Loop unroll and jam

The loop unroll and jam transformation is primarily intended to increase register exploitation and decrease memory loads and stores per operation within an iteration of a nested loop. Improved register usage decreases the need for main memory accesses and allows better exploitation of certain machine instructions.

Unroll and jam involves partially unrolling one or more loops higher in the nest than the innermost loop, and fusing ("jamming") the resulting loops back together. For unroll and jam to be effective, a loop must be nested and must contain data references that are temporally reused with respect to some loop other than the innermost (temporal reuse is described in “Data reuse”). The unroll and jam optimization is automatically applied only to those loops that consist strictly of a basic block.

Loop unroll and jam takes place at `+O3` and above and is not enabled by default in the HP compilers. To enable loop unroll and jam on the command line, use the `+Oloop_unroll_jam` option. This allows both automatic and directive-specified unroll and jam. Specifying `+Onoloop_transform` disables loop unroll and jam, loop distribution, loop interchange, loop blocking, loop fusion, and loop unroll.

The `unroll_and_jam` directive and pragma also enables this transformation. The `no_unroll_and_jam` directive and pragma is used to disable loop unroll and jam for an individual loop.

The forms of these directives and pragmas are shown in Table 5-7 “Forms of `unroll and jam, no unroll and jam` directives and pragmas”.

Table 5-7 Forms of `unroll_and_jam, no_unroll_and_jam` directives and

pragmas

Language	Form
Fortran	`C$DIR UNROLL_AND_JAM[(UNROLL_FACTOR=`*n*`)]` `C$DIR NO_UNROLL_AND_JAM`
C	`#pragma _CNX unroll_and_jam[(unroll_factor=`*n*`)]` `#pragma _CNX no_unroll_and_jam`

where

`unroll_factor=`*n*

allows you to specify an unroll factor for the loop in question.

NOTE: Because unroll and jam is only performed on nested loops, you must ensure that the directive or pragma is specified on a loop that, after any compiler-initiated interchanges, is not the innermost loop. You can determine which loops in a nest are innermost by compiling the nest without any directives and examining the Optimization Report, described in Chapter 8 “Optimization Report”.

Unroll and jam

Consider the following matrix multiply loop:

```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
    ENDDO
  ENDDO
ENDDO
```

Here, the compiler can exploit a maximum of 3 registers: one for `A(I,J)`, one for `B(I,K)`, and one for `C(K,J)`.

Register exploitation is vastly increased on this loop by unrolling and jamming the `I` and `J` loops. First, the compiler unrolls the `I` loop. To simplify the illustration, an unrolling factor of 2 for `I` is used. This is the number of times the contents of the loop are replicated.

The following Fortran example shows this replication:

```
DO I = 1, N, 2
  DO J = 1, N
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
    ENDDO
  ENDDO
  DO J = 1, N
    DO K = 1, N
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
    ENDDO
  ENDDO
ENDDO
```

The "jam" part of unroll and jam occurs when the loops are fused back together, to create the following:

```
DO I = 1, N, 2
  DO J = 1, N
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
    ENDDO
  ENDDO
ENDDO
```

This new loop can exploit registers for two additional references: `A(I,J)` and `A(I+1,J)`. However, the compiler still has the `J` loop to unroll and jam. An unroll factor of 4 for the `J` loop is used, in which case unrolling gives the following:

```
DO I = 1, N, 2
  DO J = 1, N, 4
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
    ENDDO
    DO K = 1, N
      A(I,J+1) = A(I,J+1) + B(I,K) * C(K,J+1)
      A(I+1,J+1) = A(I+1,J+1) + B(I+1,K) * C(K,J+1)
    ENDDO
    DO K = 1, N
      A(I,J+2) = A(I,J+2) + B(I,K) * C(K,J+2)
      A(I+1,J+2) = A(I+1,J+2) + B(I+1,K) * C(K,J+2)
    ENDDO
    DO K = 1, N
      A(I,J+3) = A(I,J+3) + B(I,K) * C(K,J+3)
      A(I+1,J+3) = A(I+1,J+3) + B(I+1,K) * C(K,J+3)
    ENDDO
  ENDDO
ENDDO
```

Fusing (jamming) the unrolled loop results in the following:

```
DO I = 1, N, 2
```

```
  DO J = 1, N, 4
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
      A(I,J+1) = A(I,J+1) + B(I,K) * C(K,J+1)
      A(I+1,J+1) = A(I+1,J+1) + B(I+1,K) * C(K,J+1)
      A(I,J+2) = A(I,J+2) + B(I,K) * C(K,J+2)
      A(I+1,J+2) = A(I+1,J+2) + B(I+1,K) * C(K,J+2)
      A(I,J+3) = A(I,J+3) + B(I,K) * C(K,J+3)
      A(I+1,J+3) = A(I+1,J+3) + B(I+1,K) * C(K,J+3)
    ENDDO
  ENDDO
ENDDO
```

This new loop exploits more registers and requires fewer loads and stores than the original. Recall that the original loop could use no more than 3 registers. This unrolled-and-jammed loop can use 14, one for each of the following references:

A(I,J)	B(I,K)	C(K,J)	A(I+1,J)
B(I+1,K)	A(I,J+1)	C(K,J+1)	A(I+1,J+1)
A(I,J+2)	C(K,J+2)	A(I,J+3)	A(I+1,J+2)
A(I+1,J+3)	C(K,J+3)		

Fewer loads and stores per operation are required because all of the registers containing these elements are referenced at least twice. This particular example can also benefit from the PA-RISC FMPYFADD instruction, which is available with PA-8*x*00 processors. This instruction doubles the speed of the operations in the body of the loop by simultaneously performing related adds and multiplies.

This is a very simplified example. In reality, the compiler attempts to exploit as many of the PA-RISC processor's registers as possible. For the matrix multiply algorithm used here, the compiler would select a larger unrolling factor, creating a much larger K loop body. This would result in increased register exploitation and fewer loads and stores per operation.

> **NOTE:** Excessive unrolling may introduce extra register spills if the unrolled and jammed loop body becomes too large. Each cache line has a 32-bit register value; register spills occur when this value is exceeded. This most often occurs as a result of continuous loop unrolling. Register spills may have negative effects on performance.

You should attempt to select unroll factor values that align data references in the innermost loop on cache boundaries. As a result, references to the consecutive memory regions in the innermost loop can have very high cache hit ratios. Unroll factors of 5 or 7 may not be good choices because most array element sizes are either 4 bytes or 8 bytes and the cache line size is 32 bytes. Therefore, an unroll factor of 2 or 4 is more likely to effectively exploit cache line reuse for the references that access consecutive memory regions.

As with all optimizations that replicate code, the number of new loops created when the compiler performs the unroll and jam optimization is limited by default to ensure reasonable compile times. To increase the replication limit and possibly increase your compile time and code size, specify the `+Onosize` and `+Onolimit` compiler options.

« prev next »

Loop interchange Preventing loop reordering

Printable version