**Table 3-4:** C/C++ Compiler Optimization Switches

| *Switch Name* | *Description* |
|---|---|
| `-const-read-write` | Specifies that data accessed via a pointer to `const` data may be modified elsewhere |
| `-flags-link -e` (see `-flags-{asm\|compiler\|ipa\|lib\|link\|mem\|prelink}` *`switch[, switch2[, ...]]`*) | Specifies linker section elimination |
| `-force-circbuf` | Treats array references of the form `array[i%n]` as circular buffer operations |
| `-ipa` | Turns on inter-procedural optimization. Implies use of `-O`. May be used in conjunction with `-Os` or `-Ov`. |
| `-no-fp-associative` | Does not treat floating-point multiply and addition as an associative |
| `-no-saturation` | Does not turn non-saturating operations into saturating ones |
| `-O[0\|1]` | Enables code optimizations and optimizes the file for speed |
| `-Os` | Optimizes the file for size |
| `-Ov` *`num`* | Controls speed vs. size optimizations (sliding scale) |
| `-pguide` | Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization |
| `-save-temps` | Saves intermediate files (for example, `.s`) |

## How Loop Optimization Works

Loop optimization is important to overall application performance, because any performance gain achieved within the body of a loop reaps a benefit for every iteration of that loop. This section provides an introduction to some of the concepts used in loop optimization, helping you to use the compiler features in this chapter.

This section contains:

- Terminology
- Loop Optimization Concepts
- A Worked Example

### Terminology

This section describes terms that have particular meanings for compiler behavior.

#### *Clobbered Register*

A register is *clobbered* if its value is changed so that the compiler cannot usefully make assumptions about its new contents.

For example, when the compiler generates a call to an external function, the compiler considers all caller-preserved registers to be clobbered by the called function. Once the called function returns, the compiler cannot make any assumptions about the values of those registers. This is why they are called "caller-preserved". If the caller needs the values in those registers, the caller must preserve them itself.

The set of registers clobbered by a function can be changed using `#pragma regs_clobbered`, and the set of registers changed by a gnu `asm` statement is determined by the clobber part of the `asm` statement.

### *Live Register*

A register is *live* if it contains a value needed by the compiler, and thus cannot be overwritten by a new assignment to that register. For example, to do "A = B + C", the compiler might produce:

```
reg1 = load B        // reg1 becomes live
reg2 = load C        // reg2 becomes live
reg1 = reg1 + reg2   // reg2 ceases to be live;
                     // reg1 still live, but with a different
                     // value
store reg1 to A      // reg1 ceases to be live
```

Liveness determines which registers the compiler may use. In this example, since `reg1` is used to load `B`, and that register must maintain its value until the addition, `reg1` cannot also be used to load the value of `C`, unless the value in `reg1` is first stored elsewhere.

### *Spill*

When a compiler needs to store a value in a register, and all usable registers are already live, the compiler must store the value of one of the registers to temporary storage (the stack). This *spilling* process prevents the loss of a necessary value.

### *Scheduling*

*Scheduling* is the process of re-ordering the program instructions to increase the efficiency of the generated code but without changing the program's behavior. The compiler attempts to produce the most efficient schedule.

### *Loop Kernel*

The *loop kernel* is the body of code that is executed once per iteration of the loop. It excludes any code required to set up the loop or to finalize it after completion.

### *Loop Prolog*

A *loop prolog* is a sequence of code required to set the machine into a state whereby the loop kernel can execute. For example, the prolog may pre-load some values into registers ready for use in the loop kernel. Not all loops need a prolog.

### *Loop Epilog*

A *loop epilog* is a sequence of code responsible for finalizing the execution of a loop. After each iteration of the loop kernel, the machine will be in a state where the next iteration can begin efficiently. The epilog moves values from the final iteration to where they need to be for the rest of the function to execute. For example, the epilog might save values to memory. Not all loops need an epilog.

### *Loop Invariant*

A *loop invariant* is an expression that has the same value for all iterations of a loop. For example:

```
int i, n = 10;
for (i = 0; i < n; i++) {
    val += i;
}
```

The variable n is a loop invariant. Its value is not changed during the body of the loop, so n will have the value 10 for every iteration of the loop.

### *Hoisting*

When the optimizer determines that some part of a loop is computing a value that is actually a loop invariant, it may move that computation to before the loop. This prevents the same value from being re-computed for every iteration. This is called *hoisting*.

### *Sinking*

When the optimizer determines that some part of a loop is computing a value that is not used until the loop terminates, the compiler may move that computation to after the loop. This *sinking* process ensures the value is only computed using the values from the final iteration.

## Loop Optimization Concepts

The compiler optimizer focuses considerable attention on program loops, as any gain in the loop's performance reaps the benefits on every iteration of the loop. The applied transformations can produce code that appears to be substantially different from the structure of the original source code. This section provides an introduction to the compiler's loop optimization, to help you understand why the code might be different.

This section describes:

- Software Pipelining
- Loop Rotation
- Loop Vectorization
- Modulo Scheduling

The following examples are presented in terms of a hypothetical machine. This machine is capable of issuing up to two instructions in parallel, provided one instruction is an arithmetic instruction, and the other is a load or a store. Two arithmetic instructions may not be issued at once, nor may two memory accesses:

```
t0 = t0 + t1;              // valid: single arithmetic
t2 = [p0];                 // valid: single memory access
[p1] = t2;                 // valid: single memory access
t2 = t1 + 4, t1 = [p0];    // valid: arithmetic and memory
t5 += 1, t6 -= 1;          // invalid: two arithmetic
[p3] = t2, t4 = [p5];      // invalid: two memory
```

The machine can use the old value of a register and assign a new value to it in the same cycle, for example:

```
t2 = t1 + 4, t1 = [p0];   // valid: arithmetic and memory
```

The value of `t1` on entry to the instruction is the value used in the addition. On completion of the instruction, `t1` contains the value loaded via the `p0` register.

The examples will show `START LOOP N` and `END LOOP`, to indicate the boundaries of a loop that iterates `N` times. (The mechanisms of the loop entry and exit are not relevant).

## Software Pipelining

*Software pipelining* is analogous to hardware pipelining used in some processors. Whereas hardware pipelining allows a processor to start processing one instruction before the preceding instruction has completed, software pipelining allows the generated code to begin processing the next iteration of the original source-code loop before the preceding iteration is complete.

Software pipelining makes use of a processor's ability to multi-issue instructions. Regarding known delays between instructions, it schedules instructions from later iterations where there is spare capacity.

## Loop Rotation

*Loop rotation* is a common technique of achieving software pipelining. It changes the logical start and end positions of the loop within the overall instruction sequence, to allow a better schedule within the loop itself. For example, this loop:

```
START LOOP N
   A
   B
   C
   D
   E
END LOOP
```

could be rotated to produce the following loop:

```
A
B
C
START LOOP N-1
   D
   E
   A
   B
   C
END LOOP
D
E
```

The order of instructions in the loop kernel is now different. It still circles from instruction `E` back to instruction `A`, but now it starts at `D`, rather than `A`. The loop also has a prolog and epilog added, to preserve the intended order of instructions. Since the combined prolog and epilog make up a complete iteration of the loop, the kernel is now executing `N-1` iterations, instead of `N`.

In this example, consider the following loop:

```
START LOOP N
   t0 += 1
   [p0++] = t0
END LOOP
```

This loop has a two-cycle kernel. While the machine could execute the two instructions in a single cycle-an arithmetic instruction and a memory access instruction-to do so would be invalid, because the second instruction depends upon the value computed in the first instruction.

However, if the loop is rotated, we get:

```
t0 += 1
START LOOP N-1
   [p0++] = t0
   t0 += 1
END LOOP
[p0++] = t0
```

The value being stored is computed in the previous iteration (or before the loop starts, in the prolog). This allows the two instructions to be executed in a single cycle:

```
t0 += 1
START LOOP N-1
   [p0++] = t0, t0 += 1
END LOOP
[p0++] = t0
```

Rotating the loop has presented an opportunity by which the `k`th iteration of the original loop is starting (`t0 += 1`) while the `(k-1)`th iteration is completing (`[p0++] = t0`), so rotation has achieved software pipelining, and the performance of the loop is doubled.

Notice that this process has changed the structure of the program slightly: suppose that the loop construct always executes the loop at least once; that is, it is a `1..N` count. Then if `N==1`, changing the loop to be `N-1` would be problematic. In this example, the compiler inserts a guard: a conditional jump around the loop construct for the circumstances where the compiler cannot guarantee that `N > 1`:

```
t0 += 1
IF N == 1 JUMP L1;
   START LOOP N-1
   [p0++] = t0, t0 += 1
END LOOP
L1:
   [p0++] = t0
```

## Loop Vectorization

*Loop vectorization* is another transformation that allows the generated code to execute more than one iteration in parallel. However, vectorization is different from software pipelining. Where software pipelining uses a different ordering of instructions to get better performance, vectorization uses a different set of instructions. These vector instructions act on multiple data elements concurrently to replace multiple executions of each original instruction.

For example, consider this dot-product loop:

```
int i, sum = 0;
for (i = 0; i < n; i++) {
    sum += x[i] * y[i];
}
```

This loop walks two arrays, reading consecutive values from each, multiplying them and adding the result to the ongoing sum. This loop has these important characteristics:

- Successive iterations of the loop read from adjacent locations in the arrays.
- The dependency between successive iterations is the summation, a commutative and associative operation.
- Operations such as load, multiply and add are often available in parallel versions on embedded processors.

These characteristics allow the optimizer to vectorize the loop so that two elements are read from each array per load, two multiplies are done, and two totals maintained.

The vectorized loop would be:

```
t0 = t1 = 0
START LOOP N/2
t2 = [p0++] (Wide)     // load x[i] and x[i+1]
t3 = [p1++] (Wide)     // load y[i] and y[i+1]
t0 += t2 * t3 (Low), t1 += t2 * t3 (High)    // vector mulacc
END LOOP
t0 = t0 + t1           // combine totals for low and high
```

Vectorization is most efficient when all the operations in the loop can be expressed in terms of parallel operations. Loops with conditional control flow in them are rarely vectorizable, because the compiler cannot guarantee that the condition will evaluate in the same way for all the iterations being executed in parallel. However computations that use the `?:` operator may be vectorized in many circumstances.

Vectorization is also affected by data alignment constraints and data access patterns. Data alignment affects vectorization because processors often constrain loads and stores to be aligned on certain boundaries. While the unvectorized version will guarantee this, the vectorized version imposes a greater constraint that may not be guaranteed. Data access patterns affect vectorization because memory accesses must be contiguous. If a loop accessed every tenth element, for example, then the compiler would not be able to combine the two loads for successive iterations into a single access.

Vectorization divides the generated iteration count by the number of iterations being processed in parallel. If the trip count of the original loop is unknown, the compiler will have to conditionally execute some iterations of the loop.

**NOTE:** Vectorization and software pipelining are not mutually exclusive: the compiler may vectorize a loop and then use software pipelining to obtain better performance.

## Modulo Scheduling

Loop rotation, as described earlier, is a simple software-pipelining method that can often improve loop performance, but more complex examples require a more advanced approach. The compiler uses a popular technique known as *modulo scheduling*, which can produce more efficient schedules for loops than simple loop rotation.

Modulo scheduling is used to schedule innermost loops without control flow. A modulo-scheduled loop is described using the following parameters:

- Initiation interval (II): the number of cycles between initiating two successive iterations of the original loop.
- Minimum initiation interval due to resources (res MII): a lower limit for the initiation interval (II); an II lower than this would mean at least one of the resources being used at greater capacity than the machine allows.
- Minimum initiation interval due to recurrences (rec MII): an instruction cannot be executed until earlier instructions on which it depends have also been executed. These earlier instructions may belong to a previous loop iteration. A cycle of such dependencies (a recurrence) imposes a minimum number of cycles for the loop.
- Stage count (SC): the number of initiation intervals until the first iteration of the loop has completed. This is also the number of iterations in progress at any time within the kernel.
- Modulo variable expansion unroll factor (MVE unroll): the number of times the loop has to be unrolled to generate the schedule without overlapping register lifetimes.
- Trip count: the number of times the loop kernel iterates.
- Trip modulo: a number that is known to divide the trip count.
- Trip maximum: an upper limit for the trip count.
- Trip minimum: a lower limit for the trip count.

Understanding these parameters will allow you to interpret the generated code more easily. The compiler's assembly annotations use these terms, so you can examine the source code and the generated instructions, to see how the scheduling relates to the original source. See Assembly Optimizer Annotations for more information.

Modulo scheduling performs software pipelining by:

- Ordering the original instructions in a sequence (for simplicity referred to as the *base schedule*) that can be repeated after an interval known as the *initiation interval* ("II");
- Issuing parts of the base schedule belonging to successive iterations of the original loop, in parallel.

For the purposes of this discussion, all instructions will be assumed to require only a single cycle to execute; on a real processor, stalls affect the initiation interval, so a loop that executes in II cycles may have fewer than II instructions.

### Initiation Interval (II) and the Kernel

Consider the loop

```
START LOOP N
    A
    B
    C
    D
    E
    F
    G
    H
END LOOP
```

Now consider that the compiler finds a new order for `A,B,C,D,E,F,G,H` grouping; some of them on the same cycle so that a new instance of the sequence can be started every two cycles. Say this *base schedule* is given in the ***Base Schedule*** table, where `I1,I2,...,I8` are `A,B,...,H` reordered. Albeit a valid schedule for the original loop, the base schedule is not the final modulo schedule; it may not even be the shortest schedule of the original loop. However, the base schedule is used to obtain the modulo schedule, by being able to initiate it every II=2 cycles, as seen in the ***Obtaining the Modulo Schedule by Repeating the Base Schedule every II=2 Cycles*** table.

**Table 3-5:** Base Schedule

| *Cycle* | *Instructions* |
|---|---|
| 1 | I1 |
| 2 | I2, I3 |
| 3 | I4, I5 |
| 4 | I6 |
| 5 | I7 |
| 6 | I8 |

**Table 3-6:** Obtaining the Modulo Schedule by Repeating the Base Schedule Every II=2 Cycles

| *Cycle* | *Iteration 1* | *Iteration 2* | *Iteration 3* | *Iteration 4* |
|---|---|---|---|---|
| 1 | I1 | | | |
| 2 | I2, I3 | | | |
| 3 | I4, I5 | I1 | | |
| 4 | I6 | I2, I3 | | |
| ***5*** | ***I7*** | ***I4, I5*** | ***I1*** | |
| ***6*** | ***I8*** | ***I6*** | ***I2, I3*** | |
| 7 | | I7 | I4, I5 | I1 |
| 8 | | I8 | I6 | I2, I3 |
| 9 | | | I7 | I4, I5 |

**Table 3-6:** Obtaining the Modulo Schedule by Repeating the Base Schedule Every II=2 Cycles (Continued)

| *Cycle* | *Iteration 1* | *Iteration 2* | *Iteration 3* | *Iteration 4* |
|---|---|---|---|---|
| 10 | | | I8 | I6 |

Starting at cycle 5, the pattern in the ***Loop Kernel (N>=3)*** table keeps repeating every 2 cycles. This repeating pattern is the *kernel*, and it represents the modulo scheduled loop.

**Table 3-7:** Loop Kernel (N>=3)

| *Cycle* | *Iteration N-2 (last stage)* | *Iteration N-1 (2nd stage)* | *Iteration N (1st stage)* |
|---|---|---|---|
| II*N-1 | I7 | I4, I5 | I1 |
| II*N | I8 | I6 | I2, I3 |

The initiation interval has the value II=2, because iteration `i+1` can start two cycles after the cycle on which iteration `i` starts. This way, one iteration of the original loop is initiated every II cycles, running in parallel with previous, unfinished iterations.

The initiation interval of the loop indicates several important characteristics of the schedule for the loop:

- The loop kernel will be II cycles in length.
- A new iteration of the original loop will start every II cycles. An iteration of the original loop will end every II cycles.
- The same instruction will execute on cycle `c` and on cycle `c+II` (hence the name modulo schedule).

Finding a modulo schedule implies finding a base schedule and an II such that the base schedule can be initiated every II cycles.

If the compiler can reduce the value for II, it can start the next iteration sooner, and thus increase the performance of the loop: The lower the II, the more efficient the schedule. However the II is limited by a number of factors, including:

- The machine resources required by the instructions in the loop.
- The data dependencies and stalls between instructions.

We'll examine each of these limiting factors.

### Minimum Initiation Interval Due to Resources (Res MII)

The first factor that limits II is machine resource usage. Let's start with the simple observation that the kernel of a modulo scheduled loop contains the same set of instructions as the original loop.

Assume a machine that can execute up to four instructions in parallel. If the loop has 8 instructions, then it requires a minimum of 2 lines in the kernel, since there can be at most 4 instructions on a line. This implies II has to be at least 2, and we can tell this without having found a base schedule for the loop, or even knowing what the specific instructions are.

Consider another example where the original loop contains 3 memory accesses to be scheduled on a machine that supports at most 2 memory accesses per cycle. This implies at least 2 cycles in the kernel, regardless of the rest of the instructions.

Given a set of instructions in a loop, we can determine a lower bound for the II of any modulo schedule for that loop based on resources required. This lower bound is called the *Resource based Minimum Initiation Interval (Res MII)*.

### Minimum Initiation Interval Due to Recurrences (Rec MII)

A less obvious limitation for finding a low II are cycles in the data dependencies between instructions.

Assume that the loop to be scheduled contains (among others) the instructions:

```
i3: t3=t1+t5;   // t5 carried from the previous iteration
i5: t5=t1+t3;
```

Assume each line of instructions takes 1 cycle. If `i3` is executed at cycle `c` then `t3` is available at cycle `c+1` and `t5` cannot be computed earlier than `c+1` (because it depends on `t3`), and similarly the next time we compute `t3` cannot be earlier than `c+2`. Thus if we execute `i3` at cycle `c`, the next time we can execute `i3` again cannot be earlier than `c+2`. But for any modulo schedule, if an instruction is executed at cycle `c`, the next iteration will execute the same instruction at cycle `c+II`. Therefore, II has to be at least 2 due to the circular data dependency path `t3->t5->t3`.

This lower bound for II, given by circular data dependencies (recurrences) is called the *Minimum Initiation Interval Due to Recurrences (Rec MII)*, and the data dependency path is called *loop carry path*. There can be any number of loop carry paths in a loop, including none, and they are not necessarily disjoint.

### Stage Count (SC)

The kernel in the ***Loop Kernel (N>=3)*** table (see Initiation Interval (II) and the Kernel) is formed of instructions which belong to 3 distinct iterations of the original loop: {`I7,I8`} end the "oldest" iteration - in other words they belong to the iteration started the longest time before the current cycle; {`I4,I5,I6`} belong to the next oldest initiated iteration, and so on. {`I1,I2,I3`} are the beginning of the youngest iteration.

The number of iterations of the original loop in progress at any time within the kernel is called the *Stage Count* (SC). This is also the number of initiation intervals until the first iteration of the loop completes. In our example `SC=3`.

The final schedule requires peeling a few instructions (the prolog) from the beginning of the first iteration and a few instructions (the epilog) from the end of the last iteration in order to preserve the structure of the kernel. This reduces the trip count from `N` to `N-(SC-1)`:

```
I1;                            // prolog
I2,I3;                         // prolog
I4,I5,      I1;                // prolog
I6,         I2,I3;             // prolog
LOOP N-2                       // i.e. N-(SC-1), where SC=3
I7,         I4,I5,      I1;    // kernel
I8,         I6,         I2,I3; // kernel
END LOOP
```

```
              I7,          I4, I5;  // epilog
              I8,          I6;      // epilog
                           I7;      // epilog
                           I8;      // epilog
```

Another way of viewing the modulo schedule is to group instructions into stages as in the ***Instructions Grouped into Stages*** table, where each stage is viewed as a vector of height `II=2` of instruction lists (that represent parts of instruction lines).

**Table 3-8:** Instructions Grouped into Stages

| *StageCount* | *Instructions* |
|---|---|
| `SC0` | I1,<br>I2, I3 |
| `SC1` | I4, I5,<br>I6 |
| `SC2` | I7,<br>I8 |

Now the schedule can be viewed as:

```
SC0                             // prolog
SC1    SC0                      // prolog
LOOP (N-2)                      // That is N-(SC-1), where SC=3
SC2    SC1    SC0               // kernel
END LOOP
       SC2    SC1               // epilog
              SC2               // epilog
```

where, for example, SC2 SC1 is the 2 line vector obtained from concatenating the lists in `SC2` and `SC1`.

### Variable Expansion and MVE Unroll

There is one more issue to address for modulo schedule correctness.

Consider the sequence of instructions in the ***Problematic Instance*** table.

**Table 3-9:** Problematic Instance

| *Generic Instruction* | *Specific Instance* |
|---|---|
| `I1` | `t1=[p1++]` |
| `I2` | `t2=[p2++]` |
| `I3` | `t3=t1+t5` |
| `I4` | `t4=t2+1` |
| `I5` | `t5=t1+t3` |

**Table 3-9:** Problematic Instance (Continued)

| *Generic Instruction* | *Specific Instance* |
|---|---|
| `I6` | `t6=t4*t5` |
| `I7` | `t7=t6*t3` |
| `I8` | `[p8++]=t7` |

The following table shows the base schedule that is an instance of the one in the ***Base Schedule*** table (see Initiation Interval (II) and the Kernel).

**Table 3-10:** Base Schedule (From the Base Schedule Table) Applied to the Instances in the Problematic Instance Table

| | |
|---|---|
| 1 | t1=[p1++] |
| 2 | `t2=[p2++],t3=t1+t5` |
| 3 | `t4=t2+1,t5=t1+t3` |
| 4 | `t6=t4*t5` |
| 5 | `t7=t6*t3` |
| 6 | `[p8++]=t7` |

The ***Modulo Schedule Broken by Overlapping Lifetimes of t3*** table shows the corresponding modulo schedule with `II=2`.

**Table 3-11:** Modulo Schedule Broken by Overlapping Lifetimes of t3

| | *Iteration 1* | *Iteration 2* | *Iteration 3 ...* |
|---|---|---|---|
| 1 | `t1=[p1++]` | | |
| 2 | `t2=[p2++],t3=t1+t5` | | |
| 3 | `t4=t2+1,t5=t1+t3` | `t1=[p1++]` | |
| 4 | `t6=t4*t5` | `t2=[p2++],t3=t1+t5` | |
| *5* | ***`t7=t6*t3`*** | ***`t4=t2+1,t5=t1+t3`*** | ***`t1=[p1++]`*** |
| *6* | ***`[p8++]=t7`*** | ***`t6=t4*t5`*** | ***`t2=[p2++],t3=t1+t5`*** |
| 7 | | `t7=t6*t3` | `t4=t2+1,t5=t1+t3` |
| 8 | | `[p8++]=t7` | `t6=t4*t5` |
| 9 | | | `t7=t6*t3` |
| 10 | | | `[p8++]=t7` |

However, there is a problem with the schedule in the ***Modulo Schedule Broken by Overlapping Lifetimes of t3*** table: `t3` defined in the fourth cycle (second column in the table) is used on the fifth cycle (first column); however, the intended use was of the value defined on the second cycle (first column). In general, the value of `t3` used by `t7=t6*t3` in the kernel will be the one defined in the previous cycle, instead of the one defined 3 cycles earlier, as intended. Thus, if the compiler were to use this schedule as-is, it would be clobbering the live value in `t3`. The lifetime of each value loaded into `t3` is 3 cycles, but the loop's initiation interval is only 2, so the lifetimes of `t3` from different iterations overlap.

The compiler fixes this by duplicating the kernel as many times as needed to exceed the longest lifetime in the base schedule, then renaming the variables that clash-in this case, just `t3`. In the ***Modulo Schedule Corrected by Variable Expansion (t3 and t3_2)*** table, we see that the length of the new loop body is 4, greater than the lifetimes of the values in the loop.

**Table 3-12:** Modulo Schedule Corrected by Variable Expansion: t3 and t3_2

| | *Iteration 1* | *Iteration 2* | *Iteration 3* | *Iteration 4 ...* |
|---|---|---|---|---|
| 1 | t1=[p1++] | | | |
| 2 | t2=[p2++],t3=t1+t5 | | | |
| 3 | t4=t2+1,t5=t1+t3 | t1=[p1++] | | |
| 4 | t6=t4*t5 | t2=[p2++],t3_2=t1+t5 | | |
| 5 | t7=t6*t3 | t4=t2+1,t5=t1+t3_2 | t1=[p1++] | |
| 6 | [p8++]=t7 | t6=t4*t5 | t2=[p2++],t3=t1+t5 | |
| 7 | | t7=t6*t3_2 | t4=t2+1,t5=t1+t3 | t1=[p1++] |
| 8 | | [p8++]=t7 | t6=t4*t5 | t2=[p2++],t3_2=t1+t5 |
| 9 | | | t7=t6*t3 | t4=t2+1,t5=t1+t3_2 |
| 10 | | | [p8++]=t7 | t6=t4*t5 |
| 11 | | | | t7=t6*t3_2 |
| 12 | | | | [p8++]=t7 |

So the loop becomes:

```
t1=[p1++];

t2=[p2++],t3=t1+t5;
t4=t2+1, t5=t1+t3, t1=[p1++];
t6=t4*t5,          t2=[p2++],t3_2=t1+t5;
LOOP (N-2)/2
t7=t6*t3,          t4=t2+1,t5=t1+t3_2,  t1=[p1++];
[p8++]=t7,         t6=t4*t5,            t2=[p2++],t3=t1+t5;
                   t7=t6*t3_2,          t4=t2+1,t5=t1+t3,   t1=[p1++];
                   [p8++]=t7,           t6=t4*t5, t2=[p2++],t3_2=t1+t5;
END LOOP
                   t7=t6*t3,            t4=t2+1,t5=t1+t3_2;
```

```
                                              [p8++]=t7,          t6=t4*t5;
                                                                  t7=t6*t3_2;
                                                                  [p8++]=t7;
```

This process of duplicating the kernel and renaming colliding variables is called *variable expansion*, and the number of times the compiler duplicates the kernel is referred to as the *modulo variable expansion factor (MVE)*.

Conceptually we use different set of names, "register sets", for successive iterations of the original loop in progress in the unrolled kernel (in practice we rename just the conflicting variables, see the ***Instructions after Modulo Variable Expansion*** table). In terms of reading the code, this means that a single iteration of the loop generated by the compiler will be processing more than one iteration of the original loop. Also, the compiler will be using more registers to allow the iterations of the original loop to overlap without clobbering the live values.

In terms of stages:

```
SC0                                  // prolog
SC1    SC0_2                         // prolog
LOOP (N-2)/2              // That is N-(SC-1)/MVE, where SC=3, MVE=2
SC2    SC1_2    SC0                  // kernel
       SC2_2    SC1     SC0_2        // kernel
END LOOP
                SC2     SC1_2        // epilog
                        SC2_2        // epilog
```

where SCN_2 is SCN subject to renaming; in our case only occurrences of `t3` are renamed as `t3_2` in SCN_2.

In terms of instructions:

```
I1;                                                 // prolog
I2,I3;                                              // prolog
I4,I5,      I1_2;                                   // prolog
I6,         I2_2,I3_2;                              // prolog
LOOP(N-2)/2      // That is N-(SC-1)/MVE, where SC=3, MVE=2
I7,         I4_2,I5_2,  I1;                         // kernel
I8,         I6_2,       I2,I3;                      // kernel
            I7_2,       I4,I5,      I1_2;           // kernel
            I8_2,       I6,         I2_2,I3_2;      // kernel
END LOOP
                        I7,         I4_2,I5_2;      // epilog
                        I8,         I6_2;           // epilog
                                    I7_2;           // epilog
                                    I8_2;           // epilog
```

where `IN_2` is `IN` subject to renaming, in our case only occurrences of `t3` are renamed as `t3_2` in all `IN_2`, as seen in the ***Instructions after Modulo Variable Expansion*** table.

**Table 3-13:** Instructions after Modulo Variable Expansion

| *Generic Instruction* | *Specific Instance* |
|---|---|
| | `t1=[p1++]` |

**Table 3-13:** Instructions after Modulo Variable Expansion (Continued)

| *Generic Instruction* | *Specific Instance* |
|---|---|
| `I1` and `I1_2` | |
| `I2` and `I2_2` | `t2=[p2++]` |
| `I3` | `t3=t1+t5` |
| `I3_2` | `t3_2=t1+t5` |
| `I4` and `I4_2` | `t4=t2+1` |
| `I5` | `t5=t1+t3` |
| `I5_2` | `t5=t1+t3_2` |
| `I6` and `I6_2` | `t6=t4*t5` |
| `I7` | `t7=t6*t3` |
| `I7_2` | `t7=t6*t3_2` |
| `I8` and `I8_2` | `[p8++]=t7` |

### Trip Count

Notice that as the modulo scheduler expands the loop kernel to add in the extra variable sets, the iteration count of the generated loop changes from `(N-SC)` to `(N-SC)/MVE`. This is because each iteration of the generated loop is now doing more than one iteration of the original loop, so fewer generated iterations are required.

However, this also relies on the compiler knowing that it can divide the loop count in this manner. For example, if the compiler produces a loop with `MVE=2` so that the count should be `(N-SC)/2`, an odd value of `(N-SC)` causes problems. In these cases, the compiler generates additional "peeled" iterations of the original loop to handle the remaining iteration. As with rotation, if the compiler cannot determine the value of `N`, it will make parts of the loop-the kernel or peeled iterations conditional so that they are executed only for the appropriate values of `N`.

The number of times the generated loop iterates is called the "trip count". As explained above, sometimes knowing the trip count is important for efficient scheduling. However, the trip count is not always available. Lacking it, additional information may be inferred, or passed to the compiler through the `loop_count` pragma, specifying:

- *Trip minimum*: a lower bound for the trip count
- *Trip maximum*: an upper bound for the trip count
- *Trip modulo*: a number known to divide the trip count

## A Worked Example

The following floating-point scalar product loop are used to show how the compiler optimizer works.

***Example.C source code for floating-point scalar product***

```
#include <builtins.h>
float sp(float *a, float *b, int n) {
   int i;
   float sum=0;
   aligned(a, 2 * sizeof(float));
   aligned(b, 2 * sizeof(float));
   for (i=0; i<n; i++) {
      sum+=a[i]*b[i];
   }
   return sum;
}
```

After code generation and conventional scalar optimizations, the compiler generates a loop that resembles the following example.

***Example.Initial code generated for floating-point scalar product***

```
lcntr = r3, do(pc, .P1L10-1) until lce;
.P1L9:
   r4 = dm(i1, m6);
   r2 = dm(i0, m6);
   f12 = f2 * f4;
   f10 = f10 + f12;
   // end_loop .P1L9;
.P1L10:
```

The loop exit test has been moved to the bottom and the loop counter rewritten to count down to zero. This enables a zero-overhead hardware loop to be created. (`r3` is initialized with the loop count.) `sum` is being accumulated in `r10`. `i0` and `i1` hold pointers that are initialized with the parameters `a` and `b` and incremented on each iteration.

The SHARC processors supported by the compiler have two compute units that may perform computations simultaneously. To use both these compute blocks, the optimizer unrolls the loop to run two iterations in parallel. `sum` is now being accumulated in `r10` and `s10`, which must be added together after the loop to produce the final result. On some SHARC processors, to use the dual-word loads needed for the loop to be as efficient as this, the compiler has to know that `i0` and `i1` have initial values that are even-word aligned. This is done in the above example by use of `aligned()`, although it could also be propagated with IPA.

Note also that unless the compiler knows that original loop was executed an even number of times, a conditionally-executed odd iteration must be inserted outside the loop. `r3` is now initialized with half the value of the original loop.

***Example.Code generated for floating-point scalar product after vectorization transformation***

```
bit set mode1 0x200000; nop;      // enter SIMD mode
m4 = 2;
lcntr = r3, do(pc, .P1L10-1) until lce;
.P1L9:
   r4 = dm(i1, m4);
   r2 = dm(i0, m4);
   f12 = f2 * f4;
   f10 = f10 + f12;
   // end_loop .P1L9;
```

```
.P1L10:
bit clr mode1 0x200000; nop;      // exit SIMD mode
```

Finally, the optimizer modulo-schedules the loop, unrolling and overlapping iterations to obtain highest possible use of functional units. Code similar to the following is generated, if it were known that the loop was executed at least four times and the loop count was a multiple of two.

*Example. Code generated for floating-point scalar product after software pipelining*

```
bit set mode1 0x200000; nop;      // enter SIMD mode
   m4 = 2;
   r4 = dm(i1, m4);
   r2 = dm(i0, m4);
   lcntr = r3, do(pc, .P1L10-1) until lce;
.P1L9:
   f12 = f2 * f4, r4 = dm(i1, m4);
   f10 = f10 + f12, r2 = dm(i0, m4);
   // end_loop .P1L9;
.P1L10:
   f12 = f2 * f4;
   f10 = f10 + f12;
   bit clr mode1 0x200000; nop;      // exit SIMD mode
```

If the original source code is amended to declare one of the pointers with the pm qualifier, the following optimal code is produced for the loop kernel.

*Example. Code generated for floating-point scalar product when one buffer placed in PM*

```
   bit set mode1 0x200000; nop;      // enter SIMD mode
   m4 = 2;
   r5 = pm(i1, m4);
   r2 = dm(i0, m4);
   r4 = pm(i1, m4);
   f12 = f2 * f5, r2 = dm(i0, m4);
   lcntr = r3, do(pc, .P1L10-1) until lce;
.P1L9:
   f12 = f2 * f4, f10 = f10 + f12, r2 = dm(i0, m4), r4 = pm(i1, m4);
   // end_loop .P1L9;
.P1L10:
   f12 = f2 * f4, f10 = f10 + f12;
   f10 = f10 + f12;
   bit clr mode1 0x200000; nop;      // exit SIMD mode
```

## Assembly Optimizer Annotations

When the compiler optimizations are enabled, the compiler can perform a large number of optimizations to generate the resultant assembly code. The decisions taken by the compiler as to whether certain optimizations are safe or worthwhile are generally invisible to a programmer. However, it can be beneficial to get feedback from the compiler