For: dwight
Printed on: Fri, Jul 16, 1993 10:13:32
From book: book
Document: ch4
Last saved on: Tue, Aug 6, 1991 11:00:27

4 Enhancing Program Performance

Introduction

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

- Carefully distribute your algorithm among the cells of the array, paying careful attention to balancing the load between processors as much as possible.
- Carefully devising 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, paying careful 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. Think out your algorithm and implementation carefully. Then, if you need more speed, the following suggestions can help you get your programs to run faster on the iWarp system.

General Programming Suggestions

Use the optimizer

Turn on optimizations using the -O switch on the compiler. The speed-up achieved 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.

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. To isolate code to levels finer than a subroutine, use the MARK() macro, by placing "#define MARK" and "#include <prof.h>" in your program, and then using the MARK macro to delimit sections of the program to profile. You may be able to rewrite sections of your code to be more efficient based on profiling information obtained using iwprof.

For further information on the profiler, refer to the iwprof man page and the *iWarp Programmer's Guide*.

Write in a block-structured fashion

The optimizer will work best if the program is written with minimal use of "goto". Use of "goto" to achieve unusual flow should be avoided, especially in time-critical code. The global optimizer will warn you with the following message if the program is not block structured :

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

Use arrays and avoid pointers when practical

The iWarp compiler is fairly good at optimizing code that uses pointers. However, it is often better if the code can be written using arrays. This is because array references are more restricted by the language definition than pointer accesses.

Use iteration instead of recursion wherever possible

An iterative algorithm will generally offer better performance. Use of an iterative algorithm will help avoid subroutine call overhead and offer more opportunities for optimizations, including S/W pipelining.

Use memory allocation sparingly

Programs which make frequent calls to memory allocation routines spend a lot of time waiting for the operating system to perform bookkeeping tasks. By allocating larger blocks of memory (for example, arrays of structures instead of single structures), the amount of bookkeeping overhead can be reduced.

Use I/O sparingly

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

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

Use the inter-file inlining facility

An application program often contains a number of small functions that should be written as small functions from a program maintainability standpoint, but which execute faster if the code is inlined to avoid call overhead.

The solution is to use the -K option to compile files to intra-file inlined intermediate files (containing the body of the functions which should be inlined) and then compile everything with these files as input during optimizations.

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

```
% iwcc -K file1.c file2.c
```

```
% iwcc file1.K file2.K file1.c file2.c
```

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.c, you can use the following:

```
% iwcc -K file1.c file2.c
```

```
% iwcc file1.K file2.K file2.c
```

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

Multiple .K files can be concatenated (use “cat”) to form “inlining libraries” which can be scanned for functions to inline as in the following example:

```
% iwcc -K file1.c file2.c
```

```
% cat file1.K file2.K > mylib.K
```

```
% iwcc mylib.K file2.c
```

Use loop constructs that the optimizer can S/W pipeline

Using the -O switch twice will cause the optimizer (specifically the pipeliner) to print information on stderr regarding loops which did not pipeline, and a brief comment as to why not. The following are the messages and explanations regarding why a loop may not pipeline.

*** “foo.c”, line 33: loop does not pipeline: constant bounds or index**

The optimizer cannot currently pipeline loops which 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 in any case, to transform loops into constant bounded loops.

*** “foo.c”, line 33: loop does not pipeline: multiple basic blocks**

The optimizer cannot currently pipeline loops containing flow of control (goto, continue, break), conditionals (if), or sequence operators (comma-op, &&, ||, ?:). Future versions of the optimizer will have improved abilities to pipeline loops.

*** “foo.c”, line 33: loop does not pipeline: too big**

Large loops do not generally benefit from pipelining, thus the optimizer will not waste time pipelining certain loops.

*** “foo.c”, 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.c”, 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.

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

Use -fsingle when double-precision math is not needed

The C language specifies that all floating-point math be done in double precision. This means that the multiply in the following code is done in double precision, after each operand is converted to a double. The result is converted back to single precision. So the following code:

```
register float a, b, c; a = b * c;
```

becomes:

```
fcvt2d A, R0
fcvt2d B, R2
fmul.d R2,R0,R0
fcvt2s R0, C
```

Using -fsingle, the code becomes:

```
fmul.f A, B, C
```

The code generated with -fsingle takes 2 clocks to execute, the previous code takes 16 clocks. This is a dramatic example, but is serves to show the value of -fsingle.

Use of -fsingle directs the compiler to ignore the default, and to perform math using single-precision floating-point arithmetic unless a double-precision value is involved. Math involving double variables will always be done in double precision.

Use of the -fsingle switch will not affect the convention of passing all floating-point parameters and function return values as double-precision values. This is a C language feature which should not be altered.

NOTE

> You can tell the C compiler to pass floats as floats by using the -fnowiden switch. However, it is the user's responsibility to know that all functions in libc and libm which handle floating-point numbers make no provisions for floats. Explicit conversion to doubles would be mandatory when compiling with -fnowiden. For example, when using -fnowiden, ‘printf(“%f\n”,2.0);’ would not work. You must use ‘printf(“%f\n”,(double)2.0);’.

To enhance program performance, use -fsingle where you can, but do not use -fnowiden.

#include <asm/iwarp_fp.h>

You should #include <asm/iwarp_fp.h> when using any of the following math routines:

aint()	anint()	class()
copysign()	drem()	fabs()
finite()	fmod()	ilogb()
infinity()	isinf()	isnan()
isnormal()	issubnormal()	iszero()
logb()	max_normal()	max_subnormal()
min_normal()	min_subnormal()	nextafter()
nint()	quiet_nan()	remainder()
scalb()	signaling_nan()	signbit()
sqrt()		

Using #include <asm/iwarp_fp.h> causes the listed math functions to be inlined instead of being resolved by a library call. This feature cannot be the default behavior of the C compiler without violating the users ability to write their own versions of these functions and link to them. That is why this feature is offered as an include file. If you want to compile your programs with “cc” and “iwcc”, simply use the following:

```
#ifdef __IWCC__

# include <asm/iwarp_fp.h>

#endif
```

Fill pipelines

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

```
for (i=0; i<100; i++) a[i] = a[i-1] + c[i];

for (i=0; i<100; i++) b[i] = b[i-1] * d[i];
```

would be better if written as:

```
for (i=0; i<100; i++) {

   a[i] = a[i-1] + c[i];

   b[i] = b[i-1] * d[i];
}
```

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 will be most efficient when the THEN and the ELSE clauses are of roughly equivalent complexity.

For example, it is better to use:

```
for (i=0; i<1000; i++) {
   int temp = a[i]+b[i]+c[i];
   if (a[i] > 0.0)
         a[i] = temp;
}
```

than using the slower:

```
for (i=0; i<1000; i++) {
   int temp;
   if (a[i] > 0.0)
         a[i] = a[i]+b[i]+c[i];
}
```

Limit size of time-critical loops

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