Application Report
SPRA666 – August 2006

Hand-Tuning Loops and Control Code on the TMS320C6000

Elana Granston *Software Development Organization*

ABSTRACT

The TMS320C6000 compiler automatically performs a great deal of performance-related tuning. This compiler-driven optimization usually suffices. For the occasional cases where additional CPU performance is needed, this application report presents strategies and examples for improving performance of C/C++ applications. Memory-related performance improvements (such as background DMA transfers or cache usage) are outside the scope of this report. The techniques apply to all members of the C6000 architecture family.

The target audience is intermediate to advanced application developers. Familiarity with the C6000 architecture and experience developing code for this architecture is assumed.

Contents

Figures

1 Introduction

This application report provides techniques for hand-tuning loops and control code in C/C++ applications for the TMS320C6000 architecture. The remainder of this document is structured as follows:

- Section 2 introduces the architectural features of the C6000 family of processors. These are necessary for understanding the tuning techniques presented later in this document. This section also presents an overview of the C6000 compiler.
- Section 3 outlines a general performance tuning strategy and assists with the selection of compiler options used for code optimization.
- Section 4 teaches how to optimize software-pipelined loops.
- Section 5 focuses on control code optimization.
- Section 6 concludes the document.
- Section 7 provides links to related documents.

Unless otherwise specified, examples in this application report use C6000 compiler version 5.1.3 (CCStudio version 3.1) and target the C64x.

2 Overview of the TMS320C6000

2.1 TMS320C6000 Architecture

The C6000 architecture is an eight-way enhanced VLIW architecture. The C64x, C64x+, C67x+ contain 64 registers. The C62x and C67x devices contain 32 registers. The architecture is partitioned into two nearly symmetric halves (A-side and B-side) with limited connectivity between the two. All of these registers can be used for either data or addressing; there are no dedicated address registers.

Each side contains half the registers and four functional units, noted by the types of instructions that execute on them:

- **M unit.** Multiplication
- **D unit.** Loads and stores
- **S unit.** Shifts, branches and compares
- **L unit.** Logical and arithmetic operations

Each side contains two other resources that are important to consider when scheduling code:

- **X cross-path.** Path between functional units and opposite-side register file
- **T address path.** Path to memory

Some operations can be done on more than one functional unit. For example, add instructions can execute on the L, S, or D units. Several other operations can be done on either the L or S units.

Normally, each load and store uses one D unit and one T address path. The register containing the address must be on the same side as the D unit. The data register(s) must be on the same side as the T address path. All processor variants support byte/half-word/word (that is, 8-, 16-, and 32-bit) loads and stores. The C67x, C64x and C64x+ support aligned double-word (64-bit) loads. The C64x and C64x+ support aligned double-word stores. The C64x and C64x+ also allow non-aligned word-wide and non-aligned double-word wide memory-access instructions; these non-aligned loads and stores require the use of both T address paths.[1]

Most instructions can be predicated with a condition operator (that is, whether the instruction is executed depends on the value of another register). Many allow one register source operand to come from the opposite register file. When an instruction takes one of its register operands from the opposite side, it uses the X cross-path bus. There is one X cross path resource for each side: 1X = B-side register to an A-side unit, 2X=A-side register to a B-side unit.

The T address path is a bus that connects a register set to memory. It consists of two parts:

- an address bus where that pointer value originates in either the D1 or D2 units
- a data bus that connects memory to a specific register set

T1 services the A-side. T2 services the B-side.

Here are two examples of C6000 instructions:

```
       LDNDW     .D1T2  *A0++,B5:B4  ; Non-aligned double-word load
                                     ; Address from A-side, data goes to B-side

[!A0]  MPY       .M2X   B7,A9,B8     ; Multiply instruction, uses X cross-path for A9
                                     ; Instruction executed only when A0 is zero.
```

Note that, although the non-aligned load uses both T address paths, only the one that is on the same side as the data is encoded in the instruction. The other is implicit.

All C6000 variants support a 32-bit instruction set. The C64x+ supports a compact 16-bit instruction set, as well. The 16-bit instructions map to common 32-bit instructions. The object file is compressed automatically by the assembler (by replacing a 32-bit instruction with its 16-bit equivalent), whenever possible.

The C64x+ also supports a loop buffer. Exploiting the loop buffer saves power because loop body instructions are fetched only once per loop invocation (when the first iteration is executed) rather than once per loop iteration. Using the loop buffer reduces code size because fewer instructions must be explicitly represented in the object file than without. When code must be interruptible (reference [4]) or in borderline cases where resources are tight, the loop buffer also improves performance. There are special assembly instructions to tell the processor to enable the loop buffer for a given loop.

For more information on the C6000 architecture, see reference [2].

[1] Non-aligned memory accesses are less efficient than aligned memory accesses, but available for the cases where data alignment cannot be guaranteed.

2.2 TMS320C6000 Compiler

The C6000 compiler accepts C, C++ or linear assembly as input.[2] When compiling C or C++, the compiler proceeds in five basic phases:

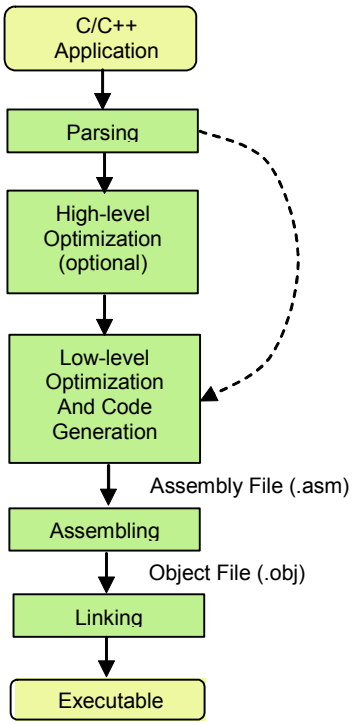

Figure 1. Compiler Phases

First, the source file is parsed. The output of this phase is a high-level intermediate representation that closely resembles the source language. Functions compiled with optimization (-o1, -o2 or –o3) then pass through the high-level optimizer. This phase operates on the high-level intermediate representation, performing analysis, function inlining, loop transformations and other code reordering. When linear assembly is compiled, high-level optimization is skipped.

Next, the high-level intermediate language is translated line-by-line into a low-level intermediate language, which closely resembles assembly. The low-level optimizer operates on this low-level intermediate language. Peephole-based optimizations, partitioning, scheduling and register allocation are performed during this phase. The output of this phase is the assembly file. The remaining two phases are assembling and linking the code.

The first compiler version to support the C64x+ is 6.0.1 (CCStudio 3.2). On the C64x+, which supports compact instructions, the assembly code is automatically compacted as it is assembled into object code. Additionally, the compiler automatically generates the required instruction sequence to exploit the loop buffer whenever possible.

[2] *Linear assembly* is a source language consisting primarily of unscheduled assembly instructions. See references [1] and [4] for more detail.

3 General Performance Tuning Strategy

This section presents a general approach to performance tuning. It is assumed that the application has been fully debugged before this process is started.

3.1 Selecting the Right Compiler Options

The compiler's ability to generate code can be tuned to better meet application goals. The easiest way to affect the compiler's output is to exploit the wide variety of options (sometimes called switches) provided within the tool. When tuning for performance, first and foremost, start with performance-oriented options:

Definitely use the following:

- **–o[2|3].** Optimization level. This option is critical for generating efficient code sequences. At –o3, file-level optimization is performed. At –o2 function-level optimization is performed. At –o1, high-level optimization is minimal. At –o0, high-level optimization is omitted completely. Note that –o defaults to –o2. In the absence of any –o options, the optimization level defaults to –o0.

  By default, the –o switch optimizes for performance. This can increase code size. If code size is an issue, *do not reduce the level of optimization*. Instead, use the –ms switch to affect the optimization goal (performance versus code size).

When safe, consider using the following:

- **–mt.** Assume no pointer-based parameter writes to a memory location that is read by any other pointer-based parameter to the same function. This option is generally safe except for *in place* transforms (where modified data is written back to the same memory location from which it was initially read). Most users avoid in-place transforms for performance reasons.

  For example, consider the following function:

```
selective_copy(int *input, int *output, int n)
{
    int i;
    for (i=0; i<n; i++)
       if (myglobal[i]) output[i] = input[i];
}
```

  –mt is safe when the memory ranges pointed to by "input" and "output" do not overlap.

  Be aware of the limitations of -mt. This option applies *only* to pointer-based function parameters.

  – It says nothing about the relationship between parameters and other pointers accessed in the function (for example, "myglobal" and "output").
  – It says nothing about non-parameter pointers used in the function.
  – It says nothing about pointers that are members of structures, even when the structures are parameters.
  – It says nothing about pointers that are dereferenced via multiple levels of indirection.

  Hence, -mt is not a substitute for using restrict-qualifiers (Sections 4.1 and 5.1), which are key to achieving good performance.

One last note, using –mt broadly acts upon the full scope of the code, either file- or project wide. Use restrict-qualifiers to handle this problem with more precision.

- **–mh<num>.** Permit compiler to fetch (but not store) array elements beyond either end of an array by <num> bytes. This option (known as *speculative loads*) provides the compiler with extra flexibility when scheduling loops. It can lead to better performance, especially for "while" loops. It can lead to smaller code size for both "while" loops and "for" loops. If –mh is used without <num>, there is no limit to the number of bytes read past either end of the arrays.

  The software-pipelined loop information in the compiler-generated assembly file notes when adding –mh<num> (or using –mh with a larger value) might improve performance or code size. For example, suppose a (function containing a) loop is compiled without –mh or with –mh<num> where <num> is less than 56. The compiler might output a message similar to:

```
;*      Minimum required memory pad : 0 bytes
;*
;*      For further improvement on this loop, try option -mh56
```

  This message communicates that currently the compiler is fetching 0 bytes beyond the end (or beginning) of any array. However, if the loop is rebuilt with –mh<num> where <num> is at least 56, there might be better performance and/or smaller code size. The compiler consultant (Section 4.1.6) also provides this information.

  When using this option, ensure that there is a buffer of <num> bytes on both ends of all sections that contain array data. *This is the user's responsibility.* Padding can be implemented by shrinking the associated memory region in the linker command file. For example, suppose the original memory region is defined as:

```
MEMORY {
         myregion: origin = 1000, length = 4000
       }
```

  If the goal is to pad the beginning and end of a region by 56 bytes, the region must be shrunk by 2 * 56. The new origin is 1000+56 and the new length is 4000-2*56 = 3888:

```
MEMORY {
           /* pad (reserved):  origin = 1000, length = 56    */
           myregion:           origin = 1056, length = 3888
           /* pad (reserved):  origin = 3944, length = 56    */
       }
```

  The commented lines are not necessary, but provide a reminder not to put other array data in those regions. Alternatively, one can use other memory areas (code or independent data) as pad regions, provided there is no conflict with EDMA transfers and/or cache-based operations.

If the source code contains many functions that are never executed, consider using:

- **–mo.** Place each function in its own *input (sub-)section*. Normally, input sections contain multiple functions. By default, all code (that is, all processor instructions) is placed into an input section called ".text" as defined by the Common Object File Format (COFF) used by the C6000 code generation tools. The linker then groups input sections into *output sections* (various ranges of memory) as defined by the linker command file. If *all* functions in an input

section are never executed (that is, if an input section contains only dead code), then the input section is omitted from the executable.

When using the –mo option, each function is put into its own sub-section; for example, the function dotp() is put into section ".text:.dotp()". Because there is only one function per input section, the linker can be more aggressive with respect to the removal of functions that are never executed. This can reduce the memory footprint of the resulting executable and, hence, reduce memory cycles as well.

This benefit comes at a cost. On the C6000, each input section must be aligned on a 32-byte boundary. The more input sections there are, the more space is potentially wasted on alignment. Hence, the benefit of this option depends on the percentage of dead source code and the original grouping of functions into input sections. Thus, this option may improve the performance and/or code size of some applications while hurting others. However, the differences can be significant, so give it a try.[3]

If code size is a concern, consider using the following:

- **–ms[0-3].** Adjust optimization goal. Higher values of <num> increasingly favor code size over performance. Thus, the higher the value of <num>, the stronger the request. Recommended to be used in conjunction with –o2 or –o3.

  Try –ms0 or –ms1 with performance critical code. Consider –ms2 or –ms3 for seldom-executed code (such as initialization routines). Note that –ms defaults to –ms0.

Do not use the following:

- **–g.** Support full symbolic debug. Great for debugging. Do not use in production code. –g inhibits code reordering across source line boundaries and limits optimizations around function boundaries. This results in less parallelism, more nops and generally less efficient schedules. Can cause a 30-50% performance degradation for control code, generally somewhat less but still significant degradation for performance critical code. Moreover, beginning with CCStudio 3.0 (C6000 compiler version 5.0), basic function-level profiling support is provided by default.
- **–gp.** Provide support for function-level profiling. Obsolete. Provided by default.
- **–ss.** Interlist source code into assembly file. As with –g, this option can negatively impact performance.
- **–ml3.** Compile all calls as far calls. Obsolete. Beginning with CCStudio 3.0 (C6000 compiler version 5.0), the linker automatically fixes up near calls that do not reach by using trampolines. In most cases, few calls need trampolines, so removing –ml3 usually makes code a few percent smaller and faster. By default, scalar data (pointers, integers etc.) are near and aggregates (arrays, structs) are far. This default works well for most applications.
- **–mu.** Turn off software-pipelining, which is a key optimization for achieving good performance on the C6000 processor. Great for debugging. Do not use in production code.

[3] An alternative to –mo is to use –pm –op2 –o3 in combination with the FUNC_EXT_CALLED pragma. This combination allows the compiler to eliminate unused functions without the additional padding overhead. Thus, overall code size may be smaller. To use this option, all files to optimize across must be simultaneously visible to the compiler; for example, cl6xa.c, cl6xb.c, cl6xc.c, … . This alternative has two drawbacks: first, a build environment modification may be required. Second, there is a limit to the size of the application for which this option can be used. The latter limitation can be overcome by splitting the application into logical modules with limited entry points and building each module separately. See reference [1] for more detail.

Options that reduce tuning time by providing additional analysis information, while having no effect on performance or code size:

- **–s [–k|–al] –o[2|3].** Output a copy of what the source code looks like after high-level optimization (Figure 1). This output, known as optimizer comments, looks much like the original C/C++, except that all inlining, transformations and other optimizations from this phase have been applied. Optimizer comments are interlisted with assembly code in the assembly file (with -k) and/or listing file (with -al). *This option is incredibly helpful in understanding the compiler-generated assembly.* See Section 3.3 for more detail.
- **–mw or –mw –al.** Output extra information about software-pipelined loops, including the *single scheduled iteration* (Section 4) of the loop. This information is used in the loop tuning examples presented later in this document.
- **–on2 –o3.** Create a .nfo file with the same base name as the .obj file. This file contains summary information regarding the high-level optimizations that have been applied, as well as advice.
- **--consultant.** Generates information to be used with the CCStudio Compiler Consultant. Provides tuning advice on a loop-by-loop level. Must be used in conjunction with CCStudio version 3.0 or higher.

  The Compiler Consultant provides beginning to intermediate-level tuning advice. Following the advice yields performance improvement in most cases, but improvement is not guaranteed. Sometimes it is necessary to iterate or apply multiple pieces of advice before seeing any improvement. See reference [3] for more detail.

For additional information on compiler options, see reference [1].

3.2 Basic Performance Tuning Tips

For simplicity, it is highly recommended to follow these guidelines before modifying source code:

- Choose performance oriented options (Section 3.1).
- Make sure index variables are signed integers. This communicates to the compiler that increments are linear (do not wrap around). This includes index variables used to do counting in loops. It also includes index variables used for subscripting. When the compiler knows that index variables are linear, there are more opportunities for optimization.
- Provide as much information as possible to the compiler. Use restrict qualifiers, MUST_ITERATE pragmas, and _nasserts() whenever possible (Sections 4.1 and 5.1).
- Align data on cache line boundaries and/or double-word boundaries when possible using the DATA_ALIGN and STRUCT_ALIGN pragmas. See reference [1] for more detail.
- Follow the advice in the Compiler Consultant (--consultant) and .nfo files (-on2 –o3).

In an ideal world, these guidelines should be applied when the code is initially written as the original author is probably the most knowledgeable regarding implicit assumptions. When retrofitting this information to a large existing code base, use profile information to determine which functions are the most performance critical. Start with those functions. Examples provided later in this document show when, where, and how to provide information. In most cases, when tuning is required, following these guidelines should suffice for meeting performance goals.

3.3 Exploiting Optimizer Comments (-s)

Optimizer comments help you understand both the resulting assembly code and the set of high-level transformations applied to your source code. When compiling with –s and [-o|-o1|-o2|-o3] -k, optimizer comments are interlisted with assembly code in the assembly file.[4] There is no performance or code size impact when optimizer comments are interlisted.

When the compiler transforms code, it attempts to reassociate source line numbers with the optimized code. The reassociated line numbers are included (to the left of each line) in the optimizer comments. Most assembly instructions in a compiler-generated assembly file also have line numbers (appended in the comment field). After serious optimization, the assembly line numbers might not correspond well to the original source. However, the assembly line numbers correspond to the line numbers in the optimizer comments almost verbatim.

Before diving into a file of generated assembly code, read the optimizer comments. Even experienced programmers save significant time by doing so. There is no good reason to avoid this option since it has no performance or code size impact when compiling with optimization.

As a simple example of exploiting optimizer comments, compile the following loop for the C64x with recommended options: –o –s –mw –mv6400.

```
void BasicLoop(int *output, int *input1, int *input2, int n)
{
    int i;
    for (i=0; i<n; i++)
        output[i] = input1[i] + input2[i];
}
```

Extract the optimizer comments by searching for all lines that begin with ";**".[5]

```
;** 4   ----------------------    if ( n <= 0 ) goto g4;
;**     ----------------------    U$11 = input1;
;**     ----------------------    U$13 = input2;
;**     ----------------------    U$16 = output;
;**     ----------------------    L$1 = n;
;**     ----------------------    #pragma MUST_ITERATE(1, 1099511627775, 1)
...
;**     ----------------------g3:
;** 5   ----------------------    *U$16++ = *U$11+++*U$13++;
;** 4   ----------------------    if ( --L$1 ) goto g3;
;**     ----------------------g4:
;**     ----------------------    return;
```

Observe that the compiler inserted a check for a zero-trip loop (the case where the body of the loop is never executed) and branches around the loop in this case. This is necessary for correctness. However, if it is known that the trip count (number of times the loop body is executed) is always greater than or equal to one, insert a MUST_ITERATE pragma immediately before the loop to communicate this to the compiler:

[4] When compiling without optimization (or with –o0), source code is interlisted instead of optimizer comments.

[5] This can be accomplished with the command `grep ";\*\*" ' on Unix, Linux or Windows Systems that support Cygwin or MKS Toolkit. If the file contains more than 1000 lines, search for ";*" instead (using `grep ";*" '). The set of file lines that match will be a superset of the optimizer comments.

```
void BasicLoop(int *output, int *input1, int *input2, int n)
{
    int i;
    #pragma MUST_ITERATE(1)
    for (i=0; i<n; i++)
        output[i] = input1[i] + input2[i];
}
```

Recompile and extract the optimizer comments. The branch around the loop has disappeared.

```
;**     ----------------------    U$9 = input1;
;**     ----------------------    U$11 = input2;
;**     ----------------------    U$14 = output;
;**     ----------------------    L$1 = n;
;**     ----------------------    #pragma MUST_ITERATE(1, 4294967295, 1)
...
;**     ----------------------g2:
;** 6   ----------------------    *U$14++ = *U$9+++*U$11++;
;** 5   ----------------------    if ( --L$1 ) goto g2;
;**     ----------------------    return;
```

The compiler-generated assembly has now been tuned just by looking at the optimizer comments. Section 5.1.2 provides another example of exploiting optimizer comments to better understand the resulting assembly code.

As a side note, it is commonly accepted that where performance is concerned, assembly language programmers leverage specific knowledge of their application (for example, characteristics of data sets such as size or range of values) while writing their code. This information cannot be obtained through compiler analysis or via visual inspection of the code. This creates an unfair advantage when assembly code performance is compared to C/C+ (compiler) performance. The preceding example provides one good example of this. Had this code been written in assembly, the programmer would have exploited knowledge about the trip count and omitted the zero-trip loop test, thereby improving both performance and code size.

Texas Instruments has worked hard to provide programmers with the ability to impart this kind of knowledge to the compiler using restrict qualifiers, MUST_ITERATE pragmas, and _nasserts(). When programmers exploit this chance to pass on information, the compiler can automatically apply improvements that depend on such information for either safety or profitability reasons.

4 Tuning Software-Pipelined Loops

A particular area where the C6000 processor family shines is its ability to speed through looped code. This is quite advantageous in digital signal processing, image processing and other mathematical routines that tend to be loop-centric. A technique called *software pipelining* contributes the biggest boost to improving looped code performance. Software pipelining is only enabled at –o2 or –o3. On all C6000 variants except the C64x+, software pipelining is completely disabled when code size flags –ms2 and –ms3 (Section 3.1) are used. On the C64x+, software pipelining is enabled if and only if the loop buffer (Section 2.1) can be used.

Without software pipelining, loops are scheduled so that loop iteration i completes before iteration i+1 begins. Software pipelining allows iterations to be overlapped. Thus, as long as correctness can be preserved, iteration i+1 can start before iteration i finishes. This generally permits a much higher utilization of the machine's resources than might be achieved from non-software-pipelined scheduling techniques.

In a software-pipelined loop, even though a single loop iteration might take s cycles to complete, a new iteration is initiated every ii cycles.

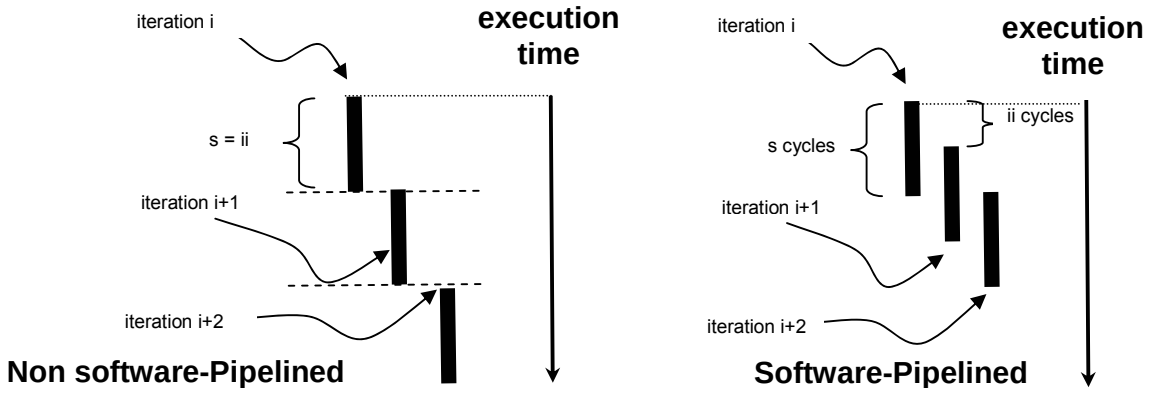

Figure 2. Software Pipelining

In an efficient software-pipelined loop, where ii<s, ii is called the *initiation interval* ; it is the number of cycles between starting iteration i and iteration i+1. ii is equivalent to the cycle count for the software-pipelined loop body. s is the number of cycles for the first iteration to complete, or equivalently, the length of a *single scheduled iteration* of the software-pipelined loop.

Because the iterations of a software-pipelined loop overlap, it can be difficult to understand the assembly code corresponding to the loop. If the source code is compiled with –mw, the software-pipelined loop information displays the scheduled instruction sequence for a single iteration of the software-pipelined loop. Examining this single scheduled iteration makes it easier to understand the compiler's output. This, in turn, makes tuning easier.[6]

Section 4.1 focuses on identifying and eliminating performance bottlenecks in software-pipelined loops. Restrict qualifiers, MUST_ITERATE pragmas and _nasserts() are used to improve loop performance. For the occasional case where these low-touch techniques do not suffice, Section 4.4 presents an example of hand-tuning a software-pipelined loop using intrinsic functions.

4.1 Using restrict qualifiers, MUST_ITERATE pragmas, and _nasserts()

The more information the compiler has, the better the optimization decisions that it can make. When using annotations, make sure the information being communicated is correct. If the information is not correct, the resulting code will not be correct either.

4.1.1 Establishing a Baseline

Compile BasicLoop() from Section 3.3 with the same options as before. Open up the assembly file and look at the software pipelining information for this loop:

[6] On the C64x+, when the loop buffer is used, only a single iteration of the loop is explicitly represented in the assembly code. See reference [4] for more detail.

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 5
;*      Loop opening brace source line   : 5
;*      Loop closing brace source line   : 6
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 7
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1
;*      .D units                     2*       1
;*      .M units                     0        0
;*      .X cross paths               1        0
;*      .T address paths             2*       1
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          1        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1
;*      Bound(.L .S .D .LS .LSD)     1        1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 7  Schedule found with 1 iterations in parallel
...
;*        SINGLE SCHEDULED ITERATION
;*
;*        C25:
;*   0              LDW     .D1T1   *A4++,A3          ; |6|  ^
;*     ||           LDW     .D2T2   *B4++,B5          ; |6|  ^
;*   1      [ B0]   BDEC    .S2     C24,B0            ; |5|
;*   2              NOP             3
;*   5              ADD     .L1X    B5,A3,A3          ; |6|  ^
;*   6              STW     .D1T1   A3,*A5++          ; |6|  ^
;*   7              ; BRANCHCC OCCURS {C25}           ; |5|
;*----------------------------------------------------------------------------*
L1:    ; PD LOOP PROLOG
;** ---------------------------------------------------------------------------*
L2:    ; PIPED LOOP KERNEL
           LDW     .D1T1   *A4++,A3          ; |6| <0,0>  ^
||         LDW     .D2T2   *B4++,B5          ; |6| <0,0>  ^

   [ B0]   BDEC    .S2     L2,B0             ; |5| <0,1>
           NOP             3
           ADD     .L1X    B5,A3,A3          ; |6| <0,5>  ^
           STW     .D1T1   A3,*A5++          ; |6| <0,6>  ^
;** ---------------------------------------------------------------------------*
L3:    ; PIPED LOOP EPILOG
;** ---------------------------------------------------------------------------*
```

The software-pipelined loop information includes the source lines from which the loop originates, a description of the resource and latency requirements for the loop, and whether the loop was unrolled (among other information). When compiling with –mw, the information also contains a copy of the single scheduled iteration. See references [1] and [4] for a more complete description.

The initiation interval (ii) is 7. This means that in the steady state, a result (equivalently, an original loop iteration) is computed every 7 CPU cycles. Therefore, the baseline CPU performance is 7 cycles/result.

Observe that the length of the single scheduled iteration is also 7. Thus, only one iteration is being executed at any given time, so there is no overlap across iterations (which would generally not be optimal).

4.1.2 Eliminating Loop-Carried Dependencies

Look again at the software pipelining information in the assembly file. Where is the bottleneck? To find it, one must understand how the compiler computes a lower bound on the cycle count for the loop. This lower bound is the maximum of the *loop-carried dependency bound* and the *resource bound*. The loop-carried dependency bound is based on ordering constraints among the assembly instructions. For example, the two loads must complete before the add can proceed. The resource bound is based on hardware constraints, such as the required number of functional units of each given type. In actuality, there are two resource bounds: partitioned and unpartitioned. In this case, both are the same.

In this case, the partitioned resource bound is 2. Even if the assembly instructions could be executed out of order, at least two cycles would be required to execute all instructions in the loop body. However, the loop-carried dependency bound is 7.

```
;*      Loop Carried Dependency Bound(^) : 7
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
```

Thus, ii ≥ max(2,7). To reduce the ii and consequently the number of cycles/result, the loop-carried dependency bound must be reduced.

The loop-carried dependency bound arises because there is a cycle in the ordering constraints for the instructions. The maximum cycle length is the loop-carried dependency bound. To reduce or eliminate a loop-carried dependency, one must identify the cycle and then find a way to shorten or break it.

To identify the maximum loop-carried dependency cycle, refer to the instructions in the single iteration. The instructions involved in the critical cycle are marked with a (^) sign. These instructions include the two loads, the add and the store.

```
;*        SINGLE SCHEDULED ITERATION
;*
;*        C25:
;*   0              LDW     .D1T1   *A4++,A3          ; |6|  ^
;*     ||           LDW     .D2T2   *B4++,B5          ; |6|  ^
;*   1      [ B0]   BDEC    .S2     C24,B0            ; |5|
;*   2              NOP             3
;*   5              ADD     .L1X    B5,A3,A3          ; |6|  ^
;*   6              STW     .D1T1   A3,*A5++          ; |6|  ^
;*   7              ; BRANCHCC OCCURS {C25}           ; |5|
```

With this information and by looking at which instructions feed into each other, one can reconstruct the loop-carried dependency cycle. The *nodes* in the graph are precisely the instructions denoted by (^) signs. The *edges* (arrows between node pairs) denote the ordering constraints. The edges are annotated by the number of cycles needed between the source and destination instructions. In most cases, results are written to registers at the end of the cycle in which the instruction executes and available on the following cycle. One of the few exceptions is that certain loads take 5 cycles for the data to be available in the target register.

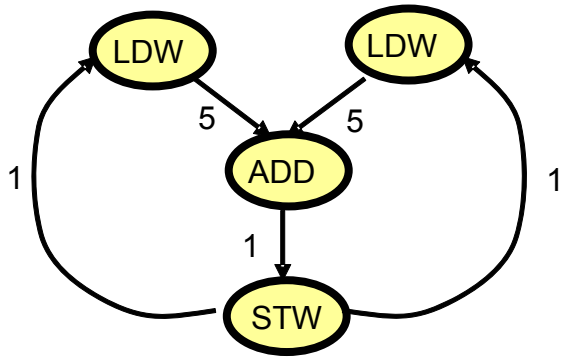

Figure 3. Loop-Carried Dependency Cycles

In this graph, there are two critical cycles, each with length 7. To reduce the loop-carried-dependency bound, the largest cycle in the graph must be shortened or eliminated. This can be accomplished by eliminating one of the edges in the cycle. To do so, one must understand the origin of the edges.

The edges from the load instructions to the add instructions are straightforward. The destination registers for the loads are the source registers for the add instruction. A load instruction takes 5 cycles to populate its destination register. Consequently, the add instruction cannot be executed until 5 cycles after the last of the two loads has been executed.

The edge from the add to the store is also straightforward since the destination of the add is the source of the store. The result of the add is available after 1 cycle. Consequently, the edge between the add and the store is annotated with a 1. The store can be executed on the cycle immediately following the add.

The edges from the store back to the loads are less obvious. How does one know to put them in? Why are they there? The answer to these questions is determined by process of elimination. Since there is no register dependence, there is most likely a memory dependence. In this case, the compiler does not know whether the input arrays could reference the same memory locations as the output, so it is conservative and assumes this is possible. The edge from the store to the loads ensures that the store from one iteration executes before the loads in the next iteration, just in case those loads try to read the data written to by the store. Whether this occurs in practice depends on the values of “input1”, “input2” and “output” at run time.

In reality, experienced programmers generally write code so that input parameter arrays and output parameter arrays are independent. The reason is that this makes their algorithm much more parallel, which in turn leads to better performance. Suppose that, across all call sites, neither “input1” nor “input2” ever accesses the same memory locations as “output”. Tell this to the compiler and the back edges from the store to the loads will be eliminated. This is done either by using the –mt option (Section 3.1) or by using the restrict keyword.

```
void BasicLoop(int *restrict output,
               int *restrict input1,
               int *restrict input2,
               int n)
{
    int i;
    #pragma MUST_ITERATE(1)
    for (i=0; i<n; i++)
        output[i] = input1[i] + input2[i];
}
```

While it is sufficient for this example to restrict qualify either both loads or the single store, it is recommended to restrict qualify all parameters that can be restrict qualified (and local pointer variables as well). First, this is usually quicker than determining which actually need to be restrict-qualified. Second, this provides information to other programmers who might maintain or modify this code base in the future. However, before inserting restrict keywords, be sure that pointers being restrict-qualified cannot overlap with any other pointers. When writing a library routine and using restrict, be sure to document parameter restrictions for library users.

After adding the restrict keyword, rebuild the function. Observe that the loop-carried dependency bound has disappeared. This means that each iteration is independent. New iterations are now initiated as soon as resources are available.

```
;*      Loop Carried Dependency Bound(^) : 0
```

Further note that a new iteration is initiated every 2 cycles. Thus, in the steady state, a new result is produced every 2 cycles (instead of every 7 cycles).

```
;*         ii = 2  Schedule found with 4 iterations in parallel.
```

4.1.3 Balancing Resources

Look at the software pipelining information for the loop. The limiting factor is now the number of functional units as indicated by the resource bound:

```
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
```

Which functional unit is the bottleneck? To determine this, look at the detailed breakdown of functional units required to execute a single iteration. Recall that the C6000 architecture is partitioned into two nearly symmetric halves. The resource breakdown displayed in the software pipelining information is computed after the compiler has partitioned instructions to either the A-side or the B-side.

Look for the machine resources with a (*) after them. Notice which ones are most congested. In this case, the bottleneck is on the D unit and the T address path.

```
;*      Resource Partition:
;*                                  A-side   B-side
;*      .L units                       0        0
;*      .S units                       0        1
;*      .D units                       2*       1
;*      .M units                       0        0
;*      .X cross paths                 1        0
;*      .T address paths               2*       1
;*      Long read paths                0        0
;*      Long write paths               0        0
;*      Logical  ops (.LS)             0        0     (.L or .S unit)
;*      Addition ops (.LSD)            1        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)               0        1
;*      Bound(.L .S .D .LS .LSD)       1        1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 4 iterations in parallel
...
;*        SINGLE SCHEDULED ITERATION
;*
;*        C26:
;*   0              LDW     .D1T1   *A5++,A4          ; |9|
;*   1              LDW     .D2T2   *B4++,B5          ; |9|
;*   2      [ B0]   BDEC    .S2     C26,B0            ; |8|
;*   3              NOP             3
;*   6              ADD     .L1X    B5,A4,A3          ; |9|
;*   7              STW     .D1T1   A3,*A6++          ; |9|
;*   8              ; BRANCHCC OCCURS {C26}           ; |8|
```

From the single scheduled iteration, it can be seen that the D units and T address paths are used for loads and stores, one D unit and one T address path for each. There are three memory access instructions but only two D units and two T address paths available on each cycle. Thus, the resource bound is two cycles. This means that in the steady state, at least two cycles are required per result.

Performance can be improved by better utilizing the D units and T address paths. Assume it is known that the number of iterations is always even. If the loop is unrolled 2x (so that the resulting loop contains two copies of the original loop body and executes half the number of iterations), the compiler could balance out the D units and T address paths and achieve better resource utilization.

The following diagrams show the concept of unrolling the loop for better (more balanced) utilization of the critical D unit resource (the situation would be analogous for the critical T address path). On the left side, four loop iterations are shown, as indicated by the double-arrows, producing eight results in four cycles. One D unit is unused every other cycle. The right side shows performance after unrolling the loop by 2x. Both D units are executing useful instructions in every cycle. Of course, the order of the loads and stores must be rearranged, but the compiler takes care of this.

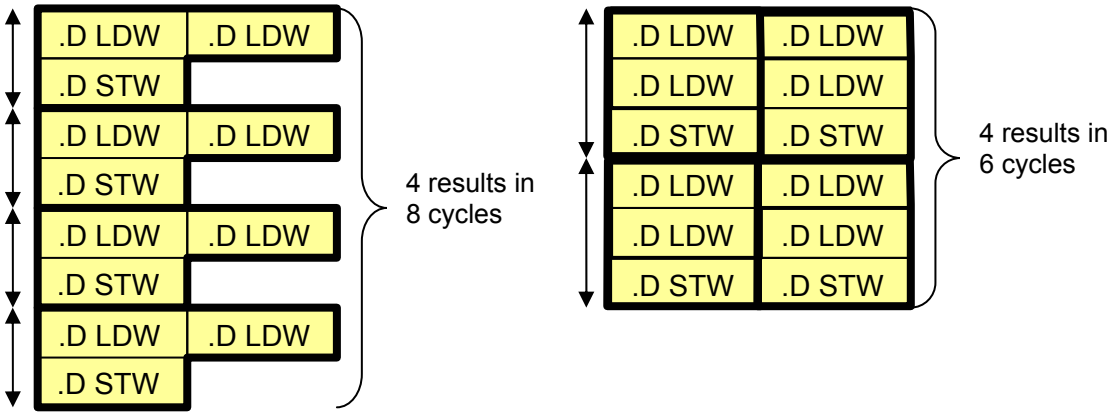

Figure 4. Loop Unrolling

One possibility for achieving this is to unroll the loop manually:

```
void BasicLoop(int *restrict output,
               int *restrict input1,
               int *restrict input2,
               int n)
{
    int i;
    #pragma MUST_ITERATE(1)
    for (i=0; i<n; i+=2) {
        output[i]   = input1[i]   + input2[i];
        output[i+1] = input1[i+1] + input2[i+1];
    }
}
```

Rebuilding yields the following results:

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 8
;*      Loop opening brace source line   : 8
;*      Loop closing brace source line   : 11
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
```

```
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1
;*      .D units                     3*       2
;*      .M units                     0        0
;*      .X cross paths               2        0
;*      .T address paths             3*       3*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          2        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1
;*      Bound(.L .S .D .LS .LSD)     2        1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 4 iterations in parallel
...
;*        SINGLE SCHEDULED ITERATION
;*
;*        C26:
;*   0              LDW     .D2T2   *B5++(8),B6       ; |10|
;*   1              NOP             1
;*   2              LDW     .D1T1   *A6++(8),A3       ; |10|
;*     ||           LDW     .D2T2   *-B5(4),B4        ; |10|
;*   3              LDW     .D1T1   *-A6(4),A3        ; |10|
;*   4              NOP             1
;*   5      [ B0]   BDEC    .S2     C26,B0            ; |8|
;*   6              NOP             1
;*   7              ADD     .S1X    B6,A3,A4          ; |10|
;*   8              ADD     .L1X    B4,A3,A5          ; |10|
;*   9              NOP             1
;*  10              STNDW   .D1T1   A5:A4,*A7++(8)    ; |10|
;*  11              ; BRANCHCC OCCURS {C26}           ; |8|
```

The T address paths are now balanced. There is one less D unit than expected because the compiler chose to use a non-aligned double word store instead of two aligned single-word stores. Recall, non-aligned memory accesses use both T address paths but only one D unit.

When the loop is unrolled 2x, each iteration takes longer, but the loop now generates two results per iteration instead of one. Thus, the unrolled loop requires 1.5 cycles/result in the steady state, whereas the non-unrolled version requires 2 cycles/result.

Although manually unrolling a loop achieves the required results, it can be rather tedious for a large loop. An alternative is to let the compiler do this. If the compiler knows that the trip count for the loop (in this case “n”) is a multiple of 2, the compiler unrolls the loop automatically, if deemed profitable. To tell the compiler that the trip count is a multiple of 2, modify the MUST_ITERATE pragma preceding the loop. The MUST_ITERATE pragma has the syntax:

```
#pragma MUST_ITERATE(lower_bound, upper_bound, factor)
```

The lower bound is the lowest possible value for “n”. The upper bound is maximum possible value for “n”. The factor is a number that always divides n. Any of these parameters can be omitted. [7]

[7] “`#pragma MUST_ITERATE(lower_bound)`” is equivalent to “`#pragma MUST_ITERATE(lower_bound,,)`”.

Instead of unrolling the loop manually, simply modify the pragma instead:

```
void BasicLoop(int *restrict output,
               int *restrict input1,
               int *restrict input2,
               int n)
{
    int i;
    #pragma MUST_ITERATE(2,,2)
    for (i=0; i<n; i++) {
        output[i] = input1[i] + input2[i];
    }
}
```

Now rebuild. The throughput is the same as when the loop was unrolled manually, namely 1.5 cycles/result. The extra line in the software pipelining information communicates that the compiler unrolled the loop 2x:

```
;*      Loop Unroll Multiple             : 2x
```

If the compiler had not deemed unrolling to be profitable (despite the use of the MUST_ITERATE pragma), one could force the compiler to unroll by inserting an UNROLL pragma in addition to the MUST_ITERATE pragma:

```
#pragma UNROLL(2)
```

Similarly, if the compiler chooses to unroll, and it is preferred not to have the loop unrolled, the loop can be preceded with

```
#pragma UNROLL(1)
```

The compiler usually succeeds at selecting the best unroll factor. Occasionally, however, one can do better by selecting the unrolling factor oneself. The reason is that the compiler must make an educated guess up front (during high-level optimization) as to how many times to unroll (if any). The actual software-pipelining is not done until low-level optimization. At this point, the unrolling decision is not reversible. In contrast, as a user, you have the opportunity to iteratively try out various unrolling factors and pick the best one.

Loop pragmas must appear immediately before a loop, without any other intervening source code instructions. Note that the compiler may ignore an UNROLL pragma, unless there is an accompanying MUST_ITERATE pragma. The MUST_ITERATE pragma must note that the trip count is divisible by the unroll factor and that the minimum trip count is at least the unroll factor.

When targeting the C64x+, *beware of over-unrolling*. Otherwise, the loop may become too large for the compiler to exploit the loop buffer (Section 2). The loop buffer has power, code size, and performance benefits, but can only be used with loops that have an ii of 14 or less and a single scheduled iteration length of 48 or less.

4.1.4 Exploiting Wide Loads and Stores (SIMD)

Although a speed up of 4.7x has already been achieved, it is possible to do better. Note that the loop is still bottlenecked on the memory access instructions. Since the memory access instructions are balanced, unrolling more will not help. However, two improvements can be made:

- Wider load instructions can be used instead of multiple loads to reduce the number of D unit resources.
- Aligned memory access instructions can be used instead of non-aligned memory access instructions to reduce the number of T address paths.

The C64x and C64x+ processors support both aligned and non-aligned double-word loads and stores.[8] If it is known that the function parameters are double-word aligned, switching to aligned, double-word memory accesses saves both D units and T address paths.

How does one get the compiler to select the double-word versions of the memory access instructions? There are two options: (1) use intrinsics, or (2) tell the compiler that the memory accesses are aligned. The second method is simpler, so it is best to try that first.

To tell the compiler that the memory accesses are aligned on double-word (64-bit) boundaries, use _nasserts() inside the function *just prior* to the loop of interest:[9]

```
_nassert((int) input1 % 8 == 0); // input1 is 64-bit aligned
_nassert((int) input2 % 8 == 0); // input2 is 64-bit aligned
_nassert((int) output % 8 == 0); // output is 64-bit aligned
```

If the data was not already aligned on a double-word boundary, it might be possible to force this alignment using the DATA_ALIGN pragma (reference [1]). _nasserts() make a statement about the value of variable at the point in the program where the _nassert() is located. From this information, the compiler can often derive the information about that variable at other locations in the program. For best performance, however, if the function contains multiple loops, it may be best to repeat the _nasserts() on entrance to each loop.

[8] Recall that the C67x and C67x+ support only aligned double-word loads and no double-word stores. The C62x supports only word-wide loads and stores.

[9] The _nassert() communicates that the pointer is aligned *at the point in the function where the _nassert() is located*. Although communicating alignment information once per function usually suffices, it is recommended to reassert the information immediately before each loop of interest.

Rebuild. The resource bound, and consequently the ii, have been reduced to 2:

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 13
;*      Loop opening brace source line   : 13
;*      Loop closing brace source line   : 15
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1
;*      .D units                     2*       1
;*      .M units                     0        0
;*      .X cross paths               2*       0
;*      .T address paths             2*       1
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          2        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1
;*      Bound(.L .S .D .LS .LSD)     2*       1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 4 iterations in parallel
...
;*        SINGLE SCHEDULED ITERATION
;*
;*        C26:
;*   0               LDDW    .D2T2   *B6++,B5:B4       ; |14|
;*     ||            LDDW    .D1T1   *A3++,A7:A6       ; |14|
;*   1               NOP             1
;*   2      [ B0]    BDEC    .S2     C26,B0            ; |13|
;*   3               NOP             2
;*   5               ADD     .S1X    B4,A6,A4          ; |14|
;*   6               ADD     .L1X    B5,A7,A5          ; |14|
;*   7               STDW    .D1T1   A5:A4,*A8++       ; |14|
;*   8               ; BRANCHCC OCCURS {C26}           ; |13|
```

In the steady state, the loop is now generating one result per cycle (more accurately, two results every two cycles), a 7x speedup, compared to the baseline.

4.1.5 Rebalancing Resources

Although the incorporation of wider loads and stores speeded up the loop, the loop is still bottlenecked on D units and T address paths. The number of memory accesses has been reduced to three. D units and the T address paths are again unbalanced. As before, further improvement can be achieved by unrolling one more time, if legal. Assuming that the trip count is indeed a multiple of 4, modify the MUST_ITERATE pragma to communicate this to the compiler. Rebuild. The resulting source code (which achieves optimal throughput) looks as follows:

```
void BasicLoop(int *restrict output,
               int *restrict input1,
               int *restrict input2,
               int n)
{
    int i;

    _nassert((int) input1 % 8 == 0); // input1 is 8-byte aligned
    _nassert((int) input2 % 8 == 0); // input2 is 8-byte aligned
    _nassert((int) output % 8 == 0); // output is 8-byte aligned

    #pragma MUST_ITERATE(4,,4)       // n >= 4, n % 4 = 0
    for (i=0; i<n; i++) {
        output[i]   = input1[i] + input2[i];
    }
}
```

The software-pipelining information for the resulting assembly code, which yields one result every 0.75 cycles (or 4 results every 3 cycles), is as follows:

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 13
;*      Loop opening brace source line   : 13
;*      Loop closing brace source line   : 15
;*      Loop Unroll Multiple             : 4x
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     1        0
;*      .D units                     3*       3*
;*      .M units                     0        0
;*      .X cross paths               2        2
;*      .T address paths             3*       3*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          2        2     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        0
;*      Bound(.L .S .D .LS .LSD)     2        2
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3   Schedule found with 3 iterations in parallel
...
```

```
;*        SINGLE SCHEDULED ITERATION
;*
;*        C26:
;*   0               LDDW    .D2T2   *B5++(16),B9:B8   ; |14|
;*     ||            LDDW    .D1T1   *A16++(16),A7:A6  ; |14|
;*   1               LDDW    .D2T2   *-B5(8),B7:B6     ; |14|
;*     ||            LDDW    .D1T1   *-A16(8),A9:A8    ; |14|
;*   2               NOP             1
;*   3      [ A0]    BDEC    .S1     C26,A0            ; |13|
;*   4               NOP             1
;*   5               ADD     .S1X    B9,A7,A5          ; |14|
;*   6               ADD     .L1X    B8,A6,A4          ; |14|
;*     ||            ADD     .L2X    B6,A8,B6          ; |14|
;*   7               ADD     .L2X    B7,A9,B7          ; |14|
;*   8               STDW    .D1T1   A5:A4,*A3++(16)   ; |14|
;*     ||            STDW    .D2T2   B7:B6,*++B4(16)   ; |14|
;*   9               ; BRANCHCC OCCURS {C26}           ; |13|
```

The loop has been sped up 9.3x compared to the original source code with no modifications other than the addition of restrict qualifiers, MUST_ITERATE pragmas and _nasserts().

For this example, only two of the three fields of the MUST_ITERATE pragma are exploited. The third (middle) field, which communicates an upper bound on the trip count, can also be useful for performance tuning. There are certain optimizations that are profitable when the loop trip count is large but can hurt performance when the trip count is small. By default, the compiler assumes that the expected trip count is large. Hence, if the upper bound on the trip count is small, it is best to communicate this to the compiler.

4.1.6 Using the Compiler Consultant

Now run the Compiler Consultant on the original version of this loop. Reference [3] explains how. Notice that the Compiler Consultant provides all the tuning suggestions covered in Section 4.1. While it is a useful exercise to read the preceding sections and follow the steps yourself, in most cases, it is not strictly necessary because the Compiler Consultant performs the analysis automatically and provides advice that matches the tuning steps performed thus far. This is not the case for most of the remaining information presented in this document.

Some, but not all, of this advice is also present in the *.nfo file.

4.2 Tuning Interruptible Applications

By default (except on the C64x+ when the loop buffer is used), software-pipelined loops cannot be safely interrupted. The compiler automatically disables interrupts before the software loop begins and reenables them when the loop completes. You can specify an upper limit on the number of cycles for which interrupts are permitted to be disabled. If the compiler cannot guarantee that the software-pipelined loop will execute for fewer cycles than this limit, it is forced to use an alternate method for software pipelining. This alternate method can incur a performance penalty.

When compiling code that must be interruptible, there are three options:

1. Use –mi<num>. The compiler ensures that interrupts are not disabled for more than <num> cycles. The alternate (potentially less efficient) software pipelining method is used for loops for which the compiler cannot guarantee that interrupts will not be disabled prohibitively long.

2. Do not use a –mi option (default). The compiler will always use the default (more efficient) software pipelining method, disabling interrupts around software-pipelined loops. Interrupts will be disabled for the duration of the software-pipelined loop, regardless of the number of cycles that the loop takes to complete.

3. Use -mi (no parameter). The compiler will always use the default software pipelining method. The compiler will *not* disable interrupts around software-pipelined loops. You are responsible for manually disabling and reenabling interrupts where needed. You can choose to control interrupts on entry/exit to the function or at whatever other granularity is safe and works best for the application.

Approaches 2) and 3) can only be used with applications that are flexible as to the duration for which interrupts are disabled.

With approach 1), if the compiler knows the upper bound on a loop's trip count, it can compute the maximum number of cycles for which interrupts would be disabled. If this number is less than <num>, then the compiler can use the default (preferred) method for software pipelining the loop and simply disable interrupts around that loop. However, if the compiler knows nothing about the upper bound on a loop's trip count, it will be forced to use the alternate method. Thus, using the MUST_ITERATE pragma to communicate upper bounds on loop trip counts can be especially valuable when compiling code using approach 1).

On the C64x+, most software-pipelined loops fit into the loop buffer. The compiler automatically exploits the loop buffer whenever possible. Loops that use the loop buffer are always interruptible. Hence, the performance and code size hit is generally averted. The biggest barrier to exploiting the loop buffer is usually the size of the loop body. If the loop buffer is not being exploited, check whether ii is greater than14 or the single scheduled iteration length is greater than 48. If so, see if the loop can be split into two smaller loops.

On rare occasions, the loop buffer is not exploited because the version of the software-pipelined loop that does not exploit the loop buffer performs better. To encourage the compiler to use the loop buffer (that is, to favor code size or power over performance), compile with –ms0 or –ms1 (Section 3.1). Be aware that this causes other code size optimizations to be invoked as well.

For more information on interrupts or the interrupt option, see references [1] and [4].

4.3 Handling Nested Loops

The compiler is most aggressive on innermost loops. Usually, this is where most of the time is spent. Thus, the performance of enclosing loops is typically much less important. For some loop nests, however, the inner loop does not execute many times, while the enclosing loop does, as in the following two-loop nest:

Original Nested Loop

```
#pragma MUST_ITERATE(1000)     // outer loop: trip count >= 1000
for (i=0; i<large_value; i++)
{
    #pragma MUST_ITERATE(1,4)  // inner loop: 1 <= trip count <= 4
    for (j=0; j<small_value; j++)
    {
        <stuff for iter i,j>
    }
}
```

For this case, optimization of the outer loop is very important to overall performance.

Several strategies can improve performance of the outer loop. First, if the trip count of the inner loop is a known small constant and the body of the loop is short, the inner loop can be unrolled completely, either manually or via pragmas (as explained in Section 4.1.3). Suppose the trip count for the inner loop is known to be 4. Then the following are two methods for completely unrolling the inner loop:

Hand-Tuned Loop After Complete Unrolling

```
#pragma MUST_ITERATE(1000,,)
for (i=0; i<large_value; i++)
{
   // small_value known to be 4
    <stuff for original iter i,0>
    <stuff for original iter i,1>
    <stuff for original iter i,2>
    <stuff for original iter i,3>
}
```

Hand-Tuned Loop After Forcing Automatic Unrolling

```
#pragma MUST_ITERATE(1000)
for (i=0; i<large_value; i++)
{
    // small_value known to be 4
    #pragma MUST_ITERATE(4,4,4)
    #pragma UNROLL(4)
    for (j=0; j<small_value; j++)
    {
        <stuff for original iter i,j>
    }
}
```

In most cases, however, this is unnecessary. If the loop body is short and the trip count is known to be a small constant, then the compiler does this automatically.

If the inner loop trip count is not constant or the loop body is too large, then the inner and outer loops can be *interchanged* when safe, as shown here:

Hand-Tuned Loop After Interchanging Inner and Outer Loops

```
#pragma MUST_ITERATE(1,4)      // 1 <= trip count <= 4
for (j=0; j<small_value; j++)
{
    #pragma MUST_ITERATE(1000) // trip count >= 1000
    for (i=0; i<large_value; i++)
    {
        <stuff for original iter i,j>
    }
}
```

Yet another option is to *coalesce* the two loops (that is, to merge the two loops into a single loop). This approach works best when the body of the outer loop (excluding the inner loop) is very short.

Loop Coalescing

```
i = j = 0;
#pragma MUST_ITERATE(1000,4000)   // 1000 <= trip count <= 4000
for (ij=0; ij<large_value*small_value; ij++)
{
    <stuff for iter i,j>

    if (++j == small_value)
    {
        i = i + 1;
        j = 0;
    }
}
j == small_value                  // omit if j is not used again
```

Beware that this approach can sometimes create a large loop-carried dependency cycle on the loop control variables “i” and “j”. If the loop-carried dependency bound does become a bottleneck, try a different optimization technique.

4.4 Using Intrinsics to Tune Software-Pipelined Loops

In most cases, exploitation of the annotations presented in the previous section suffices for achieving good performance. However, there may be a few cases where it might be necessary to revert to *intrinsics* (built-in functions that usually map to specific assembly instructions).

4.4.1 Casting Between Types

When writing code, beware of casting between types. To load and store data of a different width than the native data type, use the memory access intrinsics. *Do not cast pointers to accomplish this.* Casting pointers is unsafe because (in general) the compiler can assume that pointers of different types do not conflict. For more information, see the ANSI C/C++ Standard.

Bad

```
int    *p;
short  *q;
...
double *dp = p;
double *dq = q
...
for (i=0; i<n; i++)
    dq[i] = dp[i];
```

Good

```
int   *p
short *q;
...
for (i=0; i<n; i++)
    _memd8(&q[4*i]) = _memd8(&p[2*i]);
```

Unlike other intrinsics, memory access intrinsics can be used on either the right or left-hand side of assignments.

There are versions of the intrinsics for aligned (_amem*()) and unaligned (_mem*()) memory accesses. There are corresponding intrinsics for const pointers (_amem*_const() and _mem*_const()). At present, memory access intrinsics cannot be used with volatile pointers.

Prior to CCStudio 3.2 (compiler version 6.0.1), a double-word (64-bit quantity) must be represented using a double. The high and low halves of a double-word are extracted using _hi() and _lo(). These intrinsics copy the appropriate word without conversion. Beginning with compiler version 6.0.1, a long long type is supported that can optionally be used instead.

The following examples perform a non-aligned double-word load and then extract the high and low words:

Supported by All C6x Compiler Versions

```
char *p;
double d = _memd8((void *) p);
int hi_p = _hi(d);
int lo_p = _lo(d);
```

Supported by Version 6.0.1 and Higher

```
char *p;
long long d = _mem8((void *) p);
int hi_p    = _hill(d);
int lo_p    = _loll(d);
```

To combine two words into a double-word, using the following:

Supported by All C6x Compiler Versions

```
char *p;
int   hi          = ...
int   lo          = ...
_memd8((void *) p) = _itod(hi, lo);
```

Supported by Version 6.0.1 and Higher

```
char *p;
int   hi          = ...
int   lo          = ...
_mem8((void *) p) = _itoll(hi, lo);
```

Be careful to use _hi(), _lo() and _itod() when working with doubles. Use _hill(), _loll() and _itoll() when working with long long types. Otherwise, the compiler inserts unexpected (and unwanted) calls to the runtime support library to convert between doubles and long longs or vice versa. See references [1] and [4] for more detail.

4.4.2 Example Using Intrinsics

This section walks through the steps for generating an intrinsic version of the demux() function that follows. The input buffer “ib” contains interleaved bytes in the order cr, y, cb, y, cr, y, cb, y, etc. The demux() function separates the cr, y, and cb components into output buffers with those names. Note that this function has already been annotated as suggested in Section 4.1.

```
void demux (const char * restrict ib,
            char       * restrict y,
            char       * restrict cr,
            char       * restrict cb,
            int          input_size)
{
    int i;
    _nassert((int) ib % 8 == 0);
    _nassert((int) y  % 8 == 0);
    _nassert((int) cr % 8 == 0);
    _nassert((int) cb % 8 == 0);

    #pragma MUST_ITERATE(4,,4)
    for (i = 0; i < input_size/4; i++) {
        cr[i]    = ib[4*i];
        y[2*i]   = ib[4*i + 1];
        cb[i]    = ib[4*i + 2];
        y[2*i+1] = ib[4*i + 3];
    }
}
```

4.4.2.1 Preliminary Analysis

Compile this function with –o –s –mw –mv6400, which is implicitly little-endian. As can be seen from the optimizer comments, the result is a loop that has been unrolled 4x. There are 16 loads and 16 stores.

```
_demux:
;** 6   -----------------------    ib = ib;
;** 6   -----------------------    y  = y;
;** 6   -----------------------    cr = cr;
;** 6   -----------------------    cb = cb;
;** 15  -----------------------    // LOOP BELOW UNROLLED BY FACTOR(4)
;**     -----------------------    U$16 = &ib[-16];
;**     -----------------------    U$18 = &cr[-4];
;**     -----------------------    U$24 = &y[-8];
;**     -----------------------    U$28 = &cb[-4];
;** 15  -----------------------    L$1 = input_size>>4;
;**     -----------------------    #pragma MUST_ITERATE(1, 134217727, 1)
. . .
;**     -----------------------g2:
;** 15  -----------------------    *(U$18 += 4) = *(U$16 += 16);
;** 16  -----------------------    *(U$24 += 8) = U$16[1];
;** 17  -----------------------    *(U$28 += 4) = U$16[2];
;** 18  -----------------------    U$24[1] = U$16[3];
;** 15  -----------------------    U$18[1] = U$16[4];
;** 16  -----------------------    U$24[2] = U$16[5];
;** 17  -----------------------    U$28[1] = U$16[6];
;** 18  -----------------------    U$24[3] = U$16[7];
;** 15  -----------------------    U$18[2] = U$16[8];
;** 16  -----------------------    U$24[4] = U$16[9];
;** 17  -----------------------    U$28[2] = U$16[10];
;** 18  -----------------------    U$24[5] = U$16[11];
;** 15  -----------------------    U$18[3] = U$16[12];
;** 16  -----------------------    U$24[6] = U$16[13];
;** 17  -----------------------    U$28[3] = U$16[14];
;** 18  -----------------------    U$24[7] = U$16[15];
;** 14  -----------------------    if ( --L$1 ) goto g2;
;**     -----------------------    return;
```

The software-pipelined loop information is shown in the following example. Note that the loop is bottlenecked on D units and T address paths. Resources are balanced, but none of the loads or stores have been SIMDed (coalesced into wider memory accesses).

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 14
;*      Loop opening brace source line   : 14
;*      Loop closing brace source line   : 19
;*      Loop Unroll Multiple             : 4x
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 1
;*      Unpartitioned Resource Bound     : 16
;*      Partitioned Resource Bound(*)    : 16
```

```
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                    0        0
;*      .S units                    1        0
;*      .D units                   16*      16*
;*      .M units                    0        0
;*      .X cross paths              0        2
;*      .T address paths           16*      16*
;*      Long read paths             0        0
;*      Long write paths            0        0
;*      Logical  ops (.LS)          0        0     (.L or .S unit)
;*      Addition ops (.LSD)         0        2     (.L or .S or .D unit)
;*      Bound(.L .S .LS)            1        0
;*      Bound(.L .S .D .LS .LSD)    6        6
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 16 Schedule found with 2 iterations in parallel
...
;*        C37:
;*   0              LDB     .D1T1   *++A5(16),A4      ; |15|  ^
;*   1              NOP             4
;*   5              MV      .L2X    A4,B4             ; |15|  ^ Define a twin register
;*   6              STB     .D2T2   B4,*++B8(4)       ; |15|  ^
;*   7              LDB     .D1T2   *+A5(2),B9        ; |17|  ^
;*     ||           LDB     .D2T1   *++B5(16),A4      ; |18|  ^
;*   8              LDB     .D2T2   *+B5(1),B16       ; |15|  ^
;*     ||           LDB     .D1T1   *+A5(14),A6       ; |17|  ^
;*   9              LDB     .D1T1   *+A5(1),A4        ; |16|  ^
;*     ||           LDB     .D2T2   *+B5(9),B9        ; |15|  ^
;*  10              LDB     .D1T1   *+A5(7),A9        ; |18|  ^
;*     ||           LDB     .D2T2   *+B5(5),B17       ; |15|  ^
;*  11              LDB     .D2T2   *+B5(3),B4        ; |17|  ^
;*     ||           LDB     .D1T1   *+A5(9),A8        ; |16|  ^
;*  12              MV      .L2X    A4,B4             ; |18|  ^ Define a twin register
;*     ||           LDB     .D2T2   *+B5(2),B17       ; |16|  ^
;*     ||           LDB     .D1T1   *+A5(11),A7       ; |18|  ^
;*  13              STB     .D2T2   B9,*++B7(4)       ; |17|  ^
;*     ||           LDB     .D1T1   *+A5(13),A6       ; |16|  ^
;*  14              LDB     .D1T2   *+A5(10),B4       ; |17|  ^
;*     ||           STB     .D2T1   A6,*+B7(3)        ; |17|  ^
;*  15              STB     .D2T2   B4,*++B6(8)       ; |18|  ^
;*     ||           LDB     .D1T1   *+A5(15),A8       ; |18|  ^
;*  16              STB     .D2T2   B17,*+B8(2)       ; |15|  ^
;*  17              STB     .D1T1   A4,*++A3(8)       ; |16|  ^
;*     ||           STB     .D2T2   B4,*+B7(1)        ; |17|  ^
;*     ||   [ A0]   BDEC    .S1     C37,A0            ; |14|
;*  18              STB     .D2T2   B17,*+B6(1)       ; |16|  ^
;*     ||           STB     .D1T1   A9,*+A3(3)        ; |18|  ^
;*  19              STB     .D1T1   A8,*+A3(4)        ; |16|  ^
;*     ||           STB     .D2T2   B9,*+B8(3)        ; |15|  ^
;*  20              STB     .D2T2   B4,*+B7(2)        ; |17|  ^
;*     ||           STB     .D1T1   A7,*+A3(5)        ; |18|  ^
;*  21              STB     .D2T2   B16,*+B8(1)       ; |15|  ^
;*     ||           STB     .D1T1   A6,*+A3(6)        ; |16|  ^
;*  22              STB     .D1T1   A8,*+A3(7)        ; |18|  ^
;*  23              ; BRANCHCC OCCURS {C37}           ; |14|
```

To achieve better performance, it is necessary to exploit wider loads and stores. Since all arrays are aligned, aligned memory-access intrinsics can be used. The goal is to minimize the number of memory accesses while minimizing the size of the loop. On the C64x and C64x+, loads and stores can have widths that vary between 1 and 8 bytes.

4.4.2.2 Selecting an Unroll Factor

How much should the loop be unrolled? 2x, 4x, 8x, not at all? Consider each of these, making the simplifying assumption that loads and stores are the only critical resource. The following table shows the number of load and store operations per loop, depending on the unroll factor. The load and store instructions can work with different data sizes, and are used as indicated by the B (byte), H (half-word, 2 bytes), W (word, 4 bytes), DW (double-word, 8 bytes) designators in the table.

Unroll factor	Loads of "ib"	Stores to "y"	Stores to "cr"	Stores to "cb"	Total Memory Accesses	Lower Bound on ii	Minimum Cycles per Results
1x	1 W	1 HW	1 B	1 B	4	2	2
2x	1 DW	1 W	1 HW	1 HW	4	2	1
4x	2 DW	1 DW	1 W	1 W	5	3	0.75
8x	4 DW	2 DW	1 DW	1 DW	8	4	0.5
16x	8 DW	4 DW	2 DW	2 DW	16	8	0.5

Based on this simplistic analysis, 8x is the optimal unroll factor. 4x yields a loop with more cycles/result. 16x yields a larger loop with no reduction in cycles/result. Depending on our tolerance for code growth, we might choose a smaller unroll factor and sacrifice some CPU performance for smaller code size.

In practice, there are other considerations that are not evident from this table. One should probably go through the exercise of testing out several unroll factors. The decision to adjust an unroll factor would be based on resource constraints as described in Section 4.1. For brevity, just one unroll factor is tested here, and that is 4x. When other resources are considered, this unroll factor turns out to be the best.

Manually unroll the loop 4x and insert the memory access instructions:

```
    /*************************************************************************/
    /* LOOP UNROLLED 4x                                                      */
    /*************************************************************************/
    for (i = 0; i < input_size/4; i+=4)
    {
        /*********************************************************************/
        /* READ IN THE DATA.   LITTLE ENDIAN ASSUMED.                        */
        /*********************************************************************/
        /*-------------------------------------------------------------------*/
        /* ib_3_0   (byte j) = ib_7_0 (byte j)   = ib[4i+j]                  */
        /* ib_7_4   (byte j) = ib_7_0 (byte 4+j) = ib[4i+4+j]                */
        /* j = 0,..,3                                                        */
        /*-------------------------------------------------------------------*/
        double ib_7_0    = _amemd8((void *) &ib[4*i]);
        int    ib_3_0    = _lo(ib_7_0);                          // < 3,2,1,0>
        int    ib_7_4    = _hi(ib_7_0);                          // < 7,6,5,4>
```

```
        /*--------------------------------------------------------------*/
        /* ib_11_8  (byte j) = ib_15_8 (byte j)   = ib[4i+8+j]          */
        /* ib_15_12 (byte j) = ib_15_8 (byte 4+j) = ib[4i+12+j]         */
        /* j = 0,..,3                                                   */
        /*--------------------------------------------------------------*/
        double ib_15_8    = _amemd8((void *) &ib[4*i+8]);
        int    ib_11_8    = _lo(ib_15_8);                  // <11,10, 9, 8>
        int    ib_15_12   = _hi(ib_15_8);                  // <15,14,13,12>

        /**************************************************************/
        /* PACK DATA AND STORE.                                       */
        /**************************************************************/
        /*--------------------------------------------------------------*/
        /* cr[i]          = ib[4i]                                      */
        /* cr[i+1]        = ib[4i+4]                                    */
        /* cr[i+2]        = ib[4i+8]                                    */
        /* cr[i+3]        = ib[4i+12]                                   */
        /*--------------------------------------------------------------*/
        _amem4(&cr[i])    = ...                             // <12, 8, 4, 0>

        /*--------------------------------------------------------------*/
        /* cb[i]          = ib[4i+2]                                    */
        /* cb[i+1]        = ib[4i+6]                                    */
        /* cb[i+2]        = ib[4i+10]                                   */
        /* cb[i+3]        = ib[4i+14]                                   */
        /*--------------------------------------------------------------*/
        _amem4(&cb[i])    = ...                             // <14,10, 6, 2>

        /*--------------------------------------------------------------*/
        /* y[2*i]         = ib[4i+1]                                    */
        /* y[2i+1]        = ib[4i+3]                                    */
        /* y[2i+2]        = ib[4i+5]                                    */
        /* y[2i+3]        = ib[4i+7]                                    */
        /*                                                              */
        /* y[2i+4]        = ib[4i+9]                                    */
        /* y[2i+5]        = ib[4i+11]                                   */
        /* y[2i+6]        = ib[4i+13]                                   */
        /* y[2i+7]        = ib[4i+15]                                   */
        /*--------------------------------------------------------------*/
        _amem8(&y[2*i])   = ..                  // <15,13,11, 9, 7, 5, 3, 1>
    }
```

Note that in compiler version 6.0.1 and higher, the 64-bit fixed-point data type “long long” would have been more natural in this example than double. The appropriate long long intrinsics are described in Section 4.4.1.

4.4.2.3 Packing Data

To exploit the wider stores, the input data “ib” must be rearranged so that they are in the proper output order. What is the optimal sequence of pack instructions to accomplish this?

First, consider the 4-byte store to _amem4(&cr[i]):

```
cr[i]   = ib[4i]    = ib_3_0      (byte 0)
cr[i+1  = ib[4i+4]  = ib_7_4      (byte 0)
cr[i+2] = ib[4i+8]  = ib_11_8     (byte 0)
cr[i+3] = ib[4i+12] = ib_15_12    (byte 0)
```

Note that this amounts to packing the lower byte from each of four words (ib_3_0, ib_7_4, ib_11_8 and ib_15_12) into a single word. This can be done using _pack2() and pack4() intrinsics. These two intrinsics translate semantically to the PACK2 and PACK4 instructions, respectively.

The PACK2 instruction packs the low half of each of two source words into a single word:

```
PACK2(<x3,x2,x1,x0>, <y3,y2,y1,y0>) = <x1,x0,y1,y0>
```

The PACKL4 instruction extracts the even bytes from each of two source words:

```
PACKL4(<x3,x2,x1,x0>, <y3,y2,y1,y0>) = <x2,x0,y2,y0>
```

They can be used in combination to extract the lower byte from each of four words:

```
PACK2 (< ib7, ib6, ib5, ib4>, < ib3, ib2, ib1, ib0>) = < ib5, ib4, ib1 ,ib0>
PACK2 (<ib15,ib14,ib13,ib12>, <ib11,ib10, ib9, ib8>) = <ib13,ib12, ib9, ib8>
PACKL4(<ib13,ib12, ib9, ib8>, < ib5, ib4, ib1 ,ib0>) = <ib12, ib8, ib4, ib0>
```

The intrinsic source code to accomplish this is as follows:

```
ib_5_4_1_0        = _pack2 (ib_7_4,        ib_3_0);      // < 5, 4, 1, 0>
ib_13_12_9_8      = _pack2 (ib_15_12,      ib_11_8);     // <13,12, 9, 8>
_amem4(&cr[i])    = _packl4(ib_13_12_9_8,  ib_5_4_1_0);  // <12, 8, 4, 0>
```

Next, consider the 4-byte store to _amem(&cb[i]). This case is similar to the previous one except that byte 2 of each source word is needed instead of byte 0.

```
cb[i]   = ib[4i+2   = ib_3_0   (byte 2)
cb[i+1  = ib[4i+6]  = ib_7_4   (byte 2)
cb[i+2] = ib[4i+10] = ib_11_8  (byte 2)
cb[i+3] = ib[4i+14] = ib_15_12 (byte 2)
```

This can be done by using PACKH2 instead of PACK2. Instruction PACKH2 extracts the high half of a source word instead of the low half.

```
PACKH2(< ib7, ib6, ib5, ib4>, < ib3, ib2, ib1, ib0>) = < ib7, ib6, ib3 ,ib2>
PACKH2(<ib15,ib14,ib13,ib12>, <ib11,ib10, ib9, ib8>) = <ib15,ib14,ib11,ib10>
PACKL4(<ib15,ib14,ib11,ib10>, < ib7, ib6, ib3 ,ib2>) = <ib14,ib10, ib6, ib2>
```

The resulting intrinsic code sequence is:

```
ib_7_6_3_2        = _packh2(ib_7_4,        ib_3_0);      // < 7, 6, 3, 2>
ib_15_14_11_10    = _packh2(ib_15_12,      ib_11_8);     // <15,14,11,10>
_amem4(&cb[i])    = _packl4(ib_15_14_11_10,ib_7_6_3_2);  // <14,10, 6, 2>
```

Finally, pack the data for the 8-byte store to _amemd8(&y[2*i]):

```
y[2*i]  = ib[4i+1]  = ib_3_0   (byte 1)
y[2i+]  = ib[4i+3]  = ib_3_0   (byte 3)
y[2i+2] = ib[4i+5]  = ib_7_4   (byte 1)
y[2i+3] = ib[4i+7]  = ib_7_4   (byte 3)

y[2i+4] = ib[4i+9]  = ib_11_8  (byte 1)
y[2i+5] = ib[4i+11] = ib_11_8  (byte 3)
y[2i+6] = ib[4i+13] = ib_15_12 (byte 1)
y[2i+7] = ib[4i+15] = ib_15_12 (byte 3)
```

Extract the high bytes using PACKH4 instructions:

```
PACKH4(< ib7, ib6, ib5, ib4>, < ib3, ib2, ib1, ib0>) = < ib7, ib5, ib3, ib1>
PACKH4(<ib15,ib14,ib13,ib12>, <ib11,ib10, ib9, ib8>) = <ib15,ib13,ib11, ib9>
```

Pack together the two 4-byte quantities into an 8-byte quantity. This can be done using _itod(). The resulting intrinsic code sequence is:

```
ib_7_5_3_1       = _packh4(ib_7_4,   ib_3_0);        // < 7, 5, 3, 1>
ib_15_13_11_9    = _packh4(ib_15_12, ib_11_8);       // <15,13,11, 9>
_amemd8(&y[2*i]) = _itod(ib_15_13_11_9, ib_7_5_3_1); // <15,13,11, 9, 7, 5, 3, 1>
```

4.4.2.4 Final Results

The complete source code for the intrinsic version of the demux() is as follows:

```
void demux (const char * restrict ib,
                  char * restrict y,
                  char * restrict cr,
                  char * restrict cb,
                  int    input_size)
{
    int i;

    _nassert((int) ib % 8 == 0);
    _nassert((int) y  % 8 == 0);
    _nassert((int) cr % 8 == 0);
    _nassert((int) cb % 8 == 0);


/*************************************************************************/
/* LOOP UNROLLED 4x                                                      */
/*************************************************************************/
#pragma MUST_ITERATE(1,,1)
for (i = 0; i < input_size/4; i+=4)
{
    int ib_5_4_1_0;
    int ib_13_12_9_8;
    int ib_7_6_3_2;
    int ib_15_14_11_10;
    int ib_7_5_3_1;
    int ib_15_13_11_9;
```

```
        /*************************************************************************/
        /* READ IN THE DATA. LITTLE ENDIAN ASSUMED.                              */
        /*************************************************************************/
        /*-----------------------------------------------------------------------*/
        /* ib_3_0   (byte j) = ib_7_0 (byte j)   = ib[4i+j]                      */
        /* ib_7_4   (byte j) = ib_7_0 (byte 4+j) = ib[4i+4+j]                    */
        /* j = 0,..,3                                                            */
        /*-----------------------------------------------------------------------*/
        double ib_7_0    = _amemd8((void *) &ib[4*i]);
        int    ib_3_0    = _lo(ib_7_0);                          // < 3,2,1,0>
        int    ib_7_4    = _hi (ib_7_0);                         // < 7,6,5,4>

        /*-----------------------------------------------------------------------*/
        /* ib_11_8  (byte j) = ib_15_8 (byte j)   = ib[4i+8+j]                   */
        /* ib_15_12 (byte j) = ib_15_8 (byte 4+j) = ib[4i+12+j]                  */
        /* j = 0,..,3                                                            */
        /*-----------------------------------------------------------------------*/
        double ib_15_8    = _amemd8((void *) &ib[4*i+8]);
        int    ib_11_8    = _lo(ib_15_8);                         // <11,10, 9, 8>
        int    ib_15_12   = _hi(ib_15_8);                         // <15,14,13,12>

        /*************************************************************************/
        /* PACK DATA AND STORE.                                                  */
        /*************************************************************************/
        /*-----------------------------------------------------------------------*/
        /* cr[i]          = ib[4i]    = ib_3_0   (byte 0)                        */
        /* cr[i+1]        = ib[4i+4]  = ib_7_4   (byte 0)                        */
        /* cr[i+2]        = ib[4i+8]  = ib_11_8  (byte 0)                        */
        /* cr[i+3]        = ib[4i+12] = ib_15_12 (byte 0)                        */
        /*-----------------------------------------------------------------------*/
        ib_5_4_1_0       = _pack2 (ib_7_4,        ib_3_0);      // < 5, 4, 1, 0>
        ib_13_12_9_8     = _pack2 (ib_15_12,      ib_11_8);     // <13,12, 9, 8>
        _amem4(&cr[i])   = _packl4(ib_13_12_9_8,  ib_5_4_1_0);  // <12, 8, 4, 0>

        /*-----------------------------------------------------------------------*/
        /* cb[i]          = ib[4i+2]  = ib_3_0   (byte 2)                        */
        /* cb[i+1]        = ib[4i+6]  = ib_7_4   (byte 2)                        */
        /* cb[i+2]        = ib[4i+10] = ib_11_8  (byte 2)                        */
        /* cb[i+3]        = ib[4i+14] = ib_15_12 (byte 2)                        */
        /*-----------------------------------------------------------------------*/
        ib_7_6_3_2       = _packh2(ib_7_4,        ib_3_0);      // < 7, 6, 3, 2>
        ib_15_14_11_10   = _packh2(ib_15_12,      ib_11_8);     // <15,14,11,10>
        _amem4(&cb[i])   = _packl4(ib_15_14_11_10,ib_7_6_3_2);  // <14,10, 6, 2>

        /*-----------------------------------------------------------------------*/
        /* y[2*i]  = ib[4i+1]  = ib_3_0   (byte 1)                               */
        /* y[2i+1] = ib[4i+3]  = ib_3_0   (byte 3)                               */
        /* y[2i+2] = ib[4i+5]  = ib_7_4   (byte 1)                               */
        /* y[2i+3] = ib[4i+7]  = ib_7_4   (byte 3)                               */
        /*                                                                       */
        /* y[2i+4] = ib[4i+9]  = ib_11_8  (byte 1)                               */
        /* y[2i+5] = ib[4i+11] = ib_11_8  (byte 3)                               */
        /* y[2i+6] = ib[4i+13] = ib_15_12 (byte 1)                               */
        /* y[2i+7] = ib[4i+15] = ib_15_12 (byte 3)                               */
        /*-----------------------------------------------------------------------*/
        ib_7_5_3_1       = _packh4(ib_7_4,   ib_3_0);         // < 7, 5, 3, 1>
        ib_15_13_11_9    = _packh4(ib_15_12, ib_11_8);        // <15,13,11, 9>
        _amemd8(&y[2*i]) = _itod(ib_15_13_11_9, ib_7_5_3_1);  // <15,13,11, 9,7,5,3,1>
    }
}
```

Performance and code size results are summarized in the following table:

Source Code Version	Compiler Version	Performance Oriented Options	ii	Cycles/Result	Size (Bytes)
Original	5.1.3	–o –mv6400	16	4	340
Intrinsic	5.1.3	–o –mv6400	4	1	408
Intrinsic	5.1.3	–o –mv6400 –mh48	4	1	160
Intrinsic	6.0.3	–o –mv64+	3	0.75	116

On the C64x, the loop does not achieve the ii that was originally estimated. The reason is that the compiler is inserting a few extra moves. To reduce the ii from 4 to 3, one would need to code this loop in linear assembly. Despite this, the loop still achieves a speedup of 4x (400%) over the original version.

The software pipelining information for the intrinsic version of the function recommends compiling with –mh48 (see Section 3.1 for an explanation of this option). The third row of the table shows the dramatic decrease in code size when compiling with this option − a 61% decrease over the same code version compiled without –mh. It is often the case that compiling with –mh reduces code size.[10]

The fourth row shows the performance and code size if one were to compile instead for the C64x+ (using compiler version 6.0.3). Both speed and size is significantly improved. The reason is that the intrinsic version of this function is small enough to fit into the loop buffer, which is supported on C64x+ only. Recall that a loop must have an ii ≤14 to fit into the loop buffer. Thus, the tuned loop fits into the loop buffer, whereas the original loop does not. Note that the C64x+ version of the loop does indeed achieve optimal throughput.

As mentioned in Section 2.1, the loop buffer improves performance in two cases: when a loop is resource bound and when a loop must be interruptible. This is an example of the first case.

For additional examples of manually selecting instructions, see reference [4].

5 Optimizing Control Code

This section focuses on optimization of linked structures, “if” statements, and calls, common features in what is often termed “control code”.

5.1 Restrict Qualifying Pointers Embedded in Structures

While software-pipelined loops typically manipulate array-based data structures, control code is commonly laden with references to complex structures that are linked together in some manner. Hence, a discussion of control code optimization would not be complete without addressing these types of data structures.

[10] The compiler can often automatically eliminate part or all of the pipe up/pipe down code of a software-pipelined loop by speculating (over-executing) some of the loop instructions (when safe). The –mh option is required when loads must be speculated, which is quite common in practice.

5.1.1 Basic Rules

The ANSI C standard supports restrict qualifiers on structure members in structure definitions. At present (compiler tools version 6.0.x), the compiler allows restrict qualifiers on structure members but does *not* exploit them.[11] To communicate to the compiler that a pointer-based structure member does not overlap with other pointers, create a restrict-qualified local pointer at the top level of the function and assign the structure member to it. Then use the local pointer in the function instead of the structure member. This is shown in the following example:

```
myfunc(myStr *s)
{
    myStr *t;

    // declare local pointers at top-level of function
    int * restrict p;
    int * restrict v;

    ...

    // assign to p and v
    p = s->q->p;
    v = t->u->v;

    // use p and v instead of s->q->p and t->u->v
    *p = ...
    *v = ...
        = *p;
        = *v;
}
```

When possible, also for performance reasons, avoid dereferencing a non-restrict-qualified pointer inside loop control (for example, as part of a loop termination condition) or inside a loop body. Instead, create and use a local copy of a pointer or variable when possible. Non-restrict-qualified local pointers do not need to be declared in the top-level of the function.

```
if (...)
{
    int    y  = g->q->y;
    short *a  = g->p->a;

    while (y < 25)
    {
        a[i++] = ...
    }
}
```

[11] Putting restrict qualifiers on structure members is still a good idea for two reasons. First, this documents the original developer's assumptions. Second, this may be exploited by the compiler at some point in the future.

5.1.2 Identifying Cases When Copies of Pointers are Needed

Creating restrict-qualified locals is an important optimization both inside and outside loops. Consider the following test case. Assume all loads and stores to s->data->p[i], s->data->q[i] and s->data->sz are independent (that is, they do not overlap). Compile with -s –mw –o –mv6400:

```
typedef struct
{
   int *p, *q, sz;
} myData;

typedef struct
{
   myData *data;
} myStr;

LoopWithStructs(myStr * restrict s)
{
   int i;

   #pragma MUST_ITERATE(2,,2)
   for (i=0; i<s->data->sz; i++)
       s->data->q[i] = s->data->p[i];
}
```

Extract the optimizer comments. They should look as follows:

```
;** 12  -----------------------    s = s;
;**     -----------------------    V$0 = (*s).data;
;** 16  -----------------------    i = 0;
;**     -----------------------    #pragma MUST_ITERATE(2, 4294967294, 2)
...
;**     -----------------------g2:
;** 17  -----------------------    *(i*4+(*V$0).q) = *(i*4+(*V$0).p);
;** 16  -----------------------    if ( (*V$0).sz > (++i) ) goto g2;
;**     -----------------------    return;
```

Even though s is restrict-qualified, s->data, s->data->p and s->data->q are not. In other words, restrict qualification is not transitive. The consequence is that the compiler cannot tell that references through these pointers are independent. This causes the following inefficiencies:

- The compiler must dereference the pointers that are structure members each time they are used.
- The compiler must assume that all load-store or store-store pairs can potentially overlap.
- The compiler does not know that the termination condition (s->data->sz) is a constant. Thus it must recheck the value of this structure member during each loop iteration.
- The compiler cannot exploit wider loads and stores that improve memory system performance.

This leads to inefficient code. As shown in the software-pipelined information that follows, there are sufficient resources to allow a result to be computed every 3 cycles. However, the compiler's inability to disambiguate memory references forces a loop-carried dependency bound (Section 4.1.2) of 11. While 11 is the lower bound on the number of cycles per result, there are actually so many constraints that the compiler is unable to find a schedule until ii = 12 (12 cycles/result).

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 16
;*      Loop opening brace source line   : 17
;*      Loop closing brace source line   : 17
;*      Known Minimum Trip Count         : 2
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 11
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        1
;*      .S units                     1        0
;*      .D units                     3*       2
;*      .M units                     0        0
;*      .X cross paths               0        0
;*      .T address paths             3*       2
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          1        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        1
;*      Bound(.L .S .D .LS .LSD)     2        2
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 11 Unsafe schedule for irregular loop
;*         ii = 11 Unsafe schedule for irregular loop
;*         ii = 11 Did not find schedule
;*         ii = 12 Schedule found with 2 iterations in parallel
...
;*        SINGLE SCHEDULED ITERATION
;*
;*        C25:
;*   0              LDW     .D1T1   *A4,A6           ; |17|  ^
;*   1              NOP             4
;*   5      [ B0]   LDW     .D1T1   *+A6[A3],A5      ; |17|  ^
;*     ||           LDW     .D2T2   *+B5(4),B6       ; |17|
;*   6              NOP             4
;*  10      [ B0]   STW     .D2T1   A5,*+B6[B4]      ; |17|  ^
;*  11              LDW     .D1T2   *+A4(8),B6       ; |16|
;*  12              NOP             3
;*  15              ADD     .L2     1,B4,B4          ; |16|
;*  16              ADD     .L1     1,A3,A3          ; |16| Define a twin register
;*     ||           CMPGT   .L2     B6,B4,B0         ; |16|
;*  17      [ B0]   B       .S1     C25              ; |16|
;*  18              NOP             5
;*  23              ; BRANCHCC OCCURS {C25}          ; |16|
```

As mentioned in Section 4.1.2, the (^) symbols following the load and store instructions in the single scheduled iteration are a good hint that there may be a memory disambiguation problem (that is, the load instructions might overlap with the store instruction).

Apply the advice from Section 5.1.1:

```
LoopWithStructs(myStr * restrict s)
{
   int i;

   int * restrict p  = s->data->p;
   int * restrict q  = s->data->q;
   int            sz = s->data->sz;

   #pragma MUST_ITERATE(2,,2)
   for (i=0; i<sz; i++)
       q[i] = p[i];
}
```

Recompile. The resulting optimizer comments and software-pipelined loop information are as follows. Note that the loop carried dependency has disappeared.

```
;** 15  ----------------------    C$2 = (*s).data;
;** 15  ----------------------    p = (*C$2).p;
;** 16  ----------------------    q = (*C$2).q;
;**     ----------------------    // LOOP BELOW UNROLLED BY FACTOR(2)
;**     ----------------------    U$15 = p;
;**     ----------------------    U$18 = q;
;**     ----------------------    L$1 = (*C$2).sz>>1;
;**     ----------------------    #pragma MUST_ITERATE(1, 1073741823, 1)
...
;** ---------------------------------------------------------------------------*
;**   BEGIN LOOP L1
;** ---------------------------------------------------------------------------*
;**     ----------------------g2:
;** 21  ----------------------    _memd8((void *)U$18) = _memd8((void *)U$15);
;** 20  ----------------------    U$15 += 2;
;** 20  ----------------------    U$18 += 2;
;** 20  ----------------------    if ( --L$1 ) goto g2;
;**     ----------------------    return;
...
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 20
;*      Loop opening brace source line   : 21
;*      Loop closing brace source line   : 21
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 2
```

```
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1
;*      .D units                     1        1
;*      .M units                     0        0
;*      .X cross paths               0        0
;*      .T address paths             2*       2*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1
;*      Bound(.L .S .D .LS .LSD)     1        1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 3 iterations in parallel
...
;*        SINGLE SCHEDULED ITERATION
;*
;*        C27:
;*   0              LDNDW   .D1T1   *A3++(8),A5:A4    ; |21|
;*     ||   [ B0]   BDEC    .S2     C27,B0            ; |20|
;*   1              NOP             4
;*   5              STNDW   .D1T1   A5:A4,*A6++(8)    ; |21|
;*   6              ; BRANCHCC OCCURS {C27}           ; |20|
```

The following improvements can be observed:

- The loop is unrolled 2x
- The loop body is reduced to one wide load, one wide store and one branch.
- The termination condition is now computed once outside the loop.

The tuned loop is 12x faster. After following the advice in the software pipelining information for the tuned loop regarding –mh, the code size is smaller as well. As can be seen, –mh does not help the original loop (even when compiled to allow infinite padding).[12] The reason is that the original loop, although syntactically a “for” loop, is semantically a “while” loop. This is because the compiler does not know that the termination condition is constant due to memory disambiguation issues. The tuned loop is both syntactically and semantically a “for” loop. There is more opportunity for the compiler to minimize code expansion of “for” loops than of “while” loops. The one exception is on the C64x+ when the loop buffer is exploited. Then the code growth associated with software-pipelining is negligible for both “for” loops and “while” loops. Section 5.4.2 provides another example of the interaction between –mh and “while” loops.

Source Code Version	Compiler Version	Performance Oriented Options	ii	Cycles/Result	Size (Bytes)
Original	5.1.3	–o –mv6400	12	12	104
Tuned	5.1.3	–o –mv6400	2	1	132
Original	5.1.3	–o –mv6400 –mh	12	12	104
Tuned	5.1.3	–o –mv6400 –mh16	2	1	96

[12] Compiling code with the option –mh (no parameter) is useful during the performance tuning stage for determining the maximum potential (code size or performance) benefit from using this option. However, for safety, when doing a production build, use –mh<num>, not –mh.

5.2 Optimizing “If” Statements

This section presents several techniques for optimizing “if” statements.

5.2.1 If-Conversion

The C6000 compiler will if-convert small “if” statements (“if” statements with “if” and “else” blocks that are short or empty). In other words, given the “if” statement:

```
if (p) x=5; else x=7;
```

The compiler can if-convert this statement into the following:

```
[ p] x = 5
[!p] x = 7
```

The [] notation in the pseudo code is borrowed from the C6x assembly language and indicates conditional execution of the instruction. Only when the condition is true (that is, not zero), the instruction will be executed.

After if-conversion, the branches are eliminated and the compiler can schedule these statements in any order or in parallel:

```
[ p] x = 5                [!p] x = 7                        [ p] x = 5
            or                              or
[!p] x = 7                [ p] x = 5                    || [!p] x = 7
```

If-conversion has several benefits. First, costly branches are eliminated. In general, with any compiler, the fewer branches, the better the resulting code. Second, after if-conversion, the compiler can detect that statements from the body of the “if” have no ordering constraints with respect to statements in the body of the “else”. Thus, if-conversion is a simple but powerful transformation for the C6000 architecture.

The compiler does not if-convert large “if” statements (“if” statements where the “if” or “else” block is long). The compiler does not software-pipeline loops with “if” statements that are not if-converted or eliminated in some other fashion (Sections 5.2.3 and 5.2.4). If a loop contains an “if” statement that was not converted, a message such as the following is generated:

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*      Disqualified loop: Loop contains control code
;*----------------------------------------------------------------------------*
```

The reason that the compiler does always perform conversion is that when “if” statements are large, if-conversion is not always profitable. For example, consider the following loop containing an “if” statement:

```
for (i=0; i<n; i++)
{
    if (x[i])
    {
        <large “if” statement body>
    }
}
```

If “x[i]” is usually 0, “x” is *sparse*. If “x[i]” is usually non-zero, “x” is *dense*. If “x” is sparse and the body of the “if” statement is long, if-conversion is not profitable. However, if “x” is dense, then if-conversion is profitable. Since the compiler does not know anything about “x”, it does not automatically if-convert this “if” statement. The techniques presented in Sections 5.2 through 5.4 show how to achieve good performance in both the sparse and dense cases.

5.2.2 “If” Statement Reduction When No “Else” Block Exists

Consider a loop containing a large “if” statement and no “else” statement:

Original Loop

```
for (i=0; i<n; i++)
{
    if (x[i])
    {
        <i1>
        <i2>
        ...
        <im>
        y[i] += ...
    }
}
```

Hand-Tuned Loop After “If” Statement Reduction

```
for (i=0; i<n; i++)
{
    <i1>
    <i2>
    ...
    <im>
    if (x[i])
    {
        y[i] += ...
    }
}
```

Assume that <i_1> through <i_m> are used to compute the value for y[i] with no other side effects. Then only the assignment to y needs to be guarded. The size of the “if” statement can be shrunk radically by hoisting <i_1> through <i_m> out of the loop. Now, <i_1> through <i_m> can proceed without waiting for “x[i]” to be loaded. Additionally, the loop might be a software pipelining candidate whereas previously it was not. This optimization is profitable if “x” is dense. It might not be profitable if “x” is sparse, because <i_1> through <i_m> would be executed much more often than in the original loop, potentially outweighing the benefits of software pipelining. An alternate optimization for the case where “x” is sparse presented in Section 5.4.2.

As a side note, pulling loads and stores out of “if” statements can be particularly helpful. The compiler cannot do this automatically because this might cause a memory access to be executed that otherwise would not be. This in turn could cause correctness issues (for example, memory accesses to invalid addresses) or performance problems (for example, unnecessary off-chip memory accesses).

Original “If” Statement

```
... = *x;
*x  = ...
if (p)
{
    ... = *y;
    <stuff>
    *y  = ...
}
```

“If” Statement with Loads and Stores Pulled Out

```
... = *x;
... = *y;
if (p)
{
    <stuff>
}
*x  = ...
*y  = ...
```

5.2.3 "If" Statement Elimination

An alternative to reducing "if" statements is to eliminate them altogether. Occasionally, the compiler does this automatically. If not, the transformation can be applied manually. The following example uses the same loop as in Section 5.2.2. In a few cases, this tuning variation works better. Another use for this technique is shown in Section 5.2.6.

Original Loop

```
for (i=0; i<n; i++)
{
    if (x[i])
    {
        <i1>
        <i2>
        ...
        <im>
        y[i] += ...
    }
}
```

Hand-Tuned Loop After "If" Statement Elimination

```
for (i=0; i<n; i++)
{
    <i1>
    <i2>
    ...
    <im>
    p = (x[i] != 0);
    y[i] += p * (...);
}
```

5.2.4 "If" Statement Elimination By Use of Intrinsics

Sometimes, the compiler can match a simple "if" statement and replace it with an intrinsic. An example of this is recognizing saturated math and replacing it with a call to a single intrinsic that performs the same function. This optimization has the same benefit as if-conversion, namely, that of eliminating one or more forward branches. It also reduces code size.

However, the compiler cannot easily match all of the C6000's powerful instructions automatically. For some instructions, there is no simple expression of their functionality from C/C++. Two examples are given in this section.

Consider the calculation of two 16-bit variables "a" and "b":

```
if (a>b) max = a;
else     max = b;
```

This "if" statement could prevent the pipelining of a loop (for example, when the "if" statement is contained within another "if" statement). However, on the C64x or C64x+, the entire code sequence could be implemented using a single MAX2 the instruction (which simultaneously computes two maximums, on the high and low halves of its two inputs). This instruction is accessible from C by way of an intrinsic. Using it eliminates the "if" construct and potentially leads to better loop performance.

```
max = _max2(a, b);
```

Another example is the LMBD instruction, which returns the position of the leftmost 0 or 1 in a 32-bit operand. To express the same functionality in C, likely an "if" statement and a loop would be used. The compiler can not tell by looking at this high-level construct that it would be equivalent to the LMBD instruction. This instruction, too, has a corresponding intrinsic that is accessible from C/C++.

5.2.5 "If" Statement Reduction Via Common Code Consolidation

Now consider loops that have both an "if" block and an "else" block. Frequently, the "if" block and "else" block have common code:

Original Loop

```
for (i=0; i<n; i++)
{
    if (x[i])
    {
        int t = z[i];
        t    += ...
        y[i]  = t;
        x[i]= ...

    }
    else
    {
        int t = z[i];
        y[i]  = t;
    }
}
```

Hand-Tuned Loop After Common Code Consolidation

```
for (i=0; i<n; i++)
{
    int t = z[i];
    if (x[i])
        t += ...
    y[i] = t;
    if (x[i])
        x[i] = ...
}
```

In the previous example, the reading and writing is common to both the "if" and "else" block. The "if" and "else" blocks are quite short, but in a real example either or both could be quite long. By pulling out common code, one can eliminate duplicate instructions and create one or more shorter "if" statements. This reduces code size and potentially transforms a loop into a software-pipelining candidate whereas previously it was not. If any of the "if" statements are still too long, the techniques in the Section 5.2.2 might now be applicable.

5.2.6 Eliminating Nested "If" Statements

A nested "if" statement is an "if" statement embedded inside another "if" statement. In general, nested "if" statements are not if-converted. Techniques to eliminate "if" statements (such as those in Section 5.2.3 and 5.2.4) can be effective for reducing nesting levels.

Original Loop

```
for (i=0; i<n; i++)
{
    // nested if stmt
    if (z[i])
        i1
    else
    {
         if (x[i])
             y[i] = c;
    }
}
```

Hand-Tuned Loop After Eliminating Nested "If"

```
for (i=0; i<n; i++)
{
    // nested if stmt removed
    if (z[i])
        i1
    else
    {
        p    = (x[i] != 0);
        y[i] = !p * y[i] + p * c;
    }
}
```

5.2.7 Optimizing Conditional Expressions

When the logical operator “a && b”” is evaluated, the ANSI C/C++ standard states that lazy evaluation is to be used. “b” is not evaluated unless “a” evaluates to true. The implication is that the following two code snippets are semantically identical and should result in similar assembly output when compiled:

Implicit Nested “If” Statement

```
if (<condition1> && <condition2>)
{
    ...
}
```

Equivalent Explicit Nested “If” Statement

```
if (<condition1>)
{
    if (<condition2>)
    {
        ...
    }
}
```

Thus, the code snippet on the left is implicitly a nested if statement. If the first condition is usually false and the second condition is expensive to evaluate, then it is probably best to leave the nested if statement. An example of this case would be when the second condition involves a function call. If the reverse is true and both conditions are side-effect free, it is best to reorder the conditions:

Implicit Nested “If” – Condition Order Reversed

```
if (<condition2> && <condition1>)
{
    ...
}
```

However, if the second condition is side-effect free and both are inexpensive to evaluate, it might be better to use the Boolean operator “&” instead. When the Boolean operator is used, both conditions are evaluated and then the Boolean “&” is computed:

Single-level “If” Statement

```
if ((<condition1> != 0) & (<condition2> != 0))
{
    ...
}
```

This form has the advantage over the versions with the logical operator “&&” of eliminating the implicit nested “if” statement. If either condition already evaluates to a Boolean value, the corresponding comparison against zero can be omitted. An analogous optimization can be performed when the logical operator is “||”.

The compiler sometimes performs these optimizations automatically. However, you typically have more information than the compiler (for example, regarding the likelihood that a condition evaluates to false), so it is best to write the code efficiently from the start.

5.3 Handling Function Calls

Function calls inhibit optimization. Loops that contain calls cannot be software-pipelined. Scheduling, register allocation and other optimizations are constrained across function calls as well. Thus, large or seldom invoked functions are best left alone. However, performance around small, frequently-invoked functions improve dramatically if the called function is *inlined*.

Inlining is the process of replacing a call to a function with a copy of the function body. This can be done statically with the inline keyword. Alternatively, the compiler inlined functions automatically if *all* of the following are true:

- Inlining is enabled. –oi<num> controls the inlining threshold; the larger the value of <num>, the larger the threshold. –oi0 disables inlining.
- The compiler "sees" both the *caller* (function doing the calling) and *callee* (function being called) simultaneously.
- Inlining is deemed profitable (decision based on size of callee and number of calls).

The compiler can see both the caller and callee automatically when either of the following is true:

- The caller and callee are contained in a single file and file is compiled with –o3.
- The caller and callee are in different files; both are compiled simultaneously with –o3 –pm.

For example:

```
cl6x –o3 –pm caller.c callee.c
```

See reference [1] for more information on automatic inlining and program level optimization (-pm).

The optimizer comments show when a function has been inlined:

Source File

```
callee(int *c, int d)
{
    *c = d;
}

caller (int *a, int *b)
{
    callee(a, 5);
    callee(b, 7);
}
```

Optimizer Comments After Compiling with –o2

```
_caller:
;** 8   ----------------------    callee(a, 5);
;** 9   ----------------------    callee(b, 7);
;** 9   ----------------------    return;
```

Optimizer Comments After Compiling with –o3

```
_caller:
;** 3   ----------------------    *a = 5;  // [0]
;** 3   ----------------------    *b = 7;  // [0]
;** 3   ----------------------    return;  // [0]
. . .
;; Inlined function references:
;; [  0] callee
```

Beware of disabling inlining to reduce code growth at an application level. Inlining a large function can cause code growth. However, inlining a small function can reduce code size because the body of the callee is shorter than the code to set up the call. Customization of inlined calls (exploiting knowledge of parameter values at the site of an inlined call to automatically optimize both caller and callee after inlining) can also reduce code size. By default, the compiler intelligently decides whether to inline a candidate function based on the size of the callee and the number of call sites. In contrast, –oi0 disables *all* inlining.

5.4 Improving Performance of Large Control Code Loops

Loops can be large in the sense of having a large body or in terms of executing for a large number of iterations (that is, having a large trip count). Section 5.4.1 presents a technique for optimizing loops with excessively large bodies. Section 5.4.2 presents a technique for optimizing loops with control code and large trip counts.

5.4.1 Using Scalar Expansion to Split Loops

Loops that contain too many instructions do not software pipeline well if at all. On the C6000, if a loop contains too many instructions, a message such as the following is generated:

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*      Disqualified loop: Too many instructions
;*----------------------------------------------------------------------------*
```

Loops with a large number of instructions also take a long time to compile. In many cases, better performance can be achieved by splitting excessively large loops. The overhead for the extra loop might be more than offset by the more efficient schedule that compiler will be able to generate for the two smaller loops relative to the large loop.

Consider the original loop in the example that follows. Assume that “v” is the only intermediate state of the first “if” statement that is needed for the second “if” statement. By saving the value of “v” from each iteration into a temporary array, we can decouple the computation between the two “if” statements. Then the single huge loop can be split into two more reasonably sized loops as shown in the hand-tuned loop on the right.

Original Loop

```
#MUST_ITERATE(1,20)
for (i=0; i<n; i++)
{
    int v = 0;
    if (x[i])
    {
        <largeblock1>
        v =...
    }
    if (v)
    {
        <largeblock2>
    }
}
```

Hand-Tuned Loops After Scalar Expansion/Loop Splitting

```
int tmp[20];
#MUST_ITERATE(1,20)
for (i=0; i<n; i++)
{
    int v = 0;
    if (x[i])
    {
        <largeblock1>
        v = ...
    }
    tmp[i] = v;
}

#MUST_ITERATE(1,20,)
for (i=0; i<n; i++)
{
    int v = tmp[i];
    if (v)
    {
        <largeblock2>
    }
}
```

Variable “v” has been *scalar expanded* so that there is one copy per original loop iteration. The copies are stored in intermediate array “tmp”. After this, the loop can be split into two.

In practice, it is not uncommon that intermediate states consist of only a few variables. If the number of loop iterations is not prohibitively large, these values can be scalar-expanded and the computation split into multiple loops. Caution must be taken that the temporary arrays used to store these intermediate values do not end up in external memory.

In this case, the trip count for the loop is small. If the trip count for the loop (and consequently the size of the temporary array) were too large, the loops could be *tiled*. For example, the nested loop that follows has been tiled to reduce the memory requirements for intermediate array “tmp”.

Hand-Tuned Loops After Loop Tiling

```
for (i=0; i<n; i+=TILESZ)
{
    int tmp[TILESZ];
    #pragma MUST_ITERATE(1,TILESZ)
    for (j=i; j<min(n,i+TILESZ); j++)
    {
        int v = 0;
        if (x[j])
        {
           <largeblock1>
           v= ...
        }
        tmp[j] = v;
    }

    #pragma MUST_ITERATE(1,TILESZ)
    for (j=i; j<min(n,i+TILESZ); j++)
    {
        int v = tmp[j];
        if (v)
        {
            <largeblock2>
        }
    }
}
```

5.4.2 Optimizing Sparse Loops

Consider the loop on the left that follows. It has a large body, but little of the code for iteration i (only the increment of “x”) is actually executed when “x[i]” is zero. Recall from Section 5.2.1 that the loop is not amenable to software pipelining because of the large “if” statement. While it might be feasible to apply “if” statement reduction (Section 5.2.2), this will not be profitable if “x” is sparse (many elements of array “x” are zero).

An alternative optimization technique is to split the loop into two loops as shown on the right. The first loop finds the set of non-zero iterations and computes “shortcuts” so that “empty” iterations can simply be skipped. The second loop then iterates efficiently over (only) the non-empty iterations. The first loop has the same trip count as the original loop but the body is short. The second loop has a large body (but no “if” statement). Hence, the code in the second loop can be compiled more efficiently. The trip count for the second loop is “j”, the number of non-zero elements of “x”. If “x” is sparse, “j” is much less than “n” and the two loops (which could potentially both become software-pipelining candidates) are likely to be more efficient than the original loop.

In detail, during the first loop, the set of "j" non-empty iterations are identified. The number of empty iterations that must be skipped to get to the next non-empty iteration is also computed. Let $n_0,\ldots n_i,n_{i+1},\ldots n_{j-1}$ be the set of non-empty iterations. After the end of the first loop, n_0 = skip[0]-1, $n_i = n_{i-1}$ + skip[i]. "cnt" is number of empty iterations following the last non-empty iteration.

The second loop processes the non-empty iterations. The last increment of "p" after the end of the second loop skips past the empty iterations that follow the last non-empty iteration n_{j-1}. This final increment can be omitted if "p" is not used after the end of the loop.

Original Loop

```
#MUST_ITERATE(1,200)
for (i=0; i<n; i++)
{
    if (x[i])
    {
       // large block
       <stuff for iter i>
    }
    p++;
}
```

Hand-Tuned Loops After Sparse Loop Optimization

```
int skip[200];
int j=0, k=-1, cnt=1;
#MUST_ITERATE(1,200)
for (i=0; i<n; i++)
{
    if (x[i])
    {
       skip[j++] = cnt;
       cnt       = 1;
    }
    else cnt++;
}
for (i=0; i<j; i++)
{
    k += skip[i];
    <stuff for iter k>
    p += skip[i];
}
p += cnt-1;
```

short loop body n iterations

long loop body usually << n iterations

Reserve this technique for cases where "x" is sparse. When "x" is dense, it is more efficient to apply the techniques from Section 5.2.2, which have the advantage of preserving the linearity in the increments of "x" and "p". The compiler can exploit this linearity to apply wider loads and stores and perform other optimizations.

Initially, one might expect this to be an uncommon construct in C/C++ programs. In practice, once this idea is generalized a little bit, it turns out to be a rather commonly found pattern. In particular, the previous example is simply one instance of an algorithm that selectively performs an operation based on the value of one or more elements in a set.

In the following example, the original loop is a more generic version of this algorithm where the set is generalized to be an arbitrary data structure such as a linked list and the address of the next item in the set is a function of the previous item. In the hand-tuned version on the right, the pointer to the interesting data is stored in a temporary array to avoid additional calls to next_item_in_set().

Original Loop

```
#MUST_ITERATE(1,200)
for (i=0; i<n; i++)
{
    if (p->value == value)
    {
       // large block
       <stuff for iter i>
    }
    p = next_item_in_set(p);
}
```

Hand-Tuned Loops After Sparse Loop Optimization

```
int skip[200];
type_of_p plist[200], *plast;
int j=0, k=-1, cnt=1;
#MUST_ITERATE(1,200)
for (i=0; i<n; i++)
{
    if (p->value == value)
    {
       plist[j]  = p;
       skip[j++] = cnt;
       cnt       = 1;
    }
    else cnt++;
    p = next_item_in_set(p);
}
plast = p;
for (i=0; i<j; i++)
{
    k += skip[i];
    p = plist[j];
    <stuff for iter k>
}
p = plast;
```

As with the optimization in the previous section, be sure the transformation will not result in additional external memory accesses that might offset its benefit. This is another example where loop tiling (Section 5.4.1) could be used to reduce the size of not only the intermediate arrays but potentially the overall working set as well.

Another common variation on this theme is one in which a part of the body of the “if” statement is executed only once. The following example shows two similar versions of a loop. The difference is that the call, which is only executed once, is pulled out of the loop on the right.

Original Loop

```
int done = 0;
int i    = 0;
while (!done) {
    if (x[i])
    {
        done = 1;
        callee(&x[i]);
    }
    i++;
}
```

Hand-Tuned Loop After Pulling Out Function Call

```
int done = 0;
int i    = 0;
while (!done) {
    if (x[i])
    {
        done = 1;
    }
    i++;
}
callee(&x[i-1]);
```

Performance and code size are summarized in the following table:

Source Code Version	Compiler Version	Performance-Oriented Options[13]	Cycles/Result	Size (Bytes)
Original	5.1.3	–o –mv6400	13	84
Hand-tuned	5.1.3	–o –mv6400	6	60
Hand-tuned	5.1.3	–o –mv6400 –mh56	1	184
Hand-tuned	6.0.3	–o –mv64+ –mh56	1	56

The original loop fails to software pipeline because it contains a function call. It takes 13 cycles per iteration and its size is 84 bytes.

The hand-tuned loop, when compiled with the same options, exhibits a 2x speedup. It is 29% smaller as well. When option –mh56 is added, the speedup increases to 12x, but the code size grows to 184 bytes. As mentioned earlier, -mh can improve performance, code size or both. Whereas –mh improves the *code size* of the "for" loop from Section 4.4.2.4, it improves the *performance* of this hand-tuned "while" loop.[14]

The situation is quite different when compiling for C64x+. Because the loop buffer is exploited, the hand-tuned loop is not only faster than the original version, but smaller as well.

6 Summary

This application note is intended both for developers who are writing new code and for those who are tuning existing applications. This report shows how to do the following:

- Select performance-oriented options.
- More easily understand compiler-generated assembly.
- Identify and fix performance bottlenecks.
- Write more efficient code from the start.

The techniques presented in this application report have all been applied to real users' code. The examples have been intentionally simplified so that the methods are easy to understand.

7 References

1. *TMS320C6000 Optimizing Compiler User's Guide* (SPRU187L or SPRU187N).
2. *TMS320C6000 CPU and Instruction Set Guide* (SPRU189G).
3. *Introduction to Compiler Consultant* (SPRAA14).
4. *TMS320C6000 Programmer's Guide* (SPRU198I).

[13] The option -mh56 was selected based on the advice in the software-pipelining information when compiling the hand-tuned loop without –mh. Adding –mh does not impact the original loop because it does not software pipeline.

[14] When the loop buffer is not used, better schedules (equivalently, smaller values of ii) can sometimes lead to larger code size.

8 Acknowledgements

I would like to thank Serene Banerjee, David Bartley, Mark Blake, Jason Brand, Jackie Brenner, Alan Campbell, Brett Huber, George Mock, Andre Schnarrenberger, Scott Specker, Raja Venkateswaran, and Vincent Wan, all of whom provided feedback on this report on short notice. In some cases, this feedback was extensive, including entire paragraphs, complete figures, and/or additional tips and tricks from their own experience. I would also like to thank Yvonne Degraw who undertook the painful task of formatting and proofreading this report.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265