**For: dwight**
**Printed on: Fri, Jul 16, 1993 13:10:02**
**From book: book**
**Document: ch4**
**Last saved on: Thu, Feb 27, 1992 10:15:38**

# 4 Enhancing Program Performance

## Introduction

This chapter provides some suggestions for enhancing performance of your FORTRAN code on the iWarp system. Before proceeding to these suggestions, note the following program design considerations, which are critical to good performance:

- Pay careful attention to balancing the load between processors as much as possible when you distribute your algorithm among the cells of the array.
- How you devise communication for multi-cell applications is the most important aspect of programming iWarp applications for speed. As much as possible, keep as many cells busy at any given time. Pay close attention to distributing data so that communication between cells is minimized, and synchronizations are only done when necessary.
- Carefully devise your algorithm for the work to be done on a single cell. The best compiler in the world cannot make up for a poorly designed program. Plan your algorithm and implementation carefully. Then, if you need more speed, the following suggestions can help you run your programs faster on the iWarp system.

## General Programming Suggestions

### 1. Use the optimizer

Turn on optimizations using the -O switch on the compiler. The speed improvement will depend on your program. It is generally best to develop and debug your program without the optimizer, and use the optimizer once your code is working. The optimizer may take a considerable amount of time to analyze your program, and you may not want to wait if you are still developing your code. It is also generally preferable to use the debugger on non-optimized code when developing your code. The iwcc -g option suppresses all optimizations.

### 2. Use the profiler

Use the profiler to identify the amount of time spent in various sections of your code. Do this by using the -p compiler switch and running iwprof on the mon.out# file. Profile optimized code, since the optimizer will often improve the performance of many sections of code without any additional work needed by the programmer. You may be able to rewrite sections of your code to be more efficient based on profiling information obtained using iwprof.

For more information about the profiler, refer to the *iWarp Programmer's Guide*.

### 3. Write in a block-structured fashion

The optimizer works best if you write programs with minimal use of goto statements. Avoid the use of goto statements to achieve unusual flow, especially in time-critical code. The global optimizer warns you with the following message if the program is not block structured :

```
"Warning: function foo() has irreducible flow graph"
```

iWarp FORTRAN supports a more structured style of programming by providing the F77 and VMS-compatible control flow constructs IF-ENDIF, DO-ENDDO, and DO WHILE.

4. **Use iteration instead of recursion wherever possible**

   An iterative algorithm will generally offer better performance. Using an iterative algorithm will help avoid subroutine call overhead and offer more opportunities for optimizations, including software pipelining.

5. **Use I/O sparingly**

   I/O is generally expensive on high-performance machines, and this is especially true of parallel architectures. While intercell communication is very fast, I/O is expensive. By limiting the I/O a program performs to only what is essential, you can improve performance.

6. **Move I/O away from the time-critical section of the program**

   You should move I/O as far away as possible from the time-critical parts of a program. Performing WRITE or printf operations in a loop will seriously degrade the performance of the loop. Also, since I/O operations involve subroutine calls, optimizations are usually inhibited in sections of the program containing I/O. On iWarp cells, since all I/O is routed back to some host (or hosts), the penalty for I/O is high.

## Optimizer Suggestions

1. **Do not use the -static option**

   The -static option substantially limits the effectiveness of the optimizer.

2. **Use the inter-file inlining facility**

   Many application programs contain small functions that, from a program maintenance standpoint, should be written as small functions, but will execute faster if the code is inlined to avoid call overhead.

   The solution is to use the iwcc -K option to create intra-file inlined intermediate files containing the body of the functions that should be inlined. You can then compile using these files as input during optimizations.

   The following example illustrates how to inline as many functions as possible between file1.f and file2.f:

```
% iwf77 -K file1.f file2.f
% iwf77 file1.K file2.K file1.f file2.f
```

   In the above example, each original function will be left in the output object code, even if every call site was inlined. If you are sure that every call site will be inlined for all functions in file1.f, you can use the following:

```
% iwf77 -K file1.f file2.f
% iwf77 file1.K file2.K file2.f
```

   If the last line results in an undefined symbol error from the linker, some functions in file1.f could not be inlined at call sites in file2.f. If everything from file1.f does inline, the resulting object code will be smaller than the first example.

Multiple .K files can be concatenated, with the cat command, to form inlining libraries that can be scanned for functions to inline, as in the following example:

```
% iwf77 -K file1.f file2.f
% cat file1.K file2.K > mylib.K
% iwf77 mylib.K file2.f
```

## 3. Use loop constructs that the optimizer can software pipeline

Using the -O switch twice will cause the optimizer, specifically the pipeliner, to print messages on standard error about loops that did not pipeline, and a brief comment as to why not. The messages and explanations regarding why a loop may not pipeline follow below.

*** "foo.f", line 33: loop does not pipeline: constant bounds or index**

Currently, the optimizer cannot pipeline loops that do not have a constant lower bound, upper bound, and step (increment). Future versions of the optimizer will be able to pipeline loops having constant steps but variable loop bounds. It might be faster to transform loops into constant bounded loops.

*** "foo.f", line 33: loop does not pipeline: multiple basic blocks**

The optimizer cannot currently pipeline loops containing flow of control (goto) or conditionals (if). Future versions of the optimizer will have improved abilities to pipeline loops.

*** "foo.f", line 33: loop does not pipeline: too big**

Large loops do not generally benefit from pipelining, so the optimizer does not waste time pipelining certain loops.

*** "foo.f", line 33: loop does not pipeline: too dependent**

Loops may pipeline, but not offer any advantage over the non-pipelined version of the same loop. Such loops may be overly constrained by the flow of data between iterations (there may not be enough to do between data accesses to keep the machine busy).

*** "foo.f", line 33: loop does not pipeline: outer loop**

The pipeliner will only pipeline innermost loops. Future versions of the optimizer may unroll some innermost loops and then pipeline the next level of loop.

## 4. Use the intra-file inlining facility

Intra-file inlining is automatic when you use the -O option. The optimizer will find small functions which can be inlined in the same file, and it will inline them at the call sites when appropriate.

## Other Suggestions

1. **Fill pipelines**

   Make sure there are enough operations in a loop to keep the machine busy. For example, the following code:

```
          do 100 I=1,100
   100    A[I] = A[I-1] + C[I]
          do 200 I=1,100
   200    B[I] = B[I-1] * D[I]
```

   would be better if written as:

```
          do 110 I=1,100
      100 A[I] = A[I-1] + C[I]
      110 B[I] = B[I-1] * D[I]
```

2. **Keep the body of conditionals small inside the innermost loop**

   Inside innermost loops, try to keep the body of the THEN or the ELSE clause of an IF statement small. This is especially important if only one of the clauses is present. The compiled code is most efficient when the THEN and the ELSE clauses have equivalent complexity.

   For example, it is better to use:

```
          do 110 I=1,1000
   100    J = A[I] + B[I] + C[I]
   110    if A[I] .GT. 0.0  A[I] = J
```

   than to use the slower:

```
          do 100 I=1,1000
   100    if A[I] .GT. 0.0 A[I] = A[I] + B[I] + C[I]
```

3. **Limit size of time-critical loops**

   iWarp has a 256-word instruction cache. Any loops that compile to code larger than this will perform poorly compared to smaller loops.

4. **Consider doing I/O in C from FORTRAN programs**

   C I/O is simpler in scope than FORTRAN, and thus requires less memory. FORTRAN I/O has many complex options and capabilities that are not needed for many programs. Sometimes data from FORTRAN programs can be passed to functions written in C which use C's I/O capabilities. C and FORTRAN I/O should generally not be mixed in the same program.

5. **Use FORTRAN NAMELIST features for tuning programs**

   The FORTRAN NAMELIST feature allows you to declare a special type of COMMON block called a NAMELIST. Critical program parameters can be adjusted by modifying the values in the NAMELIST without recompiling the FORTRAN program.

6. **Do not force unnatural data alignments**

   Data is aligned on natural boundaries unless forced to other alignments through EQUIVALENCE or COMMON statements. All data alignments are supported for full VMS compatibility. Since data accesses on unnatural boundaries are done by software event handlers, a significant performance loss may result for programs forcing unnatural alignments.