specialized version of the routine for a state length of eight and a fixed set of coefficients. Eight scalar variables are used to represent the state. The two functions have been collapsed into one. Table 21 compares the performance of the two programs. After elimination of the loop and the arrays for **coeff** and **state**, only 1129 instruction cycles are necessary, compared to 5611 previously.

```
void blkfir(int *input, int *output, npoints){
   int state1 = 0, state2 = 0, state3 = 0, state4 = 0;
   int state5 = 0, state6 = 0, state7 = 0, state8 = 0;

   for( i=0; i<npoints; i++ ){
      state1 = input[i];
      output[i] = IMULM(state1, 0x10000000) + IMULM(state2, 0x08000000)
                + IMULM(state3, 0x04000000) + IMULM(state4, 0x02000000)
                + IMULM(state5, 0x01000000) + IMULM(state6, 0x00800000)
                + IMULM(state7, 0x00400000) + IMULM(state8, 0x00200000);
      state8 = state7; state7 = state6; state6 = state5; state5 = state4;
      state4 = state3; state3 = state2; state2 = state1;
   }
}
```

Figure 46 Filter with Fractional Arithmetic

Table 21 Special-Purpose Versus General-Purpose Filter

	ILP	Instruction Cycles		
		Per Tap	Per Input	Total
General Purpose Filter	1.06	17.5	140	5611
Special Purpose Filter	1.42	3.52	28.2	1129

Fixed-Point Arithmetic

Seven additions are necessary for each iteration of the loop of Figure 45. These have a latency of three cycles. Floating-point addition is commutative but not associative, for example, $10^8 + (-10^8 + 1)$ is not the same as $(10^8 + -10^8) + 1$. The C language requires that, in the absence of parentheses, floating-point arithmetic be executed in strict left to right order. In the program of Figure 45, this means that the additions must be executed in sequential order. A total of 21 cycles (7×3) is necessary to sum the seven products in sequential order using floating point.

Integer addition is both commutative and associative, so the compiler can balance the chain of additions in a tree, reducing dependences and increasing parallelism. Seven addition operations can be represented in a binary tree of height three. Integer addition has a latency of only one cycle. Three cycles (3×1) are necessary to sum the seven products in parallel. You can use integer arithmetic by changing to a fixed point representation.

You can represent fixed point numbers in what is called Q.*n* representation. The binary point is after the *n*th least significant bit. The bits to the right of the binary point correspond to the fractional part of the number. The most significant bit corresponds to the sign. The number of bits available for the integer part depends on the word length (16, 32, or 64 bits).

For this filter, the inputs are specified to be between –1 and +1. You can represent them in Q.31 form. The coefficients are between 0 and 1. The output of the filter is a sum of eight products between –1 and +1. It is between –8 and +8. Three bits are sufficient to represent the integer part (Q.28 form). The product of two numbers in Q.*n* and Q.*m* form is in Q.*n*+*m* form. If we represent the coefficient in Q.29 form and the input in Q.31 form, the 64-bit product is in Q.60 form. The high order 32 bits are given by the TriMedia **IMULM** instruction. This gives us a result in Q.28 form (60–32), as desired.

Figure 46 shows the source program after recoding to use fractional arithmetic. Table 22 shows the improvement due to the introduction of fixed-point arithmetic. Execution time is more than doubled, and the ILP is more than doubled, also.

Table 22 Fixed Versus Floating Point Arithmetic

	ILP (Inner Loop)	**Instruction Cycles**
Special Purpose + Floating Point	1.42	1129
Special Purpose + Fixed Point	3.33	489

IFIR16 Custom Operations

Changing to a fixed-point representation permits use of data-parallel custom operations. You can compute the sum of two products in a single **IFIR16** instruction. Recoding the algorithm to use **IFIR16** involves changing the representation from 32 to 16 bits. You must represent the inputs in Q.15 form, the outputs in Q.12 form, and the coefficients in Q.13 form. The smallest coefficient is 2^{-8}, which fits in 13 bits. The state and coefficients are represented in halfword pairs. The high order halfword corresponds to the first element and the low order halfword corresponds to the second element of the pair.

Representing elements in halfwords complicates the handling of the state. When there is a variable per-state element, shifting the state corresponds to seven register moves and one load. Up to five register moves can execute in parallel on TriMedia. When each register has two elements, shifting the state requires matching up the second element of each pair with the first element of the next. This corresponds to extracting the middle 32 bits of the 64-bit concatenation of the two pairs. This is possible with the TriMedia **FUNSHIFT2** instruction.

Figure 47 shows the source program after recoding to use **IFIR16** and **FUNSHIFT2.** To increase efficiency, the coefficient constants have been moved to registers. Table 23 shows the comparative performance with and without **IFIR16**. *The number of instruction cycles is reduced by 40%.*

Table 23 Comparative Performance

	ILP (Inner Loop)	**Instruction Cycles**
Special Purpose + Fixed Point	3.33	489
Special Purpose + Fixed Point + **Ifir16**	3.28	289

```
void blkfir( short *input, int *output, int npoints ){
   int i;
   int state01 = 0, state23 = 0, state45 = 0, state67 = 0;
   int coeff01 = 0x10000800, coeff23 = 0x04000200, coeff45 = 0x01000080,
       coeff67 = 0x00400020;

   for( i=0; i<npoints; i++ ){
      state01 = FUNSHIFT2(input[i],state01);       /* state1 = state0 */
                                                   /* state0 = inputi */
      output[i] = IFIR16(state01, coeff01) + IFIR16(state23, coeff23)
                + IFIR16(state45, coeff45) + IFIR16(state67, coeff67);
      state67 = FUNSHIFT2(state45,state67);        /* state67 = state56 */
      state45 = FUNSHIFT2(state23,state45);        /* state45 = state34 */
      state23 = FUNSHIFT2(state01,state23);        /* state23 = state12 */
   }
}
```

Figure 47 Filter with Custom Operations

Dual-Phase Loop

Several factors still limit the performance of the inner loop. The key factor is the 16-bit alignment of the state because of **IFIR16**. Shifting the state using **FUNSHIFT2** is cumbersome and slow. Only a halfword of data is read per cycle. A minimum of five cycles is necessary per output element because of the loop. You cannot reduce the overhead by unrolling because there is a dependence on the state.

```
void blkfir(int *input, int *output, npoints){
   int state1 = 0, state2 = 0, state3 = 0, state4 = 0;
   int state5 = 0, state6 = 0, state7 = 0, state8 = 0;

   for( i=0; i<npoints; i++ ){
      state1 = input[i];
      output[i] = IMULM(state1, 0x10000000) + IMULM(state2, 0x08000000)
                + IMULM(state3, 0x04000000) + IMULM(state4, 0x02000000)
                + IMULM(state5, 0x01000000) + IMULM(state6, 0x00800000)
                + IMULM(state7, 0x00400000) + IMULM(state8, 0x00200000);
      state8 = state7; state7 = state6; state6 = state5; state5 = state4;
      state4 = state3; state3 = state2; state2 = state1;
   }
}
```

Figure 48 Two-Phase Loop

You can sidestep all these restrictions by separating the execution of the loop into two phases. The **x** phase corresponds to the even-numbered outputs (**output**$_0$, **output**$_2$, **output**$_4$). The second phase corresponds to the odd-numbered outputs (**output**$_1$, **output**$_3$, **output**$_5$). Each has its own state. The state of the **y** phase corresponds to the state of the **x** phase shifted one input element. Two elements are processed per loop iteration. This allows register copies to be used instead of **FUNSHIFT2** for the state. Doubling the number of elements divides jump overhead by two. One half as many memory accesses (1×32 bits instead of 2×16) are made in the dual phase loop. Figure 48 shows the source program after recoding. Table 24 compares performances of the single and dual-phase loops.

Table 24 Single- and Dual-Phase Loop Comparison

	ILP (Inner Loop)	Instruction Cycles
Special Purpose + Fixed Point + **Ifir16**	3.28	289
Special Purpose + Fixed Point + **Ifir16** + Dual-Phase Loop	4.61	173

Table 25 summarizes the improvements in performance due to successive refinements of the program. There is a reduction by a factor of 32 in the execution time. The final program has more than 90% issue-slot utilization.

Table 25 Performance Improvements by Program Refinement

	ILP	Instruction Cycles		
		Per Tap	Per Input	Total
General Purpose + Floating Point	1.06	17.5	140.2	5611
Special Purpose + Floating Point	1.42	3.5	28.2	1129
Special Purpose + Fixed Point	3.33	1.52	12.2	489
Special Purpose + Fixed Point + **lfir16**	3.28	0.90	7.2	289
Special Purpose + Fixed Point + **lfir16** + Dual-Phase Loop	4.61	0.54	4.3	173

Critical Path

Horner's algorithm for evaluating a polynomial is shown in Figure 49. The array **P** gives the coefficients. $P(x) = (x+1)^{20}$. Thus, $P(-1) = 0$, $P(0) = 1$, and $P(1) = 2^{20}$.

```
#include <stdio.h>
#define DEGREE   20
float P[DEGREE+1] = {
   1, 20, 190, 1140, 4845, 15504, 38760, 77520, 125970, 167960, 184756,
   167960, 125970, 77520, 38760, 15504, 4845, 1140, 190, 20, 1
};
float poly_eval( float *a, int size, float x ){
   float result = 0;
   while (size >= 0) {
      result = result * x + a[size];
      --size;
   }
   return result;
}
main(){
   printf("y = %f\n", poly_eval(P, DEGREE, 1.0));
}
```

Figure 49 Polynomial Evaluation Using Horner's Algorithm

You can estimate the degree of ILP in the program by running **tmsim** with the **-statfile** option and examining the resulting file. Table 26 gives the line of the file corresponding to the **while** loop of the function **poly_eval.** Without grafting, the issue-slot utilization of 1.49 (188/126) is very low.

With grafting, the issue slot utilization is 1.61 (208/129), so it seems there is more ILP. However, more instructions are necessary to do the same task (129 versus 126).

The ILP is limited here by the length of the critical path. In the loop, the critical path corresponds to the calculation of a new value for the variable **result**. Each calculation requires a floating point multiplication, a floating point addition, and a coefficient load. The multiplication and the load both have a latency of three cycles. However, they can proceed in parallel. The addition has a latency of three cycles and depends on the other two operations. It is the **sum** of latencies (3 + 3 = 6) that determines the execution time. 126 cycles are necessary to execute 21 iterations of the loop.

Grafting reduces execution time when the ILP is limited by control flow. In this case, the three extra cycles with grafting are due to speculative evaluation during the final iteration. There are 21 values per result and two are evaluated per iteration.

Table 26 Line of the File Corresponding to the **while** Loop of the Function **poly_eval**

	execs	**instc**	**istallc**	**dstallc**	**cpbacks**	**cnflctc**	**isopers**	**exopers**
No Grafting	21	126	29	24	0	0	189	188
Grafting	6	129	88	31	0	0	211	208

Algebraic Transformation

Horner's algorithm is optimal in terms of the number of operations, but it is inherently sequential. By means of an algebraic transformation, you can multiply the parallelism by two. You can decompose polynomial *P(x)* into the sum of two polynomials, *Q(x)* and *Q'(x)*, corresponding to the even and odd powers of x, respectively. It is then possible to substitute $x \times R(x)$ for *Q'(x)*, where *R(x)* is a polynomial having only even powers of x also. At this point, we can substitute $y = x2$ in *Q(x)* and *R(x)*, because both polynomials contain only even powers of *x*. For example:

$$P(x) = a0 \times x0 + a1 \times x1 + a2 \times x2 + a3 \times x3 + a4 \times x4 + a5 \times x5$$
$$Q(y) = a0 \times y0 + a2 \times y1 + a4 \times y2$$
$$R(y) = a1 \times y0 + a3 \times y1 + a5 \times y2$$

You can evaluate the polynomials *Q(x)* and *R(x)* in parallel using Horner's rule, doubling the parallelism. Figure 50 shows source for a parallel version of **poly_eval**. An adjustment is necessary for the case where there is an odd number of coefficients (in this case, Q and *Q'* have differing degrees). This corresponds to a problem that occurs frequently when programming an unrolled loop with a variable size input in TriMedia. There is a reduction in the number of cycles from 129 to 101 for the parallel version. This is somewhat disappointing.

```
float poly_eval(float *a, int size, float x){
   float result1, result2, y;
   int adj;

   y = x * x;
   adj = (size+1) & 1;
   size -= adj;
   result1 = IZERO(adj, a[size+1]);
   result2 = 0;
   while (size > 0) {
      result1 = result1 * y + a[size-1];
      result2 = result2 * y + a[size];
      size -= 2;
   }
   return result1 + result2 * x;
}
```

Figure 50 Parallel Polynomial Evaluation

Balancing the Critical Path

Looking at Figure 50, note that **result1** and **result2** are calculated from **a[size]** and **a[size–1]**. The reference to **a[size]** corresponds to the scaled index addressing mode on TriMedia. Calculating **a[size–1]** requires one more cycle for the subtraction. The critical path is unbalanced as a result. There are several ways to balance the critical path. You can use pairs of index variables (**a[size]**, **a[size1]**), for example. In this case, the best solution is to adapt the algorithm to use pointers instead of indices for the arrays. The modified source code is shown in Figure 51. The references to **ap[0]** and **ap[–1]** correspond to the displacement addressing mode on TriMedia. Using pointers, only 81 cycles (compared to 101) are necessary.

```
float poly_eval(float *a, int size, float x){
   float result1, result2, y, *ap;
   int adj;

   y = x * x;
   adj = (size+1) & 1;
   size -= adj;
   ap = &a[size];
   result1 = IZERO(adj, ap[1]);
   result2 = 0;
   while( ap > a ){
      result1 = result1 * y + ap[ 0];
      result2 = result2 * y + ap[-1];
      ap -= 2;
   }
   return result1 + result2*x;
}
```

Figure 51 Balanced Critical Path

More Unrolling

Figure 52 shows the source for **poly_eval** when the loop has been unrolled to evaluate four polynomials in parallel. With unrolling, only 61, as compared to 81 cycles, are necessary to evaluate the polynomial. If the coefficients and degree of a polynomial are fixed in advance, more reduction in execution time is possible. Only 31 cycles are necessary to evaluate a polynomial of degree 20 on TriMedia. Source is given in Figure 53. Table 27 summarizes the time required to evaluate $(x+1)^{20}$, depending on the algorithm.

Table 27 Time Required to Evaluate $(x+1)^{20}$

Calculation Of 21-Point Polynomial	Instruction Cycles
Horner's Algorithm (Figure 49)	126
Two-Way Parallel (Figure 50)	101
Two-Way Parallel with Pointers (Figure 51)	81
Four-Way Parallel with Pointers (Figure 52)	61
Fixed $(x+1)^{20}$ Algorithm (Figure 53)	31

```
float poly_eval(float *a, int size, float x){
   float result1, result2, y, *ap;
   int adj;

   y = x * x;
   adj = (size+1) & 1;
   size -= adj;
   ap = &a[size];
   result1 = IZERO(adj, ap[1]);
   result2 = 0;
   while (ap > a) {
      result1 = result1 * y + ap[0];
      result2 = result2 * y + ap[-1];
      ap -= 2;
   }
   return result1 + result2*x;
}
```

Figure 52 Balanced Critical Path

```
float poly_eval( float x ){
   float result1, result2, result3, result4;
   float x2 = x*x, x3 = x2*x, x4 = x2*x2, x8 = x4*x4;

   result1 =    1 + x4* 4845 + x8*125970 + x4*x8*125970 + x8*(x8*4845+x8*x4);
   result2 =   20 + x4*15504 + x8*167960 + x8*x4* 77520 + x8*x8*1140;
   result3 =  190 + x4*38760 + x8*184756 + x8*x4* 38760 + x8*x8* 190;
   result4 = 1140 + x4*77520 + x8*167960 + x8*x4* 15504 + x8*x8*  20;
   return (result1 + result2*x) + (result3*x2 + result4*x3);
}
```

Figure 53 Source

Table 28 Instruction Cycles by Calculation

Calculation of 21-point polynomial	Instruction Cycles
fixed $(x+1)^{20}$ Algorithm (Figure 52)	31

Matrix Transpose

Computing the transpose of a matrix is useful in image processing. If the row and horizontal indices correspond to the *x* and *y* axis, transposition corresponds to a reflection about the *x*-*y* diagonal. Figure 54 shows a program. The dimension is coded as a power of two. The routine is used as follows:

```
#define SIZE 4 /* for a 16 by 16 matrix */
char matrix[1<<SIZE][1<<SIZE];
...
transpose(matrix);.
```

Table 29 indicates performance figures for different sizes. They were obtained with **tmprof** and **tmsim**.

The total execution time is the sum of the instruction cycles, the data cache miss cycles, and the instruction cache overhead (about 1000 cycles). The number of instructions and the number of memory accesses grows with the square of the image size. This is as expected. However, there is an explosion in the data cache overhead for a matrix of size 256×256. This is because the inner loop accesses the array in both row and column order. Each access to a byte in a row of the array brings in 63 other bytes. For the column order accesses, the data is used only after a full iteration of the outer loop. For n=256, an iteration of the outer loop overflows the 16K data cache. Also, each access only fetches a byte, even though 32 bits are available. This means that 75% of the memory bandwidth is wasted. Memory bandwidth is the critical limiting factor of this application. Accesses have a latency of three cycles. Cache misses have a latency of about 11 cycles for the critical word and about 30 cycles for the whole line.

```
void transpose(char *in){
   int i, j, t;

   for( i=0; i < (1<<SIZE); i++ )
      for( j=0; j<i; j++ ) {
         t = in[(i<<SIZE) + j];
         in[(i<<SIZE) + j] = in[(j<<SIZE) + i];
         in[(j<<SIZE) + i] = t;
      }
}
```

Figure 54 Iterative Matrix Transposition

Table 29 Performance Figures by Size

	Memory Accesses	Instruction Cycles	D-Cache Miss Cycles
16 x 16	574	1271	205
32 x 32	2142	4191	476
64 x 64	8350	15408	1420
128 x 128	33054	59344	5929
256 x 256	131614	233232	983218

Divide and Conquer

Two problems have to be dealt with. In a case where both the number of cache misses and the number of instructions need to be reduced, you should address the cache issues first because reducing cache overhead requires rethinking the algorithm. Figure 55 shows a solution to the matrix transposition problem, using the divide and conquer approach.

Table 30 Performance Figures by Size

	Memory Accesses	Instruction Cycles	D-Cache Miss Cycles
16 x 16	2534	2830	382
32 x 32	9654	10406	889
64 x 64	37718	40150	3380
128 x 128	149142	158006	11113
256 x 256	592045	627155	57520

```
void transpose( char * in, char * out, int step ){
   if (step == 0) {
      int t = in[0];
      in[0] = out[0];
      out[0] = t;
   }else{
      transpose(in, out, --step);
      transpose( &in[(SIZE + 1) << step], &out[(SIZE + 1) << step], step );
      transpose( &in[1<<step], &out[SIZE<<step], step );
      if( in != out )
      transpose( &in[SIZE<<step], &out[1<<step], step );
   }
}
```

Figure 55 Recursive Matrix Transposition

The matrix is divided into four equal-sized squares. The two squares along the diagonal are transposed in place. The two other squares are interchanged and transposed. On the initial call, the entire matrix is transposed in place:

```
transpose(matrix, matrix, SIZE);
```

For the recursive step, the two squares along the *x-y* diagonal are transposed in place. The two squares along the other diagonal are interchanged and transposed.

The parameter **step** indicates the array dimensions as a power of two. This is a naive algorithm that simply recurses until a 1×1 matrix is found. Table 31 indicates performance figures for different image sizes. For a 16×16 matrix, there are about five times as many memory accesses and 2.5 times as memory instruction accesses, compared to the iterative algorithm. However, the execution time is better for a 256×256 matrix because of better locality.

Using Custom Operations

The TriMedia has instructions that merge and pack bytes in registers in parallel. You can apply one of these instructions in this case to speed up the manipulation of bytes that are packed into words. Imagine that our task is to transpose a 4×4 matrix.

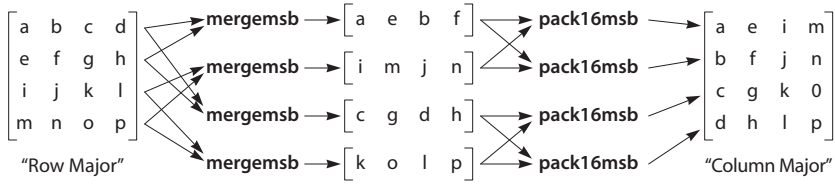

Figure 56 4×4 Transpose

Table 31 Performance Figures by Size

	Memory Accesses	Instruction Cycles	D-Cache Miss Cycles
16 x 16	362	448	237
32 x 32	1218	1252	949
64 x 64	4466	4268	3028
128 x 128	17106	16274	9292
256 x 256	65871	65948	43819

Figure 56 shows how you can use custom operations. Figure 57 extends the solution to a $2^n \times 2^n$ matrix. The elementary step is on four machine words. Table 32 shows the performance of the routine. For a 256×256 array, the overall execution time is ten times less than the iterative algorithm. For a 16×16 matrix, the execution time is two times less.

```
#include <ops/custom_defs.h>
#define WSZ  (SIZE/sizeof(int))

void transpose( int * in, int * out, int step ){
   if( step == 0 ){
      int im0 = MERGEMSB(in[0*WSZ],in[1*WSZ]),
          im1 = MERGEMSB(in[2*WSZ],in[3*WSZ]),
          im2 = MERGELSB(in[0*WSZ],in[1*WSZ]),
          im3 = MERGELSB(in[2*WSZ],in[3*WSZ]);
      int om0 = MERGEMSB(out[0*WSZ],out[1*WSZ]),
          om1 = MERGEMSB(out[2*WSZ],out[3*WSZ]),
          om2 = MERGELSB(out[0*WSZ],out[1*WSZ]),
          om3 = MERGELSB(out[2*WSZ],out[3*WSZ]);
      out[0*WSZ] = PACK16MSB(im0,im1); out[1*WSZ] = PACK16LSB(im0,im1);
      out[2*WSZ] = PACK16MSB(im2,im3); out[3*WSZ] = PACK16LSB(im2,im3);
      in [0*WSZ] = PACK16MSB(om0,om1); in [1*WSZ] = PACK16LSB(om0,om1);
      in [2*WSZ] = PACK16MSB(om2,om3); in [3*WSZ] = PACK16LSB(om2,om3);
   }else{
      transpose( in, out, --step );
      transpose( &in[(SIZE + 1) << step], &out[(SIZE + 1) << step], step );
```

```
      transpose( &in[1<<step], &out[SIZE<<step], step );
      if( in != out )
         transpose( &in[SIZE<<step], &out[1<<step], step );
  }
}
```

Figure 57 $2^n \times 2^n$ Matrix

Inlining and Shrink-Wrapping

The algorithm reads and writes the entire matrix once. For a 256×256 byte matrix, there are 16,384 word accesses for reads and 16,384 word accesses for writes (256×256/4). A total of 32,768 memory accesses are necessary. The 65,871 memory accesses are necessary running the program. Most of the other 33,103 accesses are spills of registers to the stack. These are generated by **tmccom**, the TriMedia core compiler.

You can use two techniques to reduce the number of stack spills. Inlining one level of recursion reduces the function call overhead by a factor of about three. You can remove the spills for applications of the elementary step using a technique known as shrink-wrapping.

Shrink-wrapping works by splitting a function into two parts. The first part contains the part of the function that calls other functions or itself (the nonleaf part). The second part contains the part of the function that corresponds to a leaf. This code is placed in a separate function. The compiler can use caller-saved registers instead of callee-saved registers here because it is a leaf. Overhead is also reduced in the nonleaf part because only its variables need to be spilled. The function call overhead in the leaf part is minimal.

You can use different techniques to inline a function, including using a preprocessor such as KAP, using the C preprocessor, and hand inlining. Both using the C preprocessor and hand inlining were tried with this example. Naively using the C preprocessor gave poor code.

The elementary step in **transpose** calls **transpose_leaf**. It operates on an 8×8 matrix. The code in **transpose** is as follows:

```
if( step==1 )
   transpose_leaf(in, out);
else ...
```

Figure 58 shows source for **transpose_leaf**. Figure 59 shows the C preprocessor macros used to operate on 4×4 submatrices. **READ4** reads a submatrix into four temporary variables. **WRITE4** writes out the transposed result. **MERGE4** corresponds to the intermediary step.

Table 32 shows the performance of the routine after inlining and shrink-wrapping have been applied. For a 256×256 array, 8194 out of 40962 memory accesses can be attributed

to spills. This corresponds to an overhead of 25%, which is acceptable. Eliminating spills divides the number of instructions by almost a factor of two.

Table 32 Performance After Inlining and Shrink-Wrapping

	Memory Accesses	Instruction Cycles	D-Cache Miss Cycles
16 x 16	262	294	207
32 x 32	768	684	1084
64 x 64	2834	2320	2757
128 x 128	10594	8864	9363
256 x 256	40962	36464	45202

```
void transpose_leaf( int *in, int *out ){
   int i0, i1, i2, i3;
   int im0, im1, im2,  im3,  im4,  im5,  im6,  im7;
   int im8, im9, im10, im11, im12, im13, im14, im15;

   READ4(&in[0      ]);  MERGE4( im0,  im1,  im2,  im3  );
   READ4(&in[1      ]);  MERGE4( im4,  im5,  im6,  im7  );
   READ4(&in[4*WSZ  ]);  MERGE4( im8,  im9,  im10, im11 );
   READ4(&in[4*WSZ+1]);  MERGE4( im12, im13, im14, im15 );

   READ4 ( &out[0]                    );
   WRITE4( &out[0], im0, im1, im2, im3 );
   MERGE4(          im0, im1, im2, im3 );
   WRITE4( &in[0],  im0, im1, im2, im3 );

   READ4 ( &out[1]                       );
   WRITE4( &out[1], im8, im9, im10, im11 );
   MERGE4(          im8, im9, im10, im11 );
   READ4 ( &out[4*WSZ]                      );
   WRITE4( &out[4*WSZ], im4, im5, im6, im7 );
   MERGE4(              im4, im5, im6, im7 );
   WRITE4( &in[1],      im4, im5, im6, im7 );

   WRITE4( &in[4*WSZ], im8, im9, im10, im11);

   READ4 ( &out[4*WSZ+1]                          );
   WRITE4( &out[4*WSZ+1], im12, im13, im14, im15 );
   MERGE4(                im12, im13, im14, im15 );
   WRITE4( &in[4*WSZ+1],  im12, im13, im14, im15 );
}
```

Figure 58 Source for **transpose_leaf**

Table 33 compares the performances of the original and final versions of the program, in cycles. An instruction cache overhead of 1000 cycles in both cases is assumed. Depending on the size of the input, the improvement in performance varies between 1.6 and 16.

```
#define READ4(x)    i0 = (x)[0*WSZ]; i1 = (x)[1*WSZ];           \
                    i2 = (x)[2*WSZ]; i3 = (x)[3*WSZ];

#define MERGE4(i,j) v0 = MERGEMSB(i0,i1); v1 = MERGEMSB(i2,i3); \
                    v2 = MERGELSB(i0,i1); v3 = MERGELSB(i2,i3);
```

```
#define WRITE4(x,i,j,k,l) (x)[0*WSZ] = PACK16MSB(i,j);          \
                          (x)[1*WSZ] = PACK16LSB(i,j);          \
                          (x)[2*WSZ] = PACK16MSB(k,l)           \
                          (x)[3*WSZ] = PACK16MSB(k,l);
```

Figure 59 C Preprocessor Macros Used To Operate On 4×4 Submatrices

Table 33 Performance of Original and Final Versions

	Original Program	**Final Program**
16 x 16	2476	1504
32 x 32	5667	2675
64 x 64	16828	5419
128 x 128	65634	17410
256 x 256	1226450	74336

Cache Alignment

Cache accesses have a granularity of 64 bytes and are aligned at 64-byte boundaries in memory. Fetching a structure of 64 bytes aligned at 64-byte boundary requires a single cache access compared to two for an unaligned access. Fetching an unaligned 32-byte structure requires one and one half cache accesses on average, compared to one for an aligned access. If the matrix is allocated on the heap, it can be aligned to a cache boundary. The number of memory accesses increases from 44041 to 68162 for a 256×256 matrix if it is not cache-aligned. The TriMedia C library routine **_cache_malloc** can be used for this. Code for the transposition routine to align the matrix is shown below. The second argument is the set number (0–31 where –1 means *any* cache set).

```
#define LINESIZE 64
a = (char*)_cache_malloc( SIZE*SIZE, -1 );
... initialize matrix ...
transpose( (int*)a, (int*)a, STEP );
```


Discrete Cosine Transform (DCT)

The purpose of this article is to explain the implementation of the Discrete Cosine Transform (DCT) on the TriMedia architecture and its inverse. The DCT is a key transform for multimedia encoding (video and audio) and has been adopted by the JPEG standards.

The article begins with a brief discussion of transform theory and an explanation of the DCT. Reference implementations are provided for those not already familiar with the DCT.

The 8×8 DCT used in TriMedia uses a separable algorithm based on the 8-point 1D DCT described in 1989 by Loeffler, Ligtenberg and Moschytz. The TriMedia-specific optimizations are described for both the 8×8 DCT and its inverse. Besides being of interest in itself, the DCT implementation provides an excellent in depth case study of optimization for TriMedia.

The following publications were referenced in the drafting of this section:

Practical Fast 1-D DCT Algorithms with II Multiplicators, Christoph Loeffler, Adrian Ligtenberg, George S. Moschytz, IEEE Proceedings, 1989.

Discrete Cosine Transform Algorithms, Advantages, Applications, K. R. Rao, P. Yip, Academic Press, 1990

Presentation to IEEE G.2.1.6 Video Compression Measurements Subcommittee on IEEE 1180-1990 Standard Discrete Cosine Transform Accuracy Test, Ken Vollmar, January 1998.

What is a Transform?

A transform changes the representation of a function or signal while maintaining its essential properties.

The Discrete Cosine Transform is based on the Fourier Cosine Transform (FCT). The FCT is a just a special case of the Fourier Transform applied to a function symmetric around the y axis. Like the Laplace Transform, the Fourier Transform is one of the key transformations in applied mathematics. It changes from a time to a frequency representation by applying an integral. The Inverse Fourier Transform changes the function back. The Fourier Transform uses basis functions that are complex exponentials. The Fourier Cosine Transform uses basis functions which are cosines. The Fourier Cosine Transform is easier to compute than the Fourier Transform because it is real valued.

The discrete equivalents of these transforms (Discrete Cosine Transform, Discrete Fourier Transform) operate on the function values sampled at a limited number of points (i.e., an array).

The basis function for a 1D function corresponds to a vector of values that is applied and summated (an operation called convolution). To have some intuition of how the DCT

works, an example is very helpful. The following are taken from running a program called the Transform Calculator (**tc**). We will first study a 2D 2×2 DCT.

```
$ tc
scale 3
{ 1 0 0 0 } rows 2
e1 =   1.000   0.000
       0.000   0.000
```

The scale operation indicates that on output 3 digits of precision are to be retained. The row operation organizes the 4 data values into a 2×2 matrix. The result is a matrix of having a non zero in the upper left hand corner. For a 2×2 DCT, this corresponds to a single basis function. This is the (0, 0) or so-called DC component.

```
inv dct e1
e2 =   0.500   0.500
       0.500   0.500
```

This shows the transform matrix for the DC component. The DC component is proportional to the average of all of the input elements. Actually, it is twice the average. This corresponds to the average background intensity or grey level for the 2×2 image.

The 4 (2×2) basis functions are completely independent. In mathematical language this is called orthogonal. Orthogonality means that the contribution from a basis function can be taken away from the image independently of the others. Applying this to the DC component corresponds to removing the background from the image.

```
{ 0 1 0 0 } rows 2
e4 = 0.000     1.000
     0.000     0.000
inv dct e1
e5 = 0.500    -0.500
     0.500    -0.500
```

The second or (1, 0) basis function corresponds to vertical lines. Subtracting the value of the basis function from the image removes these.

```
{ 0 0 1 0 } rows 2
e6 = 0.000     1.000
     0.000     0.000
inv dct e1
e7 = -0.500    -0.500
      0.500     0.500
{ 0 0 0 1 } rows 2
e8 =  0.500    -0.500
     -0.500     0.500
```

The third and fourth basis functions ((0,1) and (1,1)) corresponds to horizontal lines and diagonals, respectively. The orthogonality property of the discrete cosine transform means that an arbitrary image can be encoded using a maximum of four basis function values.

How the DCT Works

There were four basis functions for the 2×2 DCT, hence there are up to 4 coefficients. However, there are potentially much more zeroes in the DCT representation than in the original image data.

For example, for a fixed background there is only the DC component. The DCT takes advantage of the locality structure of an image. In mathematical language, the image data is said to be correlated. This just means that the colors on a particular point of a checkered T-shirt, for example, are much more likely to be close to that of a neighboring point than an arbitrary point.

The DCT concentrates correlated values corresponding to patterns in the image in one component. The 64 basis function coefficients for an 8×8 DCT correspond to 64 alternating 2D patterns. The rate of alternation or frequency corresponds to the (x,y) position in the coefficient matrix. It turns out that only a few basis functions are necessary to encode an 8×8 image in practise.

The DCT is applied to an 8×8 block rather than to the entire image to limit the number of calculations. n^4 calculations are required to calculate a n×n 2D DCT. The number of calculations can be reduced to n^3 using the separability property of the DCT (this will be explained later). Still, this is still too much to apply to an entire image.

The DCT in MPEG can be applied in combination with a technique called quantization to advantageously exploit properties of the human visual system. This allows the compression ratio to be reduced.

The last paragraph just means that the human eye is much more sensitive to simple than to complex patterns. In other words, the threshold of visibility of a background or thick 1D lines is greater than for a finally alternating 2D image.

In the DCT, the complex patterns correspond to high frequency components. Quantization divides the output of the DCT by a factor proportional to the complexity. Complexity is measured as a function of the distance from the (0, 0) point (the DC component) in the matrix. When decoding, an identical factor is multiplied in inverse (this is called dequantization).

The matrix after quantization has a large number of zeroes. A technique called run length encoding is used to eliminate these. Essentially, the matrix is made sparse.

After the DCT and quantization, there are typically only a few coefficients. MPEG and JPEG use Huffman encoding to encode these. Motion JPEG allows the quantization ratio to be dynamically adjusted. A coarser grain of quantization reduces the bit rate and a finer grain improves video quality.

In theory, any inversible transform could be used to encode the image and other transforms have been successfully used (e.g. wavelets).

The advantages of the DCT are fourfold.

1. It has been extensively studied since its discovery by Ahmed et. al. in 1974.

2. It can be very efficiently implemented as will be explained in this article.
3. It has been adapted by international standards (MPEG, JPEG).
4. Finally, the compression ratio obtainable with the DCT can be shown to be close to the theoretic maximum obtained with the Karhunen Loeve transform (this cannot be computed). The difference between wavelets and the DCT is in the choice of the basis functions (the wavelet basis functions are unevenly spaced).

Computation of a 1D DCT and Its Inverse

Formula 1 defines the one dimensional Discrete Cosine Transform. The DCT is just the discrete counterpart of the Fourier Cosine Transform as explained previously. The integral of the continuous transform is replaced by summation, where x_i represents the data to be encoded, and X_v is the encoded result. Note the similarity in form between the transform and its inverse.

$$X_v = \sqrt{\frac{2}{N}} \sum_{i=0}^{N-1} c_v \, x_i \cos\left[\frac{(2i+1)\,\pi\, v}{2N}\right] \qquad [1]$$

and $c_v = \frac{1}{\sqrt{2}}$ (≈ 0.707) for $i = 0$, and 1, otherwise. Formula 2 defines the inverse transform:

$$x_i = \sqrt{\frac{2}{N}} \sum_{v=0}^{N-1} c_v \, X_v \cos\left[\frac{(2i+1)\,\pi\, v}{2N}\right] \qquad [2]$$

Code to calculate a 1D DCT (forward transform or inverse) is shown, in sections. The arguments to the procedure **dct1d** are:

- A pointer to the input value.
- A pointer to the transformed result.
- The stepping factor between consecutive elements (1 if contiguous).
- The number of samples.
- A flag indicating whether the DCT (**fwd** = 1) or its inverse are to be calculated.

```
#include <math.h>
void
dct1d(double *tabval, double *ptres, int step, int dim, int fwd){
   register v, i;
   double sum, *ptval, coef;
   double facmul[2] ;
   facmul[0] = sqrt(2./dim) * M_SQRT1_2;
   facmul[1] = sqrt(2./dim);
   ...
```


These two values correspond to the scaling factors (for element 0 and for all others). The $\sqrt{1/2}$ factor (**M_SQRT1_2**) corresponds to the c_u term for the zeroth element in formulas 1 and 2 above.

Depending on the implementation, a denormalized DCT can be used. For example, the result of the 8-point 1D fast DCT introduced later is scaled by $\sqrt{2}$ from this.

```
   ...
   for( v=0; v<dim; v++ ){
      ptval = tabval;
      sum = 0;
      for( i=0; i<dim; i++ ){
         if( fwd )
            coef = cos(((2*i+1)*v*M_PI)/(2*dim)) * facmul[v!=0];
         else
            coef = cos(((2*v+1)*i*M_PI)/(2*dim)) * facmul[i!=0];
         sum   += *ptval * coef;
         ptval += step;
      }
      *ptres  = sum;
       ptres += step;
   }
}
```

The body of the function is a loop that computes the convolution in formulas 1 and 2 and element by element. For the inverse case, the formula varies because the loop order needs to be reversed.

Computation of the DCT is equivalent to multiplying the input values, represented as a vector by an n×n matrix representing the coefficients. The inverse DCT can be computed similarly. n^2 multiplications and $n^2 - n$ additions are necessary for this. The code above can be used to verify whether an particular implementation is correct or not, however, it is too slow to use in practise.

Computation of a 2D DCT

Data volume explodes with the number of dimensions (2D, 3D, etc.) in the uncompressed image. The DCT is remarkable in its ability to detect patterns in multidimensional data and remove wasted space. The formula below defines the two dimensional DCT.

An 8×8 DCT is used in the international JPEG and MPEG standards. The following formula is taken from the work by Rao and Yip, referenced on page 83,

$$G_{uv} = \sqrt{\frac{2}{\mathrm{M}}}\sqrt{\frac{2}{\mathrm{N}}}\sum_{i=1}^{\mathrm{M}-1}\cos\left[\frac{(2i+1)\,\pi\,u}{2\mathrm{M}}\right]\sum_{i=1}^{\mathrm{N}-1}c_u\;c_v\;g_{ij}\;\cos\left[\frac{(2j+1)\,\pi\,v}{2\mathrm{N}}\right] \qquad [3]$$

where g_{ij} is the data to be encoded (i = 0 to 7, j = 0 to 7), G_{uv} is the encoded result, and M=N=8.

Code shown to calculate a 2D DCT is shown below. The arguments are the same as for in the 1D case except there are two dimensions and no stepping factor.

```
void
fdct2d( double *tabval, double *ptres, int dim1, int dim2, int fwd ){
   int u,v,i,j ;
   double facu[2] , facv[2], sum, value, coef;

   facu[0] = sqrt(2./dim1) * M_SQRT1_2;
   facu[1] = sqrt(2./dim1);
   facv[0] = sqrt(2./dim2) * M_SQRT1_2;
   facv[1] = sqrt(2./dim2);
```

These correspond to the c_u and c_v factors in the formula 3.

```
   for( u=0; u<dim1; u++ ){
      for( v=0; v<dim2; v++ ){
         sum = 0;
         for( i=0; i<dim1; i++ )
            for( j=0; j<dim2; j++ ){
               value = tabval[(i*dim1) + j];
               coef = cos(((2*i+1)*u*M_PI)/(2*dim1)) *
                      cos(((2*j+1)*v*M_PI)/(2*dim2)) *
                      facu[u!=0]*facu[v!=0];
               sum += value * coef;
            }
```

The inner pair of loops compute the convolution for a single element.

```
            ptres[(u*dim1)+v] =  sum ;
         }
      }
   }
}
```

The outer pair of loops correspond to the rows and columns of the input and output.

An $m \times n$ transform requires a total of $(mn)^2$ multiplications and $(mn)(mn-1)$ additions. For the 2D transform, the elements of the transform matrix are themselves matrices.

Computation of the 2D IDCT

Formula 4 defines the 2D Inverse Discrete Cosine Transform (IDCT).

$$g_{ij} = \sqrt{\frac{2}{M}}\sqrt{\frac{2}{N}}\sum_{u=1}^{M-1}\cos\left[\frac{(2i+1)\,\pi\,u}{2M}\right]\sum_{v=1}^{N-1}c_u\;c_v\;G_{uv}\;\cos\left[\frac{(2j+1)\,\pi\,v}{2N}\right] \qquad [4]$$

Code to compute the 2D IDCT is shown in segments, following:

```
void
idct2d( double *tabval, double *ptres, int dim1, int dim2, int fwd ){
   int u,v,i,j ;
   double facu[2] , facv[2], sum, value, coef;

   facu[0] = sqrt(2./dim1);   facu[1] = sqrt(2./dim1) * M_SQRT1_2;
   facv[0] = sqrt(2./dim2);   facv[1] = sqrt(2./dim2) * M_SQRT1_2;
```



```
   for(i=0;i<dim1;i++) {
      for (j=0;j<dim2;j++) {
         sum = 0;
         for (u=0;u<dim1;u++) {
            for (v=0;v<dim2;v++) {
               value = tabval[(u*dim1) + v];
               coef  = cos(((2*i+1)*u*M_PI)/(2*dim1))*
                       cos(((2*j+1)*v*M_PI)/(2*dim2)) *
                       facu[!u]*facu[!v];
               sum += value * coef;
            }
         }
         ptres[(i*dim1)+j] =  sum ;
      }
   }
}
```

The code is very similar to the forward transform and requires the same number of operations. Like the 2D forward DCT, this algorithm is to slow to be used in practice.

Separability

Direct computation of the 2D DCT requires an excessive number of operations. The separability property can be exploited to reduce the number of computations. This means that 1D DCTs can be applied in all the dimensions.

To exploit separability, compute n 1D DCTs on the rows of the input followed by n 1D DCTs on the columns of the resulting matrix.

Using the formula for the 1D DCT computed previously, this corresponds to $n(m^2)$ multiplications and $n(m^2 - m)$ additions for an $n \times m$ DCT. The number of operations for the second pass is $m(n^2)$ and $m(n^2 - m)$.

An algorithm to compute a 2D DCT based on a 1D DCT using separability is shown.

```
void
dct2dsep( double *tabval, double *ptres, int dim1, int dim2, int fwd ){
   int u,v;
   double tabtemp[MAXVAL];

   for( u=0; u<dim1; u++ )
      dct1d(&tabval[dim2*u], &tabtemp[dim2*u], 1, dim2, fwd);
   for( v=0; v<dim2; v++ )
      dct1d(&tabtemp[v], &ptres[v], dim2, dim1, fwd);
}
```

Fast Computation of an Eight Point DCT

The basic idea behind a fast DCT algorithm is to take advantage of the highly regular structure of formulas 1 and 2 to reduce the number of multiplications. The cosine function is periodic modulo $2 \times \pi$. Consider, for example the case of a 1D four point DCT. The transform matrix below was computed by calculating the basis functions using the

inverse DCT as explained previously. Note that there are only two distinct values, not counting sign differences and shifts.

```
    0.5000    0.5000    0.5000    0.5000
    0.6533    0.2706   -0.2706   -0.6533
    0.5000   -0.5000   -0.5000    0.5000
    0.2706   -0.6533    0.6533   -0.2706
```

An eight point DCT can be efficiently computed using the algorithm cited by Loeffler, Ligtenberg and Moschytz on page 83. The algorithm calculates a denormalized result with a scaling factor C = $\sqrt{2}$.

The same factor is used for the inverse algorithm. This factor replaces the $\sqrt{2/N}$ factor in formulas 1 and 2.

Since N=8, this corresponds to an upscaling by a factor of 2 $\sqrt{2}$. The factor was chosen so that $C \times c_v = 0$ for $v = 0$, which allows the y_0 output to be computed without any multiplication.

A paper gives an algorithm to compute the DCT using 11 multiplications and 29 additions. This corresponds to the theoretical lower bound. However, multiplications are cascaded. It describes a variant requiring just 12 multiplications and 15 additions in which all the multiplications are in parallel. Code to compute an 8-point DCT according to this algorithm is shown below.

DCT algorithms have a rotation step where π /16 corresponds to the stepping angle for an 8-point DCT:

```
#define THETA   (M_PI/16)                   /* 11.25 degrees */
#define SIN_THETA (sqrt(2)*sin(2*THETA))
#define COS_THETA (sqrt(2)*cos(2*THETA))
```

These are the matrix coefficients for the even coefficients of the output. These are more straightforward to compute.

```
#define COEFF_a \
   sqrt(2.)*(-cos(1*THETA)+cos(3*THETA)+cos(5*THETA)-cos(7*THETA))
#define COEFF_b \
   sqrt(2.)*( cos(1*THETA)+cos(3*THETA)-cos(5*THETA)+cos(7*THETA))
#define COEFF_c \
   sqrt(2.)*( cos(1*THETA)+cos(3*THETA)+cos(5*THETA)-cos(7*THETA))
#define COEFF_d \
   sqrt(2.)*( cos(1*THETA)+cos(3*THETA)-cos(5*THETA)-cos(7*THETA))

#define COEFF_e sqrt(2.)*(-cos(3*THETA)+cos(7*THETA))
#define COEFF_f sqrt(2.)*(-cos(1*THETA)-cos(3*THETA))
#define COEFF_g sqrt(2.)*(-cos(3*THETA)-cos(5*THETA))
#define COEFF_h sqrt(2.)*(-cos(3*THETA)+cos(5*THETA))
#define COEFF_i sqrt(2.)*( cos(3*THETA))
```

These correspond to the matrix coefficients for the odd coefficients of the output. These are more difficult to compute. The scaling factor of $\sqrt{2}$ corresponds to the value explained previously.

```
void
dct8( double *tab, double *res, int step, int dim, int fwd ){
   double x0, x1, x2, x3, x4, x5, x6, x7;
```



```
   double y0, y1, y2, y3;
   double z1, z2, z3, z4, z5;
   double a, b, c, d, e, f, g, h, i;
   double o0, o1, o2, o3, o4, o5, o6, o7;
/* stage 1 */
   x0 = tab[0] + tab[7]; x1 = tab[1] + tab[6];
   x2 = tab[2] + tab[5]; x3 = tab[3] + tab[4];
   x4 = tab[3] - tab[4]; x5 = tab[2] - tab[5];
   x6 = tab[1] - tab[6]; x7 = tab[0] - tab[7];
```

The first stage is a "butterfly stage" that crosses the output. Eight additions are required (4 are subtractions of these).

```
/* stages 2 and 3, even part */
   y0 = x0 + x3;  y1 = x1 + x2;
   y2 = x1 - x2;  y3 = x0 - x3;
```

After the first stage, the algorithm separates for the even (0, 2, 4, 6) and odd (1, 3, 5, 7) numbered outputs.

```
o0 = y0 + y1;
o2 = o2 = COS_THETA    * y2 + SIN_THETA * y3;
o4 = y0 - y1;
o6 = o6 = (-COS_THETA) * y2 + SIN_THETA * y3;
```

The terms o_2 and o_6 correspond to a rotation (by θ) of y_2 and y_3.

```
/* stages 2 and 3, odd part */
   z1 = x4 + x7;   z2 = x5 + x6;  z3 = x4 + x6;
   z4 = x5 + x7;   z5 = (x4 + x6) + (x5 + x7);
```

Factorization is used to divide the odd part into three stages (2, 3, 4). x4+x6 and x5+x7 are common subexpressions (the parentheses are necessary for this).

```
   a = x4*COEFF_a;  b = x5*COEFF_b;  c = x6*COEFF_c;
   d = x7*COEFF_d;  e = z1*COEFF_e;  f = z2*COEFF_f;
   g = z3*COEFF_g;  h = z4*COEFF_h;  i = z5*COEFF_i;
```

The figure shows the matrix used for stage 3 (the odd part) of the algorithm. The non-zero entries correspond to constants defined previously (**COEFF_a, COEFF_b, ...**). These are all along the diagonal, corresponding to 9 parallel multiplications.

```
   o1 = (d + e) + (h + i);  o3 = (c + f) + (g + i);
   o5 = (b + f) + (h + i);  o7 = (a + e) + (g + i);
```

The final factorization step requires 10 additions, where (h+i) and (g+i) are common subexpressions). The parentheses are for common subexpressions and to balance the addition trees (+ is left associative).

```
   res[0*step] = o0; res[2*step] = o2;
   res[1*step] = o1; res[3*step] = o3;
   res[4*step] = o4; res[6*step] = o6;
   res[5*step] = o5; res[7*step] = o7;
```

The final stage generates the outputs. This algorithm can be unrolled. However, the long latency of the addition steps in stages 2 and 4 impacts the performance. The authors reduce the number of operations at the expense of available parallelism. This is not optimal for TriMedia. For example, using this algorithm, an 8×8 floating point DCT requires 270 cycles as compared to 230 cycles with a more parallel version.

TriMedia Implementation of an 8 x 8 DCT

An 8×8 DCT can be derived from the 8-point 1D DCT explained previously. Code to compute the DCT is shown.

```
void
dct8x8fix( long * restrict tab, long * restrict res ){
   int  r00, r02, r04, r06, r10, r12, r14, r16;
   int  r20, r22, r24, r26, r30, r32, r34, r36;
   int  r40, r42, r44, r46, r50, r52, r54, r56;
   int  r60, r62, r64, r66, r70, r72, r74, r76;
```

For efficiency, the TriMedia algorithm produces a result that is in transposed and shuffled order as shown in Figure 60.

0	32	8	40	16	48	24	56
1	33	9	41	17	49	25	57
2	34	10	42	18	50	26	58
3	35	11	43	19	51	27	59
4	36	12	44	20	52	28	60
5	37	13	45	21	53	29	61
6	38	14	46	22	54	30	62
7	39	15	47	23	55	31	63

Figure 60 DCT Reordering Matrix

A temporary copy of the matrix is necessary because the algorithm is separable. The TriMedia architecture allows this to be stored in registers.

The DCT is 16-bit. TriMedia stores two values per register.

```
int tmp0, tmp1, tmp2, tmp3, tmp101, tmp132, tmp176, tmp145, tmp201, tmp232;
```

These variables are temporaries for the 1D DCT.

```
#pragma TCS_no_caller_save
```

This ensures that only callee-saved registers (r9 to r32) are used. Caller-saved registers (r33 to r63) must be saved and restored on function entry and exit, increasing the number instruction cycles.

```
   int s0, s1, s2, s3, s4, s5, s6, s7;
   int t0, t1, t2, t3, t4, t5, t6, t7;
```

These two arrays hold two row from the horizontal DCT calculation (**horiz_dct**).

```
   horiz_dct( &tab[0], s0, s1, s2, s3, s4, s5, s6, s7 );
   horiz_dct( &tab[4], t0, t1, t2, t3, t4, t5, t6, t7 );
```

The DCT of rows 0 and 1 is computed in s_{0-7} and t_{0-7}, respectively.

```
   packltor( r00, r02, r04, r06, r10, r12, r14, r16 );
```


The DCT for row 0 is packed into the upper sixteen bits of r_{00} through r_{16}. Row 1 is packed into the lower 16 bits. The results must be packed so that values can be computed in parallel in the upcoming vertical pass.

```
    horiz_dct(&tab[ 8], t0, t1, t2, t3, t4, t5, t6, t7);
    horiz_dct(&tab[12], s0, s1, s2, s3, s4, s5, s6, s7);
    packltor( r20, r22, r24, r26, r30, r32, r34, r36 );
```

The order of packing for rows (2, 3) is reversed. Doing this saves a reversal step in the vertical stage.

```
    horiz_dct(&tab[16], t0, t1, t2, t3, t4, t5, t6, t7);
    horiz_dct(&tab[20], s0, s1, s2, s3, s4, s5, s6, s7);
    packltor( r40, r42, r44, r46, r50, r52, r54, r56 );

    horiz_dct(&tab[24], t0, t1, t2, t3, t4, t5, t6, t7);
    horiz_dct(&tab[28], s0, s1, s2, s3, s4, s5, s6, s7);
    packltor( r60, r62, r64, r66, r70, r72, r74, r76 );
```

The computations for rows 4-7 follows the same pattern.

```
    vertical_dct(&res[ 0], r00, r20, r40, r60);
    vertical_dct(&res[ 4], r02, r22, r42, r62);
    vertical_dct(&res[ 8], r04, r24, r44, r64);
    vertical_dct(&res[12], r06, r26, r46, r66);
    vertical_dct(&res[16], r10, r30, r50, r70);
    vertical_dct(&res[20], r12, r32, r52, r72);
    vertical_dct(&res[24], r14, r34, r54, r74);
    vertical_dct(&res[28], r16, r36, r56, r76);
}
```

The vertical DCT macro calculates the result for a single row of the output. Note that the output is transposed.

Coefficients and Rounding

The table below shows the coefficients used in the DCT. These are for a big-endian implementation. For a little-endian implementation, the coefficients need to be swapped. The hex values are coded as 16-bit pairs.

	Value (hex)	Upper 16 bits	Lower 16 bits
C0	0xA73B4B42	$-\cos(3\pi/16) - \sin(5\pi/16)$	$\cos(\pi/16) + \sin(\pi/16)$
C1	0x11A8CDB7	$\cos(3\pi/16) - \sin(3\pi/16)$	$\sin(\pi/16) - \cos(\pi/16)$
C2	0xCDB7A73B	$-\sqrt{2}\sin(3\pi/16.)$	$-\sqrt{2}\cos(\pi/16)$
C3	0x4B42EE58	$\sqrt{2}\cos(3\pi/16)$	$-\sqrt{2}\sin(\pi/16)$
C4	0x4B4211A8	$\sqrt{2}\cos(3\pi/16)$	$\sqrt{2}\sin(\pi/16)$
C5	0x3249A73B	$\sqrt{2}\sin(3\pi/16)$	$-\sqrt{2}\cos(\pi/16)$
C6	0x11A83249	$\cos(3\pi/16) - \sin(3\pi/16)$	$\cos(\pi/16) - \sin(\pi/16)$
C7	0x58C54B42	$\cos(3\pi/16) + \sin(3\pi/16)$	$\cos(\pi/16) + \sin(\pi/16)$

These values correspond to the odd part of the 1D DCT computation above. By application of the identity $\sin(\theta) = \cos(\pi/4 - \theta)$, the sines can be replaced by cosines.

	Value (hex)	Upper 16 bits	Lower 16 bits
C8	0x40004000	1	1
C9	0x4000C000	1	–1
C10	0x539E22A3	$\sqrt{2}\cos(\pi/8)$	$\sqrt{2}\cos(3\pi/8)$
C11	0x22A3AC62	$\sqrt{2}\cos(3\pi/8)$	$-\sqrt{2}\cos(\pi/8)$

These values correspond to the odd part. Note that the constants for C8 and C9 correspond to addition and subtraction of the upper and lower halves of a word, respectively.

This is a fixed point DCT so rounding macros are necessary. For a floating point DCT, the IEEE rounding mode must be set.

```
#define HROUND(x)  ((x) + (x) + 0x8000)
```

For packing purposes, the result need to be in the upper 16 bits. For the horizontal calculation, this corresponds to shifting the result left 1 bit and adding one half (0.5). Addition is used instead of a shift because there are more functional units (5 versus 2).

```
#define VROUND(x)  ((x) + 0x8000)
```

For the vertical calculation, the result is directly available in the upper 16 bits.

Horizontal DCT

A flow diagram of the algorithm used to compute the 1D DCT is shown in Figure 61.

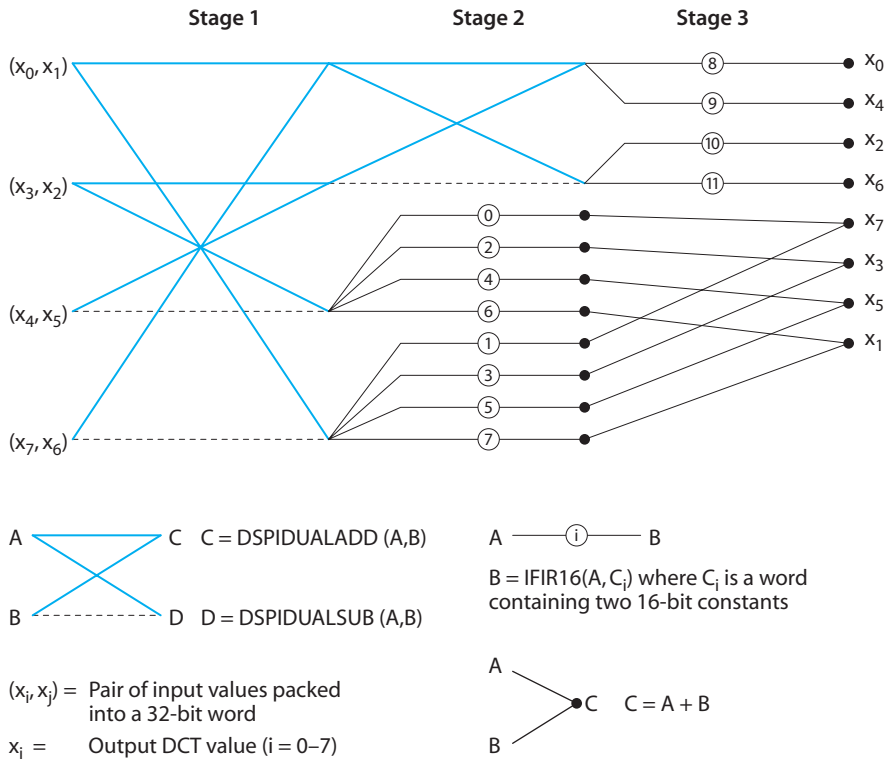

Figure 61 ID DCT Data Flow Diagram

Note that the input is in normal order and the output is in shuffled order. The input and output data are coded in 16 bits.

```
#define horiz_dct(tab, o0, o1, o2, o3, o4, o5, o6, o7)          \
        tmp0 = (tab)[0];                                        \
        tmp1 = ROLI(16, (tab)[1]);                              \
        tmp2 = (tab)[2];                                        \
        tmp3 = ROLI(16, (tab)[3]);                              \
```

First the values are read from memory in pairs. Note that (x3, x2) and (x7, x6) are read in reverse order.

```
        tmp101 = DSPIDUALADD(tmp0,tmp3);                         \
        tmp132 = DSPIDUALADD(tmp1,tmp2);                         \
        tmp176 = DSPIDUALSUB(tmp0,tmp3);                         \
        tmp145 = DSPIDUALSUB(tmp1,tmp2);                         \
```

The 2 **DSPIDUALADD**s correspond to the 4 additions of stage 1 of the algorithm explained previously. The **DSPIDUALSUB** corresponds to the subtractions.

```
        tmp201 = DSPIDUALADD(tmp101,tmp132);                    \
        tmp232 = DSPIDUALSUB(tmp101,tmp132);                    \
```

The above correspond to the two additions and two subtractions of stage 2 (even part).

```
        o0 = IFIR16(tmp201,C8);                                 \
        o4 = IFIR16(tmp201,C9);                                 \
        o2 = IFIR16(tmp232,C10);                                \
        o6 = IFIR16(tmp232,C11);                                \
```

The above correspond to the four additions and four multiplications of stage 3 (even part).

```
        o7 = IFIR16(tmp145,C0) + IFIR16(tmp176,C1);             \
        o3 = IFIR16(tmp145,C2) + IFIR16(tmp176,C3);             \
        o5 = IFIR16(tmp145,C4) + IFIR16(tmp176,C5);             \
        o1 = IFIR16(tmp145,C6) + IFIR16(tmp176,C7);
```

The above correspond to the odd part of the DCT.

The long latency of the trees of additions in stages 2 and 4 has been eliminated by introducing common sub expressions. However, the coefficient values are only represented in 16 bits.

One bit is used to represent the sign, one bit to represent the integer part, and 14 bits to represent the fractional part of the coefficient. This representation is called Q.14 format. The algorithm upscales the data by a factor of 8.

Vertical DCT

Code for the horizontal DCT computation is shown below.

```
#define vertical_dct(res, tmp0, tmp1, tmp2, tmp3)               \
                                                                \
        tmp101 = DSPIDUALADD(tmp0,tmp3);                        \
        tmp132 = DSPIDUALADD(tmp1,tmp2);                        \
        tmp176 = DSPIDUALSUB(tmp0,tmp3);                        \
        tmp145 = DSPIDUALSUB(tmp1,tmp2);                        \
```

This corresponds to the first stage as explained previously. Reversal is not necessary in the vertical stage.

```
        tmp201 = DSPIDUALADD(tmp101,tmp132);                    \
        tmp232 = DSPIDUALSUB(tmp101,tmp132);                    \
                                                                \
        s0 = IFIR16(tmp201,C8 );                                \
        s4 = IFIR16(tmp201,C9 );                                \
        s2 = IFIR16(tmp232,C10);                                \
        s6 = IFIR16(tmp232,C11);                                \
                                                                \
        s7 = IFIR16(tmp145,C0) + IFIR16(tmp176,C1);             \
        s3 = IFIR16(tmp145,C2) + IFIR16(tmp176,C3);             \
        s5 = IFIR16(tmp145,C4) + IFIR16(tmp176,C5);             \
        s1 = IFIR16(tmp145,C6) + IFIR16(tmp176,C7);             \
```


The rest of the computation is similar to the vertical DCT explained previously.

```
        (res)[0] = PACK16MSB(VROUND(s0), VROUND(s1));          \
        (res)[1] = PACK16MSB(VROUND(s2), VROUND(s3));          \
        (res)[2] = PACK16MSB(VROUND(s4), VROUND(s5));          \
        (res)[3] = PACK16MSB(VROUND(s6), VROUND(s7));
```

Note that rounding is different for the horizontal step. The result of the algorithm is the transpose of the actual DCT.

Packing

Code to pack two output lines between the horizontal and vertical calculations is shown.

```
#define packltor(o0, o1, o2, o3, o4, o5, o6, o7)               \
        o0 = PACK16MSB(HROUND(s0), HROUND(t0));                \
        o1 = PACK16MSB(HROUND(s1), HROUND(t1));                \
        ...                                                    \
        o7 = PACK16MSB(HROUND(s7), HROUND(t7));
```

The same macro is used to pack in reverse order by reversing the order of computation of the s_n and t_n arrays in the code above.

Computation of the Inverse DCT

Figure 62 shows the algorithm to calculate the inverse DCT.

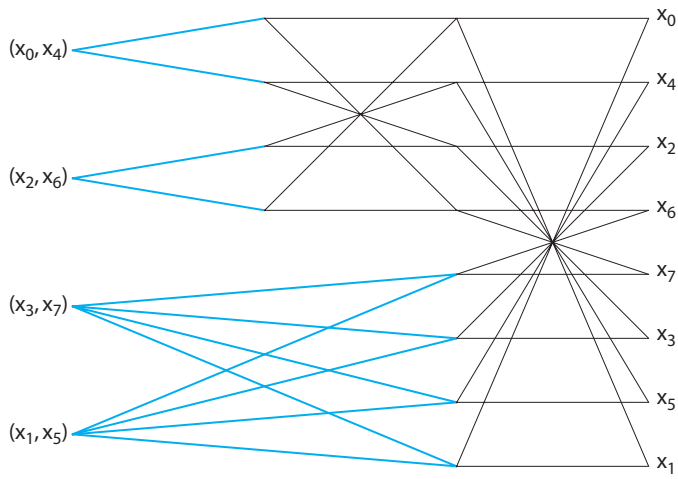

Figure 62 Inverse DCT Algorithm

The structure is similar except that rounding is applied to the input instead of the output (data flow is reversed).

Code that calculates an 8×8 DCT is shown below.

```
void do_idct( long *datain, long *dataout ){
   int   r00, r01, r02, r03;
   int   r10, r11, r12, r13;
   int   r20, r21, r22, r23;
   int   r30, r31, r32, r33;
   int   r40, r41, r42, r43;
   int   r50, r51, r52, r53;
   int   r60, r61, r62, r63;
   int   r70, r71, r72, r73;
```

These variables hold the results of pass 1 (an 8×8 matrix).

```
   int tmp0,  tmp1,  tmp2,  tmp3;
   int tmp10, tmp11, tmp12, tmp13;
   int tmp20, tmp21, tmp22, tmp23;
```

These temporaries are for computation of the DCT of one half of the input (rows 0, 1, 2, 3).

```
   int temp0,  temp1,  temp2,  temp3;
   int temp10, temp11, temp12, temp13;
   int temp20, temp21, temp22, temp23;
```

These are for computation of the other half (rows 4, 5, 6, 7).

```
   int z0,  z1,  z2,  z3,  z4,  z5,  z6, z7;
   int zz0, zz1, zz2, zz3, zz4, zz5;
```

These are intermediate temporaries.

```
   #pragma TCS_no_caller_save.
   horiz_idct(datain, 0, r00, r01, r02, r03, r40, r41, r42, r43, H_ROUNDING);
```

The first parameter points to the input. The second gives the offset into the input (row 0 in this case). The macro calculates the result for two rows offset by 4 in number. The rows of the input are required to be in shuffled order (0, 4, 2, 6, 3, 7, 1, 5). The last parameter corresponds to a 0.5 rounding factor.

```
   horiz_idct(datain, 4, r10, r11, r12, r13, r50, r51, r52, r53, 0);
```

This calculates the IDCT for rows 1 (offset 4) and 5, respectively. For the IDCT, the rounding factor only needs to be added in once. Here the result is being computed into $r_{10..13}$ and $r_{50..53}$. The upper and lower 16 bits alternate.

```
   horiz_idct( datain,  8, r20, r21, r22, r23, r60, r61, r62, r63, 0 );
   horiz_idct( datain, 12, r30, r31, r32, r33, r70, r71, r72, r73, 0 );
```

These instructions calculate 1D IDCTs for row pairs (2, 6) and (5, 7). An input of the second pass is a column of the output of the first pass.

```
   vertical_idct(r00, r10, r20, r30, dataout[0], dataout[1],
      dataout[2], dataout[3]);
```

This calculates the inverse DCT into $dataout_{0..3}$. Note that the result is transposed.

```
   vertical_idct( r01, r11, r21, r31, dataout[4], dataout[5],
      dataout[6], dataout[7] );
   ...
```



```
    vertical_idct( r43, r53, r63, r73, dataout[28], dataout[29],
       dataout[30], dataout[31] );
}
```

This calculates the inverse DCT for columns 1 to 7. Note that the algorithm is completely unrolled so there are no branches. For the IDCT, the input data is assumed to be transposed and shuffled as defined by Figure 62 on page 97. It is also assumed to be upscaled by eight.

Coefficients

The table below shows the coefficient values for the inverse DCT. Values for a big-endian implementation are given. For a little-endian implementation, the coefficients need to be swapped.

Note that $MASK_{1-4}$ in the inverse DCT correspond to the C_{8-11} coefficients in the DCT algorithm.

	Value (hex)	Upper 16 bits	Lower 16 bits
D0	0x18f96A6E	$\cos(3\pi/16) + \cos(5\pi/16)$	$\cos(\pi/16) + \cos(7\pi/16)$
D1	0xB8E38276	$\cos(7\pi/16) - \cos(\pi/16)$	$\cos(3\pi/16) + \cos(5\pi/16)$
D2	0x471D18F9	$\cos(7\pi/16) - \cos(\pi/16)$	$\cos(3\pi/16) - \cos(5\pi/16)$
D3	0x82766A6E	$\cos(3\pi/16) + \cos(5\pi/16)$	$\cos(\pi/16) + \cos(7\pi/16)$
D4	0x6A6E8276	$\cos(\pi/16) + \cos(7\pi/16)$	$\cos(3\pi/16) + \cos(5\pi/16)$
D5	0xE707B8E3	$\cos(3\pi/16) - \cos(5\pi/16)$	$\cos(7\pi/16) - \cos(\pi/16)$
D6	0x7D8A471D	$\cos(3\pi/16) + \cos(5\pi/16)$	$\cos(7\pi/16) - \cos(\pi/16)$
D7	0x6A6E18F9	$\cos(\pi/16) + \cos(7\pi/16)$	$\cos(3\pi/16) - \cos(5\pi/16)$
MASK1	0x30FC89BE	$\cos(2\pi/16) - \cos(6\pi/16)$	$\cos(6\pi/16) + \cos(2\pi/16)$
MASK2	0x76423 0FC	$\cos(6\pi/16) + \cos(2\pi/16)$	$\cos(2\pi/16) - \cos(6\pi/16)$
MASK3	0x5A835A82	1	1
MASK4	0x5A82A57E	1	–1

To maximize the precision of the result, the values shown have been scaled by $\sqrt{2}$.

Constants

Constant values used in the algorithm are shown below.

```
#define SCALED_COEFFS           1
#define PASS1_BITS              1
#define CONST_BITS2             14
```

SCALED_COEFFS corresponds to the scaling of the coefficients by $\sqrt{2}$. Two DCTs are applied so this corresponds to shifting the result left one bit. **PASS1_BITS** is the output

position of the binary point after the computation of the first pass. **CONST_BITS2** equals the precision of the coefficients.

```
#define B2                      CONST_BITS+PASS1_BITS+3+SCALED_COEFFS
#define DCT_SHIFT_1100_INTER    ((B2) - 16)
```

The down shift factor for the result is the sum of these values. The value of 3 compensates for the fact that the output is 8 times greater than that of the DCT algorithm. This corresponds to the scaling by $\sqrt{N}$ in Loeffler's algorithm as explained previously.

DCT_SHIFT_1100_INTER is the factor needed to obtain a result in the upper 16 bits.

```
#define TMP_20_21_H_BIAS        0x8000
#define H_ROUNDING              (32 << (16*!LITTLE_ENDIAN))
```

TMP_201_H_BIAS is the rounding factor for the vertical DCT (result in the upper 16 bits). **H_ROUNDING** is for the horizontal DCT. 32/64 is 0.5. The denominator in the above corresponds to an 8 x 8 matrix. This needs to be added to the first sample. This is either the upper or the lower 16 bits, depending on the endianness.

Endianness

```
#define PACK16_MSB(a, b)        PACK16MSB(a, b)
```

This macro is used to pack the results. For a little-endian computation, the following macro should be used.

```
#define PACK16_MSB(a, b)        PACK16MSB(b, a)
```

Horizontal Inverse DCT

The macro **horiz_idct** computes two rows of a horizontal DCT. The definition is shown.

```
#define horiz_idct(data, offset, r0, r1, r2, r3, r4, r5, r6, r7, comp)   \
   z2 = data[offset + 1];                                                \
   z3 = data[offset + 3];                                                \
   z5 = data[offset + 2];                                                \
   z0 = data[offset    ] + comp;                                         \
```

This reads the first row (rows 0, 1, 2, 3). The rounding factor is added to the first input.

```
   zz5 = data[offset + 18];                                              \
   zz2 = data[offset + 17];                                              \
   zz3 = data[offset + 19];                                              \
   zz0 = data[offset + 16];                                              \
```

The second row is offset by 4 from the first. This corresponds to the shuffled order (0, 4, 1, 5, 2, 6, 3, 7).

```
   tmp22 = IFIR16(z5, MASK1);                                            \
   tmp23 = IFIR16(z5, MASK2);                                            \
   tmp20 = IFIR16(z0, MASK3) + TMP_20_21_H_BIAS;                         \
   tmp21 = IFIR16(z0, MASK4) + TMP_20_21_H_BIAS;                         \
```


This calculates the even part of the DCT. The result is rounded for input to the second pass.

```
    tmp10 = tmp20 + tmp23;                                          \
    tmp13 = tmp20 - tmp23;                                          \
    tmp11 = tmp21 + tmp22;                                          \
    tmp12 = tmp21 - tmp22;                                          \

    tmp0 = IFIR16(z2, D0) + IFIR16(z3, D1);                         \
    tmp1 = IFIR16(z2, D2) + IFIR16(z3, D3);                         \
    tmp2 = IFIR16(z2, D4) + IFIR16(z3, D5);                         \
    tmp3 = IFIR16(z2, D6) + IFIR16(z3, D7);                         \
```

This corresponds to the odd part.

```
    temp22 = IFIR16(zz5, MASK1);                                    \
    temp23 = IFIR16(zz5, MASK2);                                    \
    temp20 = IFIR16(zz0, MASK3) + TMP_20_21_H_BIAS;                 \
    temp21 = IFIR16(zz0, MASK4) + TMP_20_21_H_BIAS;                 \
    temp10 = temp20 + temp23;                                       \
    temp13 = temp20 - temp23;                                       \
    temp11 = temp21 + temp22;                                       \
    temp12 = temp21 - temp22;                                       \
    temp0 = IFIR16(zz2, D0) + IFIR16(zz3, D1);                      \
    temp1 = IFIR16(zz2, D2) + IFIR16(zz3, D3);                      \
    temp2 = IFIR16(zz2, D4) + IFIR16(zz3, D5);                      \
    temp3 = IFIR16(zz2, D6) + IFIR16(zz3, D7);                      \
```

The code is the same for the second row (rows 4, 5, 6, 7).

```
    r0 =  PACK16_MSB(tmp10 + tmp3, temp10 + temp3);                 \
    r1 =  PACK16_MSB(tmp11 + tmp2, temp11 + temp2);                 \
    r2 =  PACK16_MSB(tmp12 + tmp1, temp12 + temp1);                 \
    r3 =  PACK16_MSB(tmp13 + tmp0, temp13 + temp0);                 \
    r4 =  PACK16_MSB(tmp13 - tmp0, temp13 - temp0);                 \
    r5 =  PACK16_MSB(tmp12 - tmp1, temp12 - temp1);                 \
    r6 =  PACK16_MSB(tmp11 - tmp2, temp11 - temp2);                 \
    r7 =  PACK16_MSB(tmp10 - tmp3, temp10 - temp3);                 \
    horiz_idct_clear(data, offset);
```

This corresponds to the butterfly computation in stage 1 of figure 2. The even numbered row is packed in the upper 16 bits. The odd numbered row is packed in the lower 16 bits.

The **horiz_idct_clear** macro zeroes two rows of the input prior to the next pass. Only non-zero coefficients are stored. The clear operation and computation can be overlapped can be overlapped.

```
#define horiz_idct_clear(data, offset)                              \
    data[offset+ 0] = 0;   data[offset+18] = 0;                     \
    data[offset+ 2] = 0;   data[offset+16] = 0;                     \
    data[offset+ 1] = 0;   data[offset+ 3] = 0;                     \
    data[offset+17] = 0;   data[offset+19] = 0;
```

Calculation of the Vertical DCT

Code to calculate a single column of the vertical DCT is shown below.

```
#define vertical_idct(r0, r1, r2, r3, dest1, dest2, dest3, dest4)          \
   tmp22 = IFIR16(r2, MASK1);                                              \
   tmp23 = IFIR16(r2, MASK2);                                              \
   tmp20 = IFIR16(r0, MASK3);                                              \
   tmp21 = IFIR16(r0, MASK4);                                              \
                                                                           \
   tmp10 = tmp20 + tmp23;                                                  \
   tmp13 = tmp20 - tmp23;                                                  \
   tmp11 = tmp21 + tmp22;                                                  \
   tmp12 = tmp21 - tmp22;                                                  \
                                                                           \
   tmp0 = IFIR16(r1, D0) + IFIR16(r3, D1);                                 \
   tmp1 = IFIR16(r1, D2) + IFIR16(r3, D3);                                 \
   tmp2 = IFIR16(r1, D4) + IFIR16(r3, D5);                                 \
   tmp3 = IFIR16(r1, D6) + IFIR16(r3, D7);                                 \
                                                                           \
   combinePred( tmp10 + tmp3, tmp11 + tmp2, tmp12 + tmp1,                  \
                tmp13 + tmp0, dest1, dest2 )                               \
   combinePred( tmp13 - tmp0, tmp12 - tmp1, tmp11 - tmp2,                  \
                tmp10 - tmp3, dest3, dest4 )
```

Rounding has been precomputed. The **combinePred** macro stores the result.

I Frames and P Frames

For the second and subsequent frames in a sequence (I frame, inter frame), the value transmitted is the DCT of the difference. A saturating add needs to be applied between the inverse DCT and the result.

The implementation of the **combinePred** macro for inter frames is shown.

The big-endian implementation of **combinePred** for an intra-frame is shown below. TM1100 instructions need to be used since IEEE-1180 conformance requires a calculation in 16 bits.

```
#define combinePred(dct3, dct2, dct1, dct0, pred1, pred2)                  \
                                                                           \
pred1 = mergedual16lsb(                                                    \
  dualuclipi(                                                              \
    dspidualadd(                                                           \
      dualiclipi(dualasr(pack16msb(dct0, dct1), DCT_SHIFT_INTER), 255),    \
      mergemsb(0, pred1)                                                   \
    ),                                                                     \
    255),                                                                  \
  dualuclipi(                                                              \
    dspidualadd(                                                           \
      dualiclipi(dualasr(pack16msb(dct2, dct3), DCT_SHIFT_INTER), 255),    \
      mergelsb(0, pred1)                                                   \
    ),                                                                     \
    255)                                                                   \
  ) ;                                                                      \
pred2 = mergedual16lsb(                                                    \
  dualuclipi(                                                              \
    dspidualadd(                                                           \
      dualiclipi(dualasr(pack16msb(dct0, dct1), DCT_SHIFT_INTER), 255),    \
```



```
      mergemsb(0, pred2)                                                    \
    ),                                                                      \
    255),                                                                   \
  dualuclipi(                                                               \
    dspidualadd(                                                            \
      dualiclipi(dualasr(pack16msb(dct2, dct3), DCT_SHIFT_INTER), 255),     \
      mergelsb(0, pred2)
    ),                                                                      \
    255)                                                                    \
  ) ;
```

The big-endian implementation of **combinePred** for inter frames stores into the result. For P frames, the computation is simpler because the result just needs to be stored.

```
#define combinePred(dct0, dct1, dct2, dct3, pred1, pred2)                   \
   pred1 = dualiclipi(dualasr(pack16msb(dct0, dct1),                        \
      DCT_SHIFT_1100_INTER), 255);                                          \
   pred2 = dualiclipi(dualasr(pack16msb(dct2, dct3),                        \
      DCT_SHIFT_1100_INTER), 255);
```

For a little-endian implementations, the order of the arguments $dct_{0–3}$ must be reversed.

Results

An 8×8 DCT can be calculated in 165 instruction cycles. Between 160 and 170 instruction cycles are necessary to compute an 8×8 inverse DCT (compute, store, clear).

The algorithm has been tested to conform to the IEEE 1180 standard. Using a variant, a floating point DCT can be calculated in between 220 and 230 instruction cycles.

IIR Filter

Introduction

This section will first discuss the implementation and optimization of a second order IIR filter for one channel with floating point arithmetic. Then it shows how to gain a higher performance by using integer arithmetic, and introduces a fourth order IIR filter consisting of two equal cascaded second order sections. Finally it briefly points out that the performance can be enhanced by processing two channels in parallel in a single function.

The example refers to the canonical form of an IIR filter as shown in Figure 63. The processed data is discrete; it is a sampled time continuous signal, generated by an A/D converter. For further details on this specific topic, refer to *Oppenheim/Schaefer Digital Signal*

Processing (Prentice-Hall). This reference also contains introductory material on IIR filters.

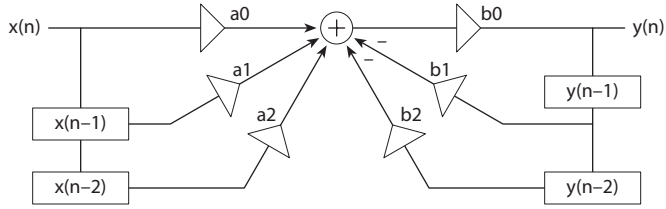

Figure 63 Canonical Structure of an IIR Filter

The filter can be described in the time domain with the following second order difference equation:

$$b_0 \times y(n) + b_1 \times y(n-1) + b_2 \times y(n-2) = a_0 \times x(n) + a_1 \times x(n-1) + a_2 \times x(n-2)$$

It is common to assume b_0 equal to 1. If not, just divide the whole equation by b_0. This leads to the following:

$$y(n) = a_0 \times x(n) + a_1 \times x(n-1) + a_2 \times x(n-2) - b_1 \times y(n-1) - b_2 \times y(n-2)$$

Where x(n) is the input and y(n) the output. A straightforward implementation using floating point arithmetic is shown in the function **iirFilter_1**.

```
a_0 = coeff[0], a_1 = coeff[1], a_2 = coeff[2],
b_1 = coeff[3], b_2 = coeff[4]
```

IMPORTANT
The algorithms and implementations presented here assume that $b_0 = 1$. If you use an array for the coefficients including b_0, you should multiply the output with b_0 before feedback or at least check if it is 1 and if not generate an error message. It is also possible to use strong typing with the struct-construct.

Includes and Macros

For filters utilizing floating point arithmetic, the following header files need to be included (no macros have been defined):

```
#include <stdlib.h>
#include <math.h>
```

For filters utilizing fixed point integer arithmetic the following header files need to be included:

```
#include <stdlib.h>
#include <math.h>
#include <ops/custom_defs.h>
```


One macro is defined. It will be explained in depth later, but it is important to be in the iirFilter_X.c file when calculating with integer arithmetic.

```
#define SHIFT 3  /* see case study on IIR filter in Cookbook */
```

Optimization for Floating Point(Second Order, One Channel)

```
int iirFilter_1( float *inputData, *float outputData,
                 float *coeff, float *state, int sampleNumber ){
   int i;

   for( i=0; i < sampleNumber; i++ ){
        outputData[i] = coeff[0]*inputData[i]
                      + coeff[1]*state[0]
                      + coeff[2]*state[1]
                      - coeff[3]*state[2]
                      - coeff[4]*state[3];

      state[1] = state[0]; state[0] = inputData [i];
      state[3] = state[2]; state[2] = outputData[i];
   }
   return 0;
}
```

It is significantly more efficient to use local variables because the coefficients and states will be stored in registers. This ensures that fewer loads and stores are necessary in the loop. Furthermore, we can use the **restrict** keyword to declare that the pointers on the input and output data are pointing to disjointed sections of the memory. It is the responsibility of the programmer to ensure that this condition is true.

```
int iirFilter_2( float *inputData, float *outputData,
                 float *coeff, float *state, int sampleNumber ){
   int i;
   float * restrict input, * restrict output;
   float coeff_a0, coeff_a1, coeff_a2, coeff_b1, coeff_b2;
   float state_0, state_1, state_2, state_3

   input = inputData; output = outputData;
   coeff_a0 = coeff[0]; coeff_a1 = coeff[1]; coeff_a2 = coeff[2];
   coeff_b1 = coeff[3]; coeff_b2 = coeff[4];
   state_0  = state[0]; state_1 = state[1];
   state_2  = state[2]; state_3 = state[3];
   for( i=0; i < sampleNumber; i++ ){
      output[i] = coeff_a0*input[i]
                + coeff_a1*state_0
                + coeff_a2*state_1
                - coeff_b1*state_2
                - coeff_b2*state_3;

      state_1 = state_0; state_0 = input [i];
      state_3 = state_2; state_2 = output[i];
   }
   state[0] = state_0; state[1] = state_1;
   state[2] = state_2; state[3] = state_3;

   return 0;
}
```

The functions have been compiled with

```
tmcc -el -O3 -o testMain.out.
```

With the **-K** option, the compiler keeps all intermediate code levels. For further details about compiling, refer to Chapter 1 of this book or Chapter 2 of *Software Tools*. The generated *.t and *.s files will be used to explain the performance improvement.

After compiling, a statfile is produced with the simulator:

```
tmsim -el -statfile testMain.stat testMain.out noiseMono.pcm
```

Finally, a performance report was generated with

```
tmprof -func -detail -scale 1.0 -threshold 0.0001
   testMain.stat > testMain.prf
```

The file noiseMono.pcm contains 36864 samples and all the functions run twice on this input. In other words, a total of 73728 samples (for each channel) has been processed at a sampling frequency of 44100 Hertz.

To compare performance, a number for cycles per sample for the second order section is used. This is obtained by taking the cycle number from the tmprof report and dividing by the number of channels, the number of biquads and the number of samples.

$$\#(\text{cycl./sample}) = \#(\text{totalCycl.}) \,/\, (\#\text{channel} \times \#\text{ sample} \times \#\text{biquad}).$$

The number of channels is 2 for iirFilter_10.c, otherwise it is 1. The number of biquads is 1, except for iirFilter_9.c and iirFilter_10.c, where it is 2.

Furthermore, the MIPS are given.

$$x\ (\text{MIPS in } \%) = 44100 \times \#\text{totalCycles} \,/\, (\,\#\text{channel} \times \#\text{sample} \times 10^6\,)$$

The total number of cycles is the sum of the instruction cycles, the instruction cache stall cycles, and the data cache stall cycles.

Table 34 Performance Table for IIR Filters 1 and 2

Treename	_iirFilter_1	_iirFilter_2
Total Cycles (Instructions + Stall)	2370888	1127256
MIPS (%)	1.418	0.674
Cycles per Sample	32.16	15.29
Instruction Cycles	2361312	1126944
I-Cache Stall Cycles	223	290
D-Cache Stall Cycles	9353	22
D-Conflict Cycles	9216	0
(Useful) Operations per Instruction. Maximum is 5	0.94(0.92)	0.97(0.92)

Looking at the *.t files for the filters, you can see that the decision tree 2 refers to an unrolled version of the main loop while decision tree 5 contains the loop only once.

This is similar for both, with the slight difference that the main loop for iirFilter_1.c is unrolled two times and for the iirFilter_2.c four times. This is due to more parallelism in the second implementation because the coefficients and states do not need to be loaded in the loop. They are already assigned to local variables, which correspond to registers.

This increase in parallelism can easily be seen in the *.t files when you compare the number of after-constraints. The file iirFilter_1.t has much more than iirFilter_2.t.

After-constraints are specified by the keyword **after** and a list of operation names. These constraints impose absolute ordering between pairs of operations. Refer to Chapter 4, *Using the Instruction Scheduler*, of *Software Tools* for more information about decision trees and decision tree features.

It can be seen that function iirFilter_1.c has 14 loads and 5 stores per loop while iirFilter_2.c has only 2 loads and 1 store. This is due to the use of local variables for the states and the coefficients - 4 variables for state, 5 for the coefficients, and 1 for the input sample. This means 12 variables have to be loaded, and 4 states plus 1 output sample need to be stored. In the second case only the input and output samples need to be loaded and stored.

Since a floating point addition is commutative but not associative, the compiler executes this in a strict left to right order. The latency of a floating point addition is three cycles. Hence, for four additions, four times three cycles are necessary. It is possible to reduce this number by introducing local variables for the intermediate results to do the adds in parallel. Now only three times three cycles are required. In other words we save three cycles per sample. It is important to note that the results produced with iirFilter_2.c and iirFilter_3.c may differ slightly. This depends upon the coefficients and/or the stability of the designed IIR filter and must be tested. In most of the cases it should not matter.

```
int iirFilter_3( float *inputData, float *outputData,
                 float *coeff, float *state, int sampleNumber ){
   int i;
   float * restrict input, * restrict output;
   float temp1, temp2, temp3, temp4, temp5, temp6, temp7, temp8;
   float coeff_a0, coeff_a1, coeff_a2, coeff_b1, coeff_b2;
   float state_0, state_1, state_2, state_3;

   input = inputData; output = outputData;
   coeff_a0 = coeff[0]; coeff_a1 = coeff[1]; coeff_a2 = coeff[2];
   coeff_b1 = coeff[3]; coeff_b2 = coeff[4];
   state_0 = state[0];  state_1 = state[1];
   state_2 = state[2];  state_3 = state[3];

   for( i=0; i < sampleNumber; i++ ){
      temp1 = coeff_a0*input[i];
      temp2 = coeff_a1*state_0;
      temp3 = coeff_a2*state_1;
      temp4 = coeff_b1*state_2;
      temp5 = coeff_b2*state_3;

      temp6 = temp1 + temp2; temp7 = temp3 - temp4;
      temp8 = temp6 - temp5;
      output[i] = temp7 + temp8;

      state_1 = state_0; state_0 = input [i];
```

```
      state_3 = state_2; state_2 = output[i];
   }
   state[0] = state_0; state[1] = state_1;
   state[2] = state_2; state[3] = state_3;

   return 0;
}
```

Table 35 Performance Table for IIR Filter 3

Treename	_iirFilter_3
Total Cycles (Instructions + Stall)	1071960
MIPS (%)	0.641
Cycles per Sample	14.54
Instruction Cycles	1071648
I-Cache Stall Cycles	319
D-Cache Stall Cycles	22
D-Conflict Cycles	0
(Useful) Operations per Instruction. Maximum is 5	1.08(1.06)

At the end of a decision tree, there is always a jump operation either to another tree or to the tree itself. The jump occurs three cycles later. This means that after a jump instruction only operations which require less than two cycles can be issued. However, delay slot cycles are often unused. Regarding the loop in the IIR filter, we can load the input data once outside the loop, store it in a register and then load the next data at the end of the loop using the branch delay slots. This technique is also referred to as *software pipelining*.

This effect would be much more visible if we would load more variables in one loop. This means for example processing two channels at a time, as discussed later.

```
int iirFilter_4( float *inputData, float *outputData,
                 float *coeff, float *state, int sampleNumber ){
   int i;
   float * restrict input, * restrict output;
   float temp1, temp2, temp3, temp4, temp5, temp6, temp7, temp8;
   float coeff_a0, coeff_a1, coeff_a2, coeff_b1, coeff_b2;
   float state_0, state_1, state_2, state_3;
   float inSample1, outSample1;

   input = inputData; output = outputData;
   coeff_a0 = coeff[0]; coeff_a1 = coeff[1]; coeff_a2 = coeff[2];
   coeff_b1 = coeff[3]; coeff_b2 = coeff[4];
   state_0 = state[0]; state_1 = state[1];
   state_2 = state[2]; state_3 = state[3];

   inSample1 = input[0];

   for( i=0; i < sampleNumber; i++ ){
      temp1 = coeff_a0*inSample1;
      temp2 = coeff_a1*state_0;
      temp3 = coeff_a2*state_1;
```



```
      temp4 = coeff_b1*state_2;
      temp5 = coeff_b2*state_3;

      temp6 = temp1 + temp2; temp7 = temp3 - temp4;
      temp8 = temp6 - temp5;
      outSample1 = temp7 + temp8;
      output[i] = outSample1;

      state_1 = state_0; state_0 = inSample1;
      state_3 = state_2; state_2 = output[i];

      inSample1 = input[i+1];
   }
   state[0] = state_0; state[1] = state_1;
   state[2] = state_2; state[3] = state_3;

   return 0;
}
```

Table 36 Performance Table for IIR Filter 4

Treename	_iirFilter_4
Total Cycles (Instructions + Stall)	1016693
MIPS (%)	0.608
Cycles per Sample	13.79
Instruction Cycles	1016352
I-Cache Stall Cycles	319
D-Cache Stall Cycles	22
D-Conflict Cycles	0
(Useful) Operations per Instruction. Maximum is 5	1.08(1.06)

As mentioned previously, the compiler performs loop unrolling automatically. There are several methods to tune this with flags for the compiler. Note that this will affect all the loops in the application. You can also use pragmas to tune unrolling for each function. However, it can be shown that loop unrolling by hand sometimes generates slightly more efficient code.

Considering a continuous data stream, we can assume that there are always enough samples to fill a buffer. Suppose the user of the processing function knows about the n-times loop unrolling. It is easy for the user to provide a buffer length that is a multiple of n. Hence, no testing for the loop counter is necessary. This means also that loop unrolling by hand causes a decrease in code size compared to the loop unrolling generated by the compiler, because there is no extra decision tree for the not-unrolled loop.

It is important to switch off loop unrolling for those loops, because sometimes the compiler tries to unroll them and this might result in inefficient code. This can be easily done by using the following pragma right before the loop as shown in iirFilter_5.c.

```
#pragma TCS_unroll=0
```

This can be used to separately tune loop unrolling for each loop in the whole application. Refer to Chapter 2 of *Software Tools* for details.

```
int iirFilter_5( float *inputData, float *outputData,
                 float *coeff, float *state, int sampleNumber ){
   #pragma TCS_unroll=0

   int i;
   float * restrict input, * restrict output;
   float temp1, temp2, temp3, temp4, temp5, temp6, temp7, temp8;
   float coeff_a0, coeff_a1, coeff_a2, coeff_b1, coeff_b2;
   float state_0, state_1, state_2, state_3;
   float inSample1, inSample2, inSample3, inSample4;
   float outSample1, outSample2, outSample3, outSample4;

   input = inputData; output = outputData;
   coeff_a0 = coeff[0]; coeff_a1 = coeff[1]; coeff_a2 = coeff[2];
   coeff_b1 = coeff[3]; coeff_b2 = coeff[4];
   state_0  = state[0]; state_1  = state[1];
   state_2  = state[2]; state_3  = state[3];

   inSample1 = input[0]; inSample2 = input[1];
   inSample3 = input[0]; inSample4 = input[1];

   for( i=0; i < sampleNumber; i+=4 ){
      temp1 = coeff_a0*inSample1;
      temp2 = coeff_a1*state_0; temp3 = coeff_a2*state_1;
      temp4 = coeff_b1*state_2; temp5 = coeff_b2*state_3;

      temp6 = temp1 + temp2; temp7 = temp3 - temp4;
      temp8 = temp6 - temp5;
      outSample1 = temp7 + temp8;
      output[i] = outSample1;

      temp1 = coeff_a0*inSample2;
      temp2 = coeff_a1*inSample1;  temp3 = coeff_a2*state_0;
      temp4 = coeff_b1*outSample1; temp5 = coeff_b2*state_2;

      temp6 = temp1 + temp2; temp7 = temp3 - temp4;
      temp8 = temp6 - temp5;
      outSample2 = temp7 + temp8;
      output[i+1] = outSample2;

      temp1 = coeff_a0*inSample3;
      temp2 = coeff_a1*inSample2;  temp3 = coeff_a2*inSample1;
      temp4 = coeff_b1*outSample2; temp5 = coeff_b2*outSample1;

      temp6 = temp1 + temp2; temp7 = temp3 - temp4;
      temp8 = temp6 - temp5;
      outSample3 = temp7 + temp8;
        output[i+2] = outSample3;

      temp1 = coeff_a0*inSample4;
      temp2 = coeff_a1*inSample3;  temp3 = coeff_a2*inSample2;
      temp4 = coeff_b1*outSample3; temp5 = coeff_b2*outSample2;

      temp6 = temp1 + temp2; temp7 = temp3 - temp4;
      temp8 = temp6 - temp5;
      outSample4 = temp7 + temp8;
      output[i+3] = outSample4;

      state_0 = inSample4;  state_1 = inSample3;
```



```
      state_2 = outSample4; state_3 = outSample3;

      inSample1 = input[i+4]; inSample2 = input[i+5];
      inSample3 = input[i+6]; inSample4 = input[i+7];
   }
   state[0] = state_0; state[1] = state_1;
   state[2] = state_2; state[3] = state_3;

   return 0;
}
```

Table 37 Performance Table for IIR Filter 5

Treename	_iirFilter_5
Total Cycles (Instructions + Stall)	739291
MIPS (%)	0.442
Cycles per Sample	10.03
Instruction Cycles	739008
I-Cache Stall Cycles	261
D-Cache Stall Cycles	22
D-Conflict Cycles	0
(Useful) Operations per Instruction. Maximum is 5	1.40(1.38)

This improvement would also be more visible, if we had a longer loop body, especially when calculating more than one channel at a time.

Looking in detail, there are 25 local variables plus the pointer for the input and output data. Even if they are assigned to one register each for the whole loop, we would not run into the problem of register spills, because there are more registers available. If we had more local variables than registers at a certain time, loads and stores would be introduced by the scheduler. This would decrease the performance. Since the compiler uses the registers dynamically (which means that it frees registers in the loop which are no longer required), we could also use more local variables. However, when running into register spills the compiler generates a warning.

When you look at the number for operations per issue slots (maximum is five out of five) you recognize that it has been increased. But it is important to realize that a high number does not necessarily mean that the code is efficient. Otherwise, assuming a high performance, a number close to 5 indicates that there is not much more parallelism to gain.

Optimization for Fixed Point Integer (Second Order, One Channel)

Using fixed point integer arithmetic in the first unoptimized version (iirFilter_1.c) leads to iirFilter_6.c. It has slightly better performance than iirFilte_1.c.

```
int iirFilter_6( int *inputData, int *outputData,
                 int *coeff, int *state, int sampleNumber ){
   int i;
```

```
   for( i=0; i < sampleNumber; i++ ){
      outputData[i] = (   IMULM( coeff[0], inputData[i] )
                        + IMULM( coeff[1], state[0] )
                        + IMULM( coeff[2], state[1] )
                        - IMULM( coeff[3], state[2] )
                        - IMULM( coeff[4], state[3] ) ) << SHIFT;

      state[1] = state[0]; state[0] = inputData [i];
      state[3] = state[2]; state[2] = outputData[i];
   }

   return 0;
}
```

The values for data, states, and coefficients are represented in a fixed point notation. For example, 1.31 refers to a floating point signal in the range of –1 to $1–2^{-31}$,which means that the size of one quantization step is 2^{-31}. Refer to *Fixed-Point Arithmetic* on page 68 for more information.

The input in this case is assumed to be 2.30. For converting the coefficients from floating to fixed point, a function **convertCoeff()** in convertCoeff.c is used. It assumes that the coefficients are in a range of –2 to $2–2^{-30}$ and multiplies them by 2^{29}. Hence, the maximum values of the coefficients are still in the range of a 32-bit signed integer. The coefficients used here refer to a Linkwitz-Riley Crossover Filter consisting of two cascaded second order IIR filters(Butterworth).

The consideration of the number representation for the input data and the coefficients depends on the designed filter. You will need some headroom, if the filter applies a gain. In addition, intermediate values may cause overflow since there are additions and subtractions. These are things that need to be investigated, or at least tested. In this case we use input data in a 2.30 format and coefficients in a 3.29 format, as mentioned before.

Another point is that these filters are often used in audio signal processing. Since common data formats are below 1.31 or 2.30 (maximum is 1.23), it becomes necessary to shift the data up and down. This is not a bottleneck, because the TriMedia has two shifters which can operate in parallel and require only one cycle. It is worth considering integrating this shift into the filter and passing it as a parameter when the function is called. This could provide an almost zero overhead shift.

Furthermore, because of the feedback in the loop it becomes necessary to left-shift the output data to get the same representation as the input data. This shift depends only on the representation of the coefficients. Assume the input data is Q.n and the coefficients are P.m, then the intern result is 64 bits in Q+P.n+m representation. The IMULM custom operator gives us the upper 32 bits. Hence, the output data has a Q+P.n-P representation, and we have to do a left shift of P to get the initial representation of Q.n again. In the examples this shift is 3, because the representation of the coefficient is 3.29, P equals 3. This shift causes the last 3 bits to be zero. To prevent artifacts with low level signals, dither techniques can be used. A very simple solution for this could be copying the last three bits of the unshifted value to the ones that are zero after shifting.

It might for example look like this, but further discussion and optimization is beyond the scope of this case study.

```
#define DITHERMASK 7   /* 0x00000007 */
temp = temp1 + temp2 + temp3 - temp4 - temp5;
outSample = ( temp << ( WORDLENGTH - COEFFSHIFT ) ) | ( temp & DITHERMASK );
```

When using local variables for the states and coefficients in the integer version, the performance is better than the best that was obtained using floating point. This is due to the fact that there are five integer ALUs but only two for floating point. Also an addition or subtraction in integer has a latency of only one cycle, while for floating point this is 3 cycles. This also causes the shift overhead to be virtually non-existent.

As integer addition is both commutative and associative, we would not need to introduce more local variables for performance reasons. Nevertheless, it is introduced here for better readability of the code.

```
int iirFilter_7( int *inputData, int *outputData,
                 int *coeff, int *state, int sampleNumber ){
   int i;
   int * restrict input, * restrict output;
   int temp1,    temp2,    temp3,    temp4,    temp5;
   int coeff_a0, coeff_a1, coeff_a2, coeff_b1, coeff_b2;
   int state_0,  state_1,  state_2,  state_3;

   input = inputData; output = outputData;
   coeff_a0 = coeff[0]; coeff_a1 = coeff[1]; coeff_a2 = coeff[2];
   coeff_b1 = coeff[3]; coeff_b2 = coeff[4];
   state_0  = state[0]; state_1  = state[1];
   state_2  = state[2]; state_3  = state[3];

   for( i=0; i < sampleNumber; i++ ){
      temp1 = IMULM( coeff_a0, input[i] );
      temp2 = IMULM( coeff_a1, state_0 );
      temp3 = IMULM( coeff_a2, state_1 );
      temp4 = IMULM( coeff_b1, state_2 );
      temp5 = IMULM( coeff_b2, state_3 );

      output[i] = (temp1 + temp2 + temp3 - temp4 - temp5) << SHIFT;

      state_1 = state_0; state_0 = input[i];
      state_3 = state_2; state_2 = output[i];
   }
   state[0] = state_0; state[1] = state_1;
   state[2] = state_2; state[3] = state_3;

   return 0;
}
```

Table 38 Performance Table for IIR Filters 6 and 7

Treename	_iirFilter_6	_iirFilter_7
Total Cycles (Instructions + Stall)	1854857	813864
MIPS (%)	1.109	0.487
Cycles per Sample	25.16	11.04
Instruction Cycles	1845216	813600

Table 38 Performance Table for IIR Filters 6 and 7

Treename	_iirFilter_6	_iirFilter_7
I-Cache Stall Cycles	203	232
D-Cache Stall Cycles	9438	32
D-Conflict Cycles	9234	2
(Useful) Operations per Instruction. Maximum is 5	1.24(1.22)	1.59(1.55)

For performance comparison, iirFilter_8.c gives an unrolled version of the integer IIR filter. The input data is loaded once before the loop and then at the end, as discussed previously.

```
int iirFilter_8( int *inputData, int *outputData,
                 int *coeff, int *state, int sampleNumber ){
   #pragma TCS_unroll=0

   int i;
   int * restrict input, * restrict output;
   int temp6,  temp7,  temp8,  temp9,  temp10;
   int temp11, temp12, temp13, temp14, temp15;
   int temp16, temp17, temp18, temp19, temp20;
   int coeff_a0, coeff_a1, coeff_a2, coeff_b1, coeff_b2;
   int state_0,  state_1,  state_2,  state_3;
   int inSample1,  inSample2,  inSample3,  inSample4;
   int outSample1, outSample2, outSample3, outSample4;

   input = inputData; output = outputData;
   coeff_a0 = coeff[0]; coeff_a1 = coeff[1]; coeff_a2 = coeff[2];
   coeff_b1 = coeff[3]; coeff_b2 = coeff[4];
   state_0  = state[0]; state_1  = state[1];
   state_2  = state[2]; state_3  = state[3];

   inSample1 = input[0]; inSample2 = input[1];
   inSample3 = input[2]; inSample4 = input[3];

   for( i=0; i < sampleNumber; i+=4 ){
      temp1 = IMULM( coeff_a0, inSample1 );
      temp2 = IMULM( coeff_a1, state_0   );
      temp3 = IMULM( coeff_a2, state_1   );
      temp4 = IMULM( coeff_b1, state_2   );
      temp5 = IMULM( coeff_b2, state_3   );

      outSample1 = (temp1 + temp2 + temp3 - temp4 - temp5) <<SHIFT;
      output[i] = outSample1;

      temp6  = IMULM( coeff_a0, inSample2  );
      temp7  = IMULM( coeff_a1, inSample1  );
      temp8  = IMULM( coeff_a2, state_0    );
      temp9  = IMULM( coeff_b1, outSample1 );
      temp10 = IMULM( coeff_b2, state_2    );

      outSample2 = (temp1 + temp2 + temp3 - temp4 - temp5) <<SHIFT;
      output[i+1] = outSample2;

      temp11 = IMULM( coeff_a0, inSample3  );
      temp12 = IMULM( coeff_a1, inSample2  );
      temp13 = IMULM( coeff_a2, inSample1  );
```



```
      temp14 = IMULM( coeff_b1, outSample2 );
      temp15 = IMULM( coeff_b2, outSample1 );

      outSample3 = (temp1 + temp2 + temp3 - temp4 - temp5) <<SHIFT;
      output[i+2] = outSample3;

      temp16 = IMULM( coeff_a0, inSample4  );
      temp17 = IMULM( coeff_a1, inSample3  );
      temp18 = IMULM( coeff_a2, inSample2  );
      temp19 = IMULM( coeff_b1, outSample3 );
      temp20 = IMULM( coeff_b2, outSample2 );

      outSample4 = (temp1 + temp2 + temp3 - temp4 - temp5) <<SHIFT;
      output[i+3] = outSample4;

      state_0 = inSample4;  state_1 = inSample3;
      state_2 = outSample4; state_3 = outSample3;

      inSample1 = input[i+4]; inSample2 = input[i+5];
      inSample3 = input[i+6]; inSample4 = input[i+7];
   }
   state[0] = state_0; state[1] = state_1;
   state[2] = state_2; state[3] = state_3;

   return 0;
}
```

Table 39 Performance Table for IIR Filter 8

Treename	_iirFilter_8
Total Cycles (Instructions + Stall)	444472
MIPS (%)	0.266
Cycles per Sample	6.03
Instruction Cycles	444240
I-Cache Stall Cycles	203
D-Cache Stall Cycles	29
D-Conflict Cycles	0
(Useful) Operations per Instruction. Maximum is 5	2.50(2.46)

Further Optimization (4th Order, One Channel and Two Channels)

In audio signal processing, a cascade of two of the same second order IIR filters is often used. In this case the performance is much better when it is implemented in one function. This is shown in iirFilter_9.c. The number of local variables does not necessarily mean that they will all be assigned to registers for the whole loop; in this case we could have to deal with register spills. This can easily be proved, when looking in the iirFilter_9.s file. Decision tree 2 refers to the loop and it contains two loads and two

stores. This is due to the loop unrolling being performed twice. There is not more unrolling, because we used a pragma to switch it off.

```
int iirFilter_9( int *inputData, int *outputData,
                 int *coeff, int *state, int sampleNumber ){
   #pragma TCS_unroll=0

   int i;
   int * restrict input, * restrict output;
   int temp1A, temp2A, temp3A, temp4A, temp5A;
   int temp6A, temp7A, temp8A, temp9A, temp10A;
   int temp1B, temp2B, temp3B, temp4B, temp5B;
   int temp6B, temp7B, temp8B, temp9B, temp10B;
   int coeff_a0, coeff_a1, coeff_a2, coeff_b1, coeff_b2;
   int state_0,  state_1,  state_2,  state_3,  state_4, state_5;
   int inSample1A,  inSample1B;
   int outSample1A, outSample1B, outSample2A, outSample2B;

   input = inputData; output = outputData;
   coeff_a0 = coeff[0]; coeff_a1 = coeff[1]; coeff_a2 = coeff[2];
   coeff_b1 = coeff[3]; coeff_b2 = coeff[4];
   state_0  = state[0]; state_1  = state[1]; state_2  = state[2];
   state_3  = state[3]; state_4  = state[4]; state_5  = state[5];

   inSample1A = input[0]; inSample1B = input[1];

   for( i=0; i < sampleNumber; i+=2 ){
      temp1A = IMULM( coeff_a0, inSample1A );
      temp2A = IMULM( coeff_a1, state_0    );
      temp3A = IMULM( coeff_a2, state_1    );
      temp4A = IMULM( coeff_b1, state_2    );
      temp5A = IMULM( coeff_b2, state_3    );
      temp1B = IMULM( coeff_a0, inSample1B );
      temp2B = IMULM( coeff_a1, inSample1A );
      temp3B = IMULM( coeff_a2, state_0    );
      temp5B = IMULM( coeff_b2, state_2    );

      outSample1A = (temp1A + temp2A + temp3A - temp4A - temp5A) << SHIFT;
      temp4B = IMULM( coeff_b1, outSample1A );
      outSample1B = (temp1B + temp2B + temp3B - temp4B - temp5B) << SHIFT;

      temp6A  = IMULM( coeff_a0, outSample1A );
      temp7A  = IMULM( coeff_a1, state_2     );
      temp8A  = IMULM( coeff_a2, state_3     );
      temp9A  = IMULM( coeff_b1, state_4     );
      temp10A = IMULM( coeff_b2, state_5     );
      temp6B  = IMULM( coeff_a0, outSample1B );
      temp7B  = IMULM( coeff_a1, outSample1A );
      temp8B  = IMULM( coeff_a2, state_2     );
      temp10B = IMULM( coeff_b2, state_4     );

      outSample2A = (temp6A + temp7A + temp8A - temp9A - temp10A)<< SHIFT;
      temp9B = IMULM( coeff_b1, outSample2A );
      output[i] = outSample2A;

      outSample2B = (temp6B + temp7B + temp8B - temp9B - temp10B)<< SHIFT;
      output[i+1] = outSample2B;

      state_0 = inSample1B;  state_1 = inSample1A;
      state_2 = outSample1B; state_3 = outSample1A;
      state_4 = outSample2B; state_5 = outSample2A;
```



```
      inSample1A = input[i+2]; inSample1B = input[i+3];
   }
   state[0] = state_0; state[1] = state_1; state[2] = state_2;
   state[3] = state_3; state[4] = state_4; state[5] = state_5;

   return 0;
}
```

Furthermore it is seldom necessary to process only one channel. So it is a good practice to implement two and three channel versions of the filter. iirFilter_10.c shows a two channel fourth order IIR filter, constructed from two cascaded second order IIR filters.

```
int iirFilter_10( int *inputDataA,  int *inputDataB,
                  int *outputDataA, int *outputDataB,
                  int *coeffA, int *coeffB, int *stateA, int *stateB,
                  int sampleNumber ){
   int i;
   int * restrict inputA,  * restrict inputB;
   int * restrict outputA, * restrict outputB;
   int temp1A, temp2A, temp3A, temp4A, temp5A;
   int temp6A, temp7A, temp8A, temp9A, temp10A;
   int temp1B, temp2B, temp3B, temp4B, temp5B;
   int temp6B, temp7B, temp8B, temp9B, temp10B;
   int coeffA_a0, coeffA_a1, coeffA_a2, coeffA_b1, coeffA_b2;
   int coeffB_a0, coeffB_a1, coeffB_a2, coeffB_b1, coeffB_b2;
   int stateA_0,  stateA_1,  stateA_2,  stateA_3,  stateA_4, stateA_5;
   int stateB_0,  stateB_1,  stateB_2,  stateB_3,  stateB_4, stateB_5;
   int inSampleA,   inSampleB;
   int outSample1A, outSample1B, outSample2A, outSample2B;

   inputA = inputDataA; outputA = outputDataA;
   inputB = inputDataB; outputB = outputDataB;
   coeffA_a0 = coeffA[0]; coeffA_a1 = coeffA[1]; coeffA_a2 = coeffA[2];
   coeffA_b1 = coeffA[3]; coeffA_b2 = coeffA[4];
   coeffB_a0 = coeffB[0]; coeffB_a1 = coeffB[1]; coeffB_a2 = coeffB[2];
   coeffB_b1 = coeffB[3]; coeffB_b2 = coeffB[4];
   stateA_0 = stateA[0]; stateA_1 = stateA[1]; stateA_2 = stateA[2];
   stateA_3 = stateA[3]; stateA_4 = stateA[4]; stateA_5 = stateA[5];
   stateB_0 = stateB[0]; stateB_1 = stateB[1]; stateB_2 = stateB[2];
   stateB_3 = stateB[3]; stateB_4 = stateB[4]; stateB_5 = stateB[5];

   inSampleA = inputA[0]; inSampleB = inputB[0];

   for( i=0; i < sampleNumber; i++ ){
      temp1A = IMULM( coeffA_a0, inSampleA );
      temp2A = IMULM( coeffA_a1, stateA_0  );
      temp3A = IMULM( coeffA_a2, stateA_1  );
      temp4A = IMULM( coeffA_b1, stateA_2  );
      temp5A = IMULM( coeffA_b2, stateA_3  );
      temp1B = IMULM( coeffB_a0, inSampleB );
      temp2B = IMULM( coeffB_a1, stateB_0  );
      temp3B = IMULM( coeffB_a2, stateB_1  );
      temp4B = IMULM( coeffB_b1, stateB_2  );
      temp5B = IMULM( coeffB_b2, stateB_3  );

      outSample1A = (temp1A + temp2A + temp3A - temp4A - temp5A) << SHIFT;
      outSample1B = (temp1B + temp2B + temp3B - temp4B - temp5B) << SHIFT;

      temp6A  = IMULM( coeffA_a0, outSample1A );
      temp7A  = IMULM( coeffA_a1, stateA_2    );
      temp8A  = IMULM( coeffA_a2, stateA_3    );
```

```
      temp9A  = IMULM( coeffA_b1, stateA_4    );
      temp10A = IMULM( coeffA_b2, stateA_5    );
      temp6B  = IMULM( coeffB_a0, outSample1B );
      temp7B  = IMULM( coeffB_a1, stateB_2    );
      temp8B  = IMULM( coeffB_a2, stateB_3    );
      temp9B  = IMULM( coeffB_b1, stateB_4    );
      temp10B = IMULM( coeffB_b2, stateB_5    );

      outSample2A = (temp6A + temp7A + temp8A - temp9A - temp10A) << SHIFT;
      outputA[i] = outSample2A;

      outSample2B = (temp6B + temp7B + temp8B - temp9B - temp10B) << SHIFT;
      outputB[i] = outSample2B;

      stateA_1 = stateA_0; stateA_0 = inSampleA; stateA_3 = stateA_2;
      stateA_2 = outSample1A; stateA_5 = stateA_4; stateA_4 = outSample2A;
      stateB_1 = stateB_0; stateB_0 = inSampleB; stateB_3 = stateB_2;
      stateB_2 = outSample1B; stateB_5 = stateB_4; stateB_4 = outSample2B;

      inSampleA = inputA[i+1]; inSampleB = inputB[i+1];
   }
   stateA[0] = stateA_0; stateA[1] = stateA_1; stateA[2] = stateA_2;
   stateA[3] = stateA_3; stateA[4] = stateA_4; stateA[5] = stateA_5;
   stateB[0] = stateB_0; stateB[1] = stateB_1; stateB[2] = stateB_2;
   stateB[3] = stateB_3; stateB[4] = stateB_4; stateB[5] = stateB_5;

   return 0;
}
```

Table 40 Performance Table for IIR Filters 9 and 10

Treename	_iirFilter_9	_iirFilter_10
Total Cycles (Instructions + Stall)	739387	1080589
MIPS (%)	0.442	0.323
Cycles per Sample	5.01	3.66
Instruction Cycles	739152	1000368
I-Cache Stall Cycles	203	406
D-Cache Stall Cycles	32	79815
D-Conflict Cycles	4	74275
(Useful) Operations per Instruction. Maximum is 5	2.70(2.65)	3.70(3.66)

In function iirFilter_10 we also see a significant increase in the data cache stall cycles, most of which are conflicts. This is due to the ordering of data in the cache.

The data cache on TriMedia is 16 KB in size with a 64-Byte block size. To allow two accesses to proceed in parallel, the data cache is quasi-dual ported. The cache is implemented as eight banks of single-ported memory, but the hardware allows each bank to operate independently. Thus, when the addresses of two simultaneous accesses select two different banks, both accesses can complete simultaneously; trying to access the same bank simultaneously results in a data conflict.

A sample with the physical address &inputBufferA[i] is stored in one bank; the sample with the next address &inputBufferA[i+1] is in the next bank and so on. This means samples with addresses which are multiples of eight are stored in the same bank. This is useful when reading one array in ascending order. However, it leads to problems in the case discussed here. The distances between the physical addresses &inputBufferA[i], &inputBufferB[i], &outputBufferA[i] and &outputBufferB[i] are all multiples of eight and therefore in the same bank. In the end of the loop in iirFilter_10() 2 loads and 2 stores are issued. They are scheduled in the same instruction because of scheduling latency constraints. As a result this causes the data cache conflicts.

Hence, there are more than 74000 data conflict cycles when allocating memory for the data in a naive way:

```
#define SAMPLENUMBER 512

int inputBufferA[SAMPLENUMBER], outputBufferA[SAMPLENUMBER];
int inputBufferB[SAMPLENUMBER], outputBufferB[SAMPLENUMBER];
```

To avoid this problem we can either choose SAMPLENUMBER being 513 or 515, or rearrange the data as shown below. In both cases we allocate slightly more memory than necessary as a price for an increase in performance. For details refer to the appropriate TriMedia data book.

```
int intInputBufferA[SAMPLENUMBER+1], intOutputBufferA[SAMPLENUMBER+3];
int intInputBufferB[SAMPLENUMBER+5], intOutputBufferB[SAMPLENUMBER];
```

Table 41 Performance Table for IIR Filter 10 (Data-Cache Optimization)

Treename	_iirFilter_10
Total Cycles (Instructions + Stall)	1008143
MIPS (%)	0.302
Cycles per Sample	3.42
Instruction Cycles	1000368
I-Cache Stall Cycles	441
D-Cache Stall Cycles	7334
D-Conflict Cycles	63
(Useful) Operations per Instruction. Maximum is 5	3.70(3.66)

It can be seen that this saves more than 76000 cycles. Unfortunately this is not in the responsibility of the programmer of the processing function. This must be done in the context where the filter is used and the memory for the data allocated.

For completeness, another method of generating compact code shall be briefly mentioned. The compiler supports a technique called grafting. It increases parallelism in decision trees. This technique replaces any jump with a copy of a decision tree based on the probability a certain branch is taken. As a result the program size increases. Grafting can be considered similar to loop unrolling, but it does not reduce the loop overhead.

The advantage is mainly due to the use of the branch delay slots. There are several methods to tune the grafting performed by the compiler.

Results for the function **iirFilter_10** are given below:

Table 42 Performance Table for IIR Filter 10 (Grafting)

Treename	_iirFilter_10
Total Cycles (Instructions + Stall)	839742
MIPS (%)	0.251
Cycles per Sample	2.85
Instruction Cycles	823968
I-Cache Stall Cycles	1042
D-Cache Stall Cycles	14732
D-Conflict Cycles	9345
(Useful) Operations per Instruction. Maximum is 5	4.81(4.80)

Performance Summary

The methods described in this chapter were:

- Extensive use of local variables. Register spills should not be a problem in most cases. Best is always a before-and-after performance comparison.
- Using restricted pointers.
- Loads in the end of a loop (software pipelining).
- Loop unrolling by hand. Sometimes it can gain performance, but note that this can also be done by the compiler. There are several ways to tune this.
- Implementing more functionality in one function 2nd order to 4th order, multi-channel processing.
- Data cache optimization.
- Grafting (briefly).

Table 43 summarizes the results.

Table 43 Summary of Results

	Cycles per Sample[A]	**MIPS (%)[B]**
floating point arithmetic, 2nd order IIR, 1 channel		
iirFilter_1	32.16	1.418
iirFilter_2	11.54	0.509
iirFilter_3	10.79	0.476

Table 43 Summary of Results

	Cycles per Sample[A]	MIPS (%)[B]
iirFilter_4	10.04	0.443
iirFilter_5	10.03	0.442
fixed point integer arithmetic, 2nd order IIR, 1 channel		
iirFilter_6	25.16	1.110
iirFilter_7	7.04	0.310
iirFilter_8	6.03	0.266
fixed point integer arithmetic, 4th order IIR, 1 channel		
iirFilter_9	5.02	0.442
fixed point integer arithmetic, 4th order IIR, 2 channel		
iirFilter_10	3.67	0.324
-> data cache optimization	3.42	0.302
-> grafting	2.85	0.251

A. Number of cycles that is necessary to calculate one biquad for one sample.
B. The MIPS stated belong to the calculation of one channel, regardless whether a second or fourth order IIR filter is applied.

Using all these optimizations extensively for a 4th order IIR filter for three channels, the performance measured in cycles per sample for a second order IIR filter can be even less than 2.85 with a usage of almost 5 out of 5 issue slots.

However, if only a second order filter for one channel is to be used, it is probably difficult to get lower than 5 cycles per sample with a usage of less than 3 out of 5 issue slots. Therefore the first consideration for developing a signal processing system is to exploit parallelism and create blocks including loops with a reasonable amount of operations in the loop body. This also avoids unnecessary function calls.

Two techniques have been used. First, processing two channels in parallel in one loop results in completely independent operations that can be calculated in parallel. These are considerations a designer must examine.

Second, merging two equal second order IIR filters into one loop avoids loads and stores. Figure 64 shows how one load and one store can be saved per sample.

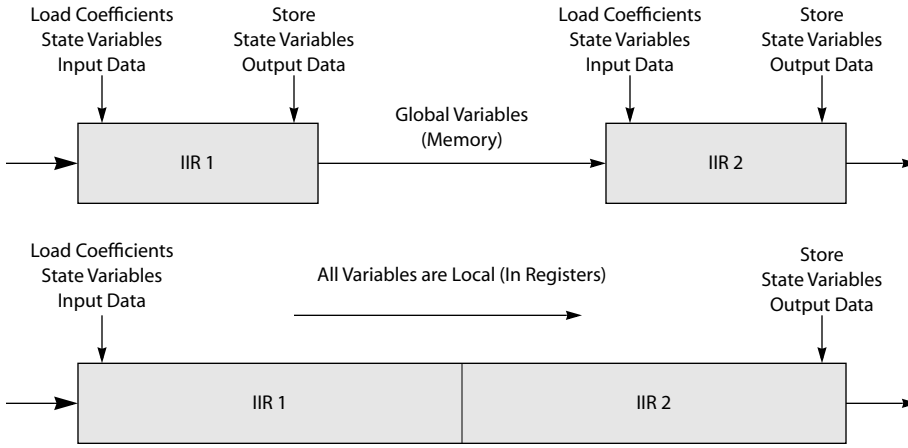

Figure 64 Merging Two Equal Second Order IIR Filters

Note that if the loop body is too long the compiler might break the resulting decision tree, and this will definitely affect the performance. In other words there is a trade-off that has to be investigated while optimizing.

Chapter 13

Interrupt Latency Support

Overview

Like in any other real time system, TriMedia-based multimedia applications are mostly driven by time-critical events. Such events are passed between the application and its environment by means of interrupts, which often also announce or request data. For instance, an MPEG-2 decoder is continuously reacting on interrupts announcing new MPEG data to decode, on interrupts requesting new frames to display, on interrupts notifying that a VLD- or ICP operation or a DMA transfer has completed, or on interrupts providing real time synchronization.

For many interrupts it is extremely important that they are handled and acknowledged in time. There are a number of reasons for this: first, contrary to software, which can implement various buffering schemes to overcome transient timing problems (*jitter*), hardware is relatively simple in nature. When an interrupt is not served in time, input data might get lost or a device might go into error because it did not get instructions on what to do next. As second reason for timely handling interrupts, especially in high frequency systems: any delay in handling the interrupt reduces the time available for processing the related event, thereby increasing the probability of real time problems 'higher up' the chain.

The meaning of the term 'in time', and the severity of the 'real time problems' is strongly dependent on the application and the devices which it uses. For example, in video capturing, all timings are related to the input frame rate so that 'in time' will probably be in the order of magnitude of several milliseconds, which is in contrast, for example, to an ssi interrupt, which must be served strictly within a few hundred microseconds. Similarly, the penalty of occasional timing problems in displaying video might only be some short, hardly noticeable reduction in video quality, while an occasional timing problem in an audio renderer might enrage the listener.

This chapter deals with interrupt latencies, as being an important concept in application timing. It describes a mechanism to measure interrupt latencies, giving insight in the timing aspects of applications. It also describes how to find the cause of long latencies, and concludes with latency information of a number of TriMedia applications and libraries.

Terminology

The following terminology is related to interrupt handling and application timing:

- An *interrupt handler* is a parameterless C function that is triggered by an interrupt. It must be compiled with a **#pragma TCS_handler** or **#pragma TCS_interruptible_handler**, and installed as corresponding to a specific interrupt using the tmInterrupts functions in the TriMedia device library. The difference between these two pragmas is that the **TCS_handler** causes the interrupt enable bit to be cleared for the duration of the handler, thereby disabling nested interrupts, while a **TCS_interruptible_handler** runs with interrupts on.

- Decision trees (dtrees) are TriMedia instruction sequences generated by the compiler which terminate in jump instructions (to the beginning of other dtrees). Interrupts will never take control during execution of a dtree. Instead, pending interrupts may take control only during jumps to other dtrees.
- More precisely, interrupts may take control only during the interruptible jump instructions generated by default by the TriMedia C compiler. Interrupt handling can be prevented even while jumping to other dtrees by using noninterruptible jump instructions. This special class of jump instructions is sometimes used in hand-coded assembly, e.g. to allow loop pipelining. (See the appropriate TriMedia data book for more information).
- *Grafting* is a technique, exploited by the TriMedia C compiler, to enlarge dtrees by merging it with copies of jump targets. It increases instruction level parallelism at the cost of (moderately) longer dtrees.
- The interrupt enable bit (IEN) in the TriMedia processor status word (PCSW) determines whether asserted interrupts are kept pending, or lead to invocation of their interrupt handler at the next jump instruction. The IEN controls all interrupts of interrupt priority 6 and lower (see further). It has no effect on interrupts of priority 7.
- An interrupt priority is a number in the range 0..7 (on TriMedia) assigned to each interrupt, and which controls the relative importance of the interrupt as follows. First, the hardware guarantees that, when multiple interrupts are pending at a particular jump instruction, an interrupt with highest priority value is selected for taking control. Second, the tmInterrupts functions of the TriMedia device library implements a scheme on top of the IMASK (see further) by which all interrupts of a specific priority or lower can be disabled "en masse" while leaving the higher priority interrupts enabled, by setting a global interrupt priority level.
- The IMASK is a bitvector on TriMedia by which interrupts can be specifically enabled or disabled. It should be accessed only via the tmInterrupts functions of the TriMedia device library. Contrary to the IEN bit, also interrupts of priority 7 can be disabled using the IMASK.
- Anon maskable interrupt (NMI) is an interrupt of priority 7. It is called this way because it cannot be disabled via the IEN. Because it is common practice to 'disable all interrupts' using the IEN, nonmaskable interrupts should only be used with extreme care.
- Disabling an interrupt is defined as any measure by which the interrupt's handler is prevented from taking control. Taking control is then postponed, and the interrupt remains pending. A particular interrupt is disabled (or masked, or blocked) during any of the following:
  - — During execution of a dtree.
  - — During a noninterruptible jump.
  - — When the IEN is cleared and when the interrupt's priority is lower than 7.

— When the corresponding bit in the IMASK is cleared. In terms of the tmInterrupts library, this is the case when the interrupt is not yet opened, or otherwise when the interrupt has been individually disabled or when the global interrupt priority level is larger than the interrupt's own priority.

- A *latency* of an interrupt is the difference in time between the moment at which the interrupt is asserted and the moment at which its handler starts executing. In other words, it is the time after asserting at which the interrupt handler is started. Any (noticeable) latency is caused by the application, by having disabled the particular interrupt.
- An *overrun* is a condition in which input data of a particular device (typically announced via an interrupt) is not timely consumed, and overwritten by subsequent data. Overruns are generally caused by interrupt latency problems.
- An *underrun* is a condition in which output data has not been given in time to a particular output device, causing the device to halt, or to continue with old, stale, or undefined data. Similar to overruns, underruns are generally caused by interrupt latency problems, for instance because response to a previous data request interrupt from the device was too late.

Summarized, using above terminology, longer latencies in an application may be harmful, since they reduce real time response. This may result in overrun errors of input devices, in which captured data is lost because the processor was notified to late to timely read it away and process it; or it may result in output device underrun errors in which no new output data has been made available in time because the processor has been too slow in reacting on a previous device notification.

Reasons for Long Interrupt Latencies

Interrupt latencies in the order of magnitude of about 10 microseconds and higher may theoretically be caused by long dtrees, especially in grafted code generated by the compiler.

However, it appears that such latencies very often are caused just by the application itself, by carelessly disabling and enabling interrupts.

Disabling all interrupts, or disabling one or several particular interrupts specifically, is generally applied to create critical sections for accessing global data structures which might also be accessed by interrupt handlers or by other tasks. This is best illustrated by means of a toy example:

volatile int g_count;

Handler

```
#pragma TCS_handler
g_count = g_count+1;
```

Task 1

```
Int ien = intClearIEN();
g_count = g_count+1;
```

Task 2

```
Int ien = intClearIEN();
g_count = g_count+1;
```


In this example, several tasks and an interrupt handler each modify a 'global data structure', g_count. The classical problem is that this modification involves a read of the old value, followed by a write of a new (incremented) value, and that a race condition results when this sequence is interrupted by one of the others between the read and the write. The following interruptions are possible:

- Task1 by Task 2, due to a pSOS timer interrupt which ends Task 1's time slice in favor of Task 2.
- Task 2 by Task 1 in a similar way.
- Task 1, or Task 2, by Handler due to occurrence of its interrupt.

Both tasks and the handler prevent such interruptions from happening during g_count's update by simply disabling 'all' interrupts; the tasks by calling the functions intended for this in the tmInterrupts library, and the handler by making use of compiler support via the TCS_handler pragma. This interrupt disabling is very effective, since it prevents time slicing because the pSOS timer interrupt is disabled, and it prevents the handler from interrupting the tasks and from interrupting itself (via a nested interrupt) because the handler interrupt is disabled.

Note that it is good practice to not simply enable the interrupts again at the end of a critical section which is started with a intClearIEN; rather, the old IEN should be restored because one can not always be sure that the interrupts were not already disabled. Also note that also handlers can create critical sections using intClearIEN and intRestoreIEN: use of these functions, in combination with pragma TCS_interruptible_handler, allow finer grain interrupt disabling in longer interrupt handlers. Especially note the following:

WARNING
it is not disallowed to call other functions when interrupts are disabled, but never call a function which might deschedule the current task when running under a multitasking operating system like pSOS.

Although interrupt disabling as described above is extremely effective for creating critical sections, it is also a very course method which should be avoided for critical sections longer than a few microseconds. The reason of this is that it might also lock out unknown interrupts with possible stringent latency requirements. Such an interrupt probably will not interfere at all with the critical section, and disabling it might unnecessarily increase its latency. The following sections go over the different mechanisms by which interrupts can be disabled. Some of these are more selective, and should be considered as an alternative.

Clearing the IEN

As mentioned above, disabling of an interrupt may be achieved by clearing the IE bit in the PCSW. Massively disabling all interrupts in this way, and later enabling them again is the usual way to achieve without much overhead a critical section in which a device, or global data structure can be accessed without the danger of a task context switch or a new intervening interrupt. Manipulating the IEN can be explicitly performed using the functions intClearIEN, intSetIEN and intRestoreIEN exported by the tmInterrupts device library. Two compiler- sup-ported mechanisms provide an effect similar to clearing the IEN:

- Defining an interrupt handler as using a pragma TCS_handler (in contrast to a TCS_interruptible_handler). The generated code for such a handler clears the IEN at the start, to be enabled at the end of the handler.
- Defining a function or handler as a TCS_atomic. For these functions, the compiler will generate non-interruptible jumps.

Note
Explicit use of non-interruptible jumps in handcoded assembly also locks out interrupts in an similar way.

Changing the Global Interrupt Priority

Interrupt disabling can also be achieved by raising the global interrupt priority to a higher value. This mechanism is generally used in interrupt handlers, to let serving not be disturbed by “less urgent” interrupts, while still allowing “more urgent” ones. So while the IEN is generally used to achieve atomicity, disabling based on interrupt priority is used to (temporarily) allocate processor cycles only to a certain minimal urgency. Although similar to clearing the IEN, raising the interrupt priority might also lock out unknown interrupts, it selects on a notion of urgency and for this reason it is less likely that interrupts with stringent latency requirements will be involuntarily locked out.

The global interrupt priority can be modified by means of a call to intSetPriority from the tmInterrupts library.

Individual Disabling

Interrupts can also be individually disabled. For instance, using a call to intInstanceS-etup from the tmInterrupts library, interrupt intVIDEOIN can be individually disabled; regardless of its priority, and it has no effect on other interrupts.

Preventing Task Preemption

Individual disabling, and raising the global interrupt priority level may be used to lock out all interrupts which might interfere with a particular critical section. However, it provides no control over task preemption. In other words, even with all "nasty" interrupts locked out, the current task might still be preempted by pSOS in favor of another which might enter the same critical section. Note that the actual problem here is that the identity of the pSOS timer interrupt and its priority are hidden.

Task preemption can be (temporarily) prevented by means of the pSOS function t_mode. This function does not disable any interrupt at all, but just prevents scheduling. In libraries or applications which may run either under pSOS or in stand-alone mode, it is better to use the AppModel functions, which can be used to abstract from the currently running operating system, as follows:

Task 1

```
#include <tmlib/AppModel.h>;
...
AppModel_suspend_scheduling();
g_count= g_count+1;
AppModel_resume_scheduling()
```

Task 2

```
#include <tmlib/AppModel.h>;
...
AppModel_suspend_scheduling();
g_count= g_count+1;
AppModel_resume_scheduling()
```

Interrupt Latency Sampling

Applications might encounter interrupt latency- related problems, even in case interrupts have been disabled with extreme care. Libraries might have to be certified on "decent" interrupt latency behavior. And both applications and libraries might be investigated on the (timing) effects of running them together with other applications or libraries. This section describes a sampling method which can be used for all these situations for gaining interrupt latency information. This chapter refers to an example program in which latency is measured. This example is reproduced at the end of this chapter, and it is also available among the examples included with the TriMedia SDE. See TCS/examples/misc/latency_sampler.

The latency sampling method records the interrupt latencies encountered by a periodic timer interrupt over a specified duration of time while the sampled application is running. Using a timer-based interrupt has the pleasant property that the times at which it is raised are known exactly (up to a few cycles), so that the latency can be easily obtained by subtracting this time from the actual time of handler invocation. The obtained timer interrupt latencies are recorded in a bucket array, where each bucket represents the number of timer interrupt latencies encountered during sampling. After termination of sampling, a latency histogram can be obtained by printing the values in the bucket array.

Although the latencies are measured for the timer interrupt only, they can be interpreted more generally: each measured latency would have been the latency of any interrupt

which was also enabled at the moment at which the timer interrupt occurred. The sampler as shown installs the timer interrupt at (lowest) priority 0, and hence, for any interrupt i which is not individually disabled: i is enabled whenever the timer interrupt is enabled, and this means that at any moment, i's latency is smaller than the timer interrupt latency. In other words, the measured interrupt latencies form a lowerbound, or worst case information, on the interrupt latency of any interrupt which is not individually disabled. This lowerbound could be tightened by running the timer interrupt at a higher interrupt level, thereby disregarding interrupt latencies encountered by non time critical interrupts.

In an application with one time critical, high priority interrupt, the sampled latencies are lowerbounds also in another sense: a measured latency could be caused by the high priority interrupt handler itself, because it was invoked at elapse of the sample timer. In this case the latency which was encountered by the timer interrupt would obviously not have been encountered by the high priority interrupt itself.

By the above, the described sampling method can be used to obtain information on interrupt latencies encountered by interrupts which have not been individually disabled.

Using the Sampler

Sampling can be performed simply by compiling and linking the listed C code to the application, and by calling function init_latency when sampling should start. This function clears the bucket array, allocates a timer, and sets it up to start sampling. After "some" time, sampling can be stopped by term_latency, which deallocates the timer and prints the histogram on the standard output.

Note that, being sampling based, the reliability of the obtained information is dependent on the sample frequency, the sample duration, and the code coverage of the application during sampling. For instance, no guarantee is given that the largest measured latency indeed is the theoretical worst case latency.

No analysis is made in this document on this reliability.

Detection of Latency Violators

The listed sampler can also be used to detect the causes of long latencies, as follows: upon any sampled latency larger than NROF_BUCKETS * 2 LOGS, the function LATENCY_VIOLATOR_DETECTED is called. This can be used to detect the part of the application which was responsible for this long latency, by placing a breakpoint in this function using the TriMedia debugger tmdbg. When hitting this breakpoint, the application completely stops with all interrupts disabled. A stack traversal will reveal the function which ended the violating critical section.

Breaking Decision Trees: #pragma TCS_break_dtree

When you find the interrupt latency is too high, you can control the latency by changing the way the program gets compiled. A high interrupt latency implies a large decision tree. The reason for the presence of a large decision tree could be too much grafting, or it could be a large decision tree even without grafting, where you might have hand unrolled loops to gain performance, something that is not too uncommon in DSP programming.

If grafting is the cause of large interrupt latency, you can use grafting parameters to reduce the amount of grafting performed. Because this might have a performance impact, you should exercise care in achieving a balance between performance and interrupt latency.

If the code has large decision trees even without grafting, you can use the pragma **TCS_break_dtree** to break the dtree at appropriate places. This also might have a performance impact. You must take special care to minimize the number of values living across the break of the dtree. These values now have to be stored in the global register set of the compiler with the accompanying save and restore code.

Latency Sampler Code

```
/*---------------------------- includes ----------------------------*/
#include <tm1/tmTimers.h>
#include <tm1/tmTimersmmio.h>
#include <tm1/mmio.h>
/*------------------------ local definitions ----------------------*/
#define NROF_BUCKETS 1000 /* number of sample buckets */
#define LOGS 4 /* binary logarithm of sample bucket size */
#define SAMPLE_PERIOD 1000 /* cycles */
static Int buckets[NROF_BUCKETS];
static Int sample_timer;
static Int last_tick;
custom_op Int cycles(void);
/*------------------------ utility functions ----------------------*/
LATENCY_VIOLATOR_DETECTED(){
intClearIEN();
   /* Place a breakpoint here */
   intSetIEN();
}
static void
sampler(void){
   #pragma TCS_handler
   Int now = cycles();
   Int sample_timer_value = timGetVALUE(sample_timer);
   Int this_tick = now - sample_timer_value;
   Int latency = now - last_tick - SAMPLE_PERIOD;
   Int bucket_nr = latency >> LOGS;
   last_tick = this_tick;
   if( bucket_nr >= NROF_BUCKETS ){
      buckets[NROF_BUCKETS - 1]++;
      LATENCY_VIOLATOR_DETECTED();
   }else if( bucket_nr < 0 ){
```

```
      buckets[0]++;
   }else{
      buckets[bucket_nr]++;
   }
}
Bool
init_latency(){
   timInstanceSetup_t setup;
   if (timOpen(&sample_timer) != TMLIBDEV_OK) {
      return False;
   }else{
      memset((Pointer) buckets, 0, sizeof (buckets));
      last_tick = cycles();
      setup.source = timCLOCK;
      setup.prescale = 1;
      setup.modulus = SAMPLE_PERIOD;
      setup.running = True;
      setup.handler = sampler;
      setup.priority = intPRIO_0;
      timInstanceSetup(sample_timer, &setup);
      return True;
   }
}
void
term_latency(){
   Int i;
   timClose(sample_timer);
   for( i = 0; i < NROF_BUCKETS; i++ ){
      if( buckets[i] ){
         printf(" %7d : %7d\n", i << LOGS, buckets[i]);
      }
   }
}
```

```
/*---------------------------- includes ----------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*---------------------------- functions --------------------------*/
main(){
   int i;
   init_latency();
   for( i = 1; i < 1000; i++ ){
      printf("cos(%d)= %e\n", i, cos(i));
   }
   term_latency();
   exit(0);
}
```

