Audio & MIDI
Home
General Questions
- ADAT Optical Links

Applications
- IRIX 6.2 Bundled
- Commercial
- Public

Software Synthesis
Desktop Sounds
Audio Development
- Pointer Aliasing

Performance Implications of Pointer Aliasing

by Doug Cook
Silicon Graphics, Inc.
August, 1997

Pointer aliasing can have a severe impact on program performance. Understanding its implications is critical to writing high-performance code. This document provides a brief introduction to the problem, and suggests several approaches to solving it through source-code restructuring, compiler options, and C or C++ language extensions.

Aliasing

Here’s a brief overview of aliasing. Consider the following function:

```
void
process_data(float *in, float *out, float gain, int nsamps)
{
        int i;

        for (i = 0; i < nsamps; i++) {
                out[i] = in[i] * gain;
        }
}
```

In C or C++, it is legal for the parameters *in* and *out* to point to overlapping regions in memory. When this happens, *in* and *out* are said to be *aliases*. When the compiler optimizes the function, it does not in general know whether *in* and *out* are aliases. It must therefore assume that any store through *out* can affect the memory pointed to by *in,* which severely limits its ability to reorder or parallelize the code (For some simple cases, the compiler could analyze the entire program to determine that two pointers cannot be aliases. But in general, it is impossible for the compiler to determine whether or not two pointers are aliases, so to be safe, it must assume that they are).

In the code above, the compiler must issue the instructions in roughly the following order. The code here is simplified a little bit:

```
top:    lwc1     $f1,0(v1)          # load in[i]          0
        mul.s    $f1,$f1,$f0        # multiply by gain    1
        addiu    v1,v1,4            # increment in        1
        swc1     $f1,0(v0)          # store the result    5
        addiu    v0,v0,4            # increment out       6
        bne      v0,a0,top          # see if we're done   7
```

In this code, the compiler decided that there was no benefit to unrolling the loop because of the limited options for reordering. This version theoretically requires 8 cycles per iteration.

For this example, we use the MIPS R5000 processor, an excellent processor to use for this type of benchmark: though it can be very fast, it is quite sensitive to instruction scheduling. Because the R10000 dynamically reorders instructions, it is considerably less sensitive to compiler scheduling, and thus less suitable for this type of benchmark. A few pertinent notes about the R5000:

1. The single-precision multiply or multiply/add (mul.s or madd.s) can be issued every cycle, but it has a 4-cycle latency. Therefore pipelining the floating-point instructions is critical to good performance.
2. A floating-point instruction can execute in parallel with an integer instruction.
3. lwc1 and swc1 (floating-point loads and stores) count as integer instructions.

The fundamental problem in this code example is that the compiled code has to wait for the result of a multiply and store it to memory before it can load the next value, because a store to *out[i]* might affect a subsequent load from *in[i+k]*. This completely prevents the compiler from pipelining the instructions, or from taking advantage of the parallelism in the processor.

Here is the performance benchmark for this version of the program (compiled with the MIPSPro 7.2 beta compilers, using **-n32 -mips4 -r5000 -O3**). The benchmark simply calls **process_data** a large number of times and looks at the elapsed clock time. All the data fits in the cache.

```
macondo 2% gainex 100000
22891752.000000 samps/sec (519.087341 voices @ 44.1kHz)
```

The numbers here assume that the data is audio, and measure the number of simultaneous **process_data** calls ("voices") that could be made in real-time at 44.1 kHz (44100 audio samples per second). But the techniques used here apply to any kind of data. The results will be more dramatic with floating-point data, where the independent floating-point unit provides more opportunities for instruction-level parallelism, and where the difference between FP instruction latencies (sometimes several cycles) and FP instruction issue rates (usually 1 per cycle) provides more opportunities for software pipelining.

Improving Performance Through Code Restructuring

You can often improve code performance by manually unrolling the loops in your code. This is a conventional method for speeding up loop performance. An unrolled loop is often faster because it performs fewer loop-bound tests. But if done properly, loop unrolling can also minimize the effects of pointer aliasing.

The basic idea is to unroll a loop and defer the stores as long as possible in the unrolled loop. Then the compiler is free to reschedule all the instructions between the loads and the stores. Here is the previous example using temporary variables to defer the stores:

```
void
process_data(float * in, float * out, float gain, int nsamps)
{
        int i;
        float y,y1,y2,y3;

        for (i = 0; i < nsamps; i+= 4) {
                y = in[i] * gain;
                y1 = in[i+1] * gain;
                y2 = in[i+2] * gain;
                y3 = in[i+3] * gain;
                out[i] = y;
                out[i+1] = y1;
                out[i+2] = y2;
                out[i+3] = y3;
        }
}
```

Note that this version has the additional restriction of requiring that ***nsamps*** *be a multiple of 4. The common scheme for avoiding this restriction is to add a separate loop which does the leftover 0-3 iterations. Since that is irrelevant to the aliasing discussion, it is omitted here.*

```
macondo 1% /usr/tmp/gainex.new 100000
64134524.000000 samps/sec (1454.297607 voices @ 44.1kHz)
```

This method has some drawbacks. The compiler still cannot reorder operations past the stores, because it still does not know if the pointers are aliased. There are just more operations to reorder between the loads and stores. In other words, the store to **out[i]** could still affect subsequent values of **in[i+k]**, *except* **in[i]**, **in[i+1]**, **in[i+2]**, and **in[i+3]**, because those have already been loaded. So the compiler is only free to reorder the calculations based upon **in[i]**,

in[i+1], **in[i+2]**, and **in[i+3]**.

If we unroll the loop in larger blocks, we may get better performance. However, at some point, this hand-unrolling will backfire: the number of temporary variables will exceed the number of readily-available registers, or the loop's cache behavior will change, and its performance will degrade. The optimal block size can be difficult to determine, and changes from processor to processor, making it hard to write portable code.

Also, the hand-unrolled code becomes more difficult to read and maintain. What we'd really like is to keep the code simple and tell the compiler that pointers cannot be aliased, then let the compiler do the loop unrolling and scheduling as it sees fit. There are a couple of ways to do this.

Optimizer Options

Many compilers have options to override their default aliasing assumptions. The MIPSPro 7.X compilers support a set of **-OPT:alias** command-line switches for this purpose. For example, **-OPT:alias=restrict** implements the following semantics: Memory operations dereferencing different named pointers in the program are assumed not to alias with each other, nor with any named scalar in the program.

Example: if p and q are pointers, *p does not alias with *q, nor does *p alias with p, nor does *p alias with any named scalar variable.

There is also a switch, introduced in MIPSPro 7.2, called **-OPT:alias=disjoint**. This implements the following semantics: Memory operations dereferencing different named pointers in the program are assumed not to alias with each other, and in addition, different "dereferencing depths" of the same named pointer are assumed not to alias with each other.

Example: If p and q are pointers, *p does not alias with *q, nor does *p alias with **p, nor does *p alias with **q.

Both these switches make promises to the compiler about the behavior of the program; if either switch is enabled, programs violating the corresponding aliasing assumptions are liable to be compiled incorrectly.

Often these options are difficult to use; in a typical application, some pointers are aliases, and others are not. You may be able to move certain functions to separate source files and compile just those files with these switches. In some cases, you cannot use these switches at all, because within a single function, some pointers are aliases and others are not.

The *restrict* keyword

Fundamentally, however, this problem is an issue with the C and C++ languages themselves, because there is no way to specify which pointers may be aliased, and which may not. The Numerical C Extensions Group / X3J11.1 proposed a **restrict** keyword for the C language to address the issue (see *Restricted Pointers in C, Numerical C Extensions Group / X3J11.1, Aliasing Subcommittee, June 1993*) The **restrict** keyword is a very clean, high-performance way of addressing the pointer aliasing problem, without many of the drawbacks of the other methods.

The **restrict** keyword tells the compiler to assume that dereferencing the qualified pointer is the only way the program can access the memory pointed to by that pointer. Hence loads and stores through such a pointer are assumed not to alias with any other loads and stores in the program, except other loads and stores through the same pointer variable. Here is the previous code example rewritten to indicate that *in* and *out* cannot overlap:

```
void
process_data(float *  restrict  in, float *  restrict  out, float g
{
        int i;

        for (i = 0; i < nsamps; i++) {
                out[i] = in[i] * gain;
        }
}
```

Another example may clarify things a bit. In this example, qualifying **a** with **restrict** is sufficient to indicate that **a** and **c** are not aliased. In other words, you need only qualify the pointers being stored through.

```
float x[ARRAY_SIZE];
float *c = x;

void f4_opt(int n, float * restrict a, float * restrict b)
{
  int i;
  /* No data dependence across iterations because of restrict */
  for (i = 0; i < n; i++)
    a[i] = b[i] + c[i];
}
```

The MIPSPro 7.2 compilers accept the restrict keyword with the **-LANG:restrict** option, and use it to perform aggressive optimization. The 7.1 compilers do *not* support the restrict keyword, though they will not issue warnings if you use it. You can determine which compilers you have by using the command "cc -n32 -version". Additionally, the compilers will set the _COMPILER_VERSION preprocessor macro to indicate which version of the compiler is in use.

Here are the results of our original example using the **restrict** keyword. This code is over 3 times as fast as the original version, and somewhat faster than even the hand-unrolled version. For this simple example, performance would be identical using either the **-OPT:alias=restrict** option or the **restrict** keyword.

```
macondo 3% ./gainex 100000
70395424.000000 samps/sec (1596.268066 voices @ 44.1kHz)
```

In this case, the compiler was able to unroll the loop and freely reschedule instructions within the loop to maximize performance. Here is part of the unrolled loop which processes 4 samples. Note that all the multiplies are pipelined and overlapped with integer instructions as well. The result is that this version takes closer to 2 cycles per sample.

```
  [...]
                                      Cycle
  lwc1      $f0,24(a1)                0
  mul.s     $f4,$f2,$f5               1
  lwc1      $f1,20(a1)                1
  mul.s     $f3,$f0,$f5               2
  lwc1      $f0,16(a1)                2
  mul.s     $f1,$f1,$f5               3
  lwc1      $f2,44(a1)                3
  beq       a1,a0,0x10000ed4          4
  mul.s     $f0,$f0,$f5               5
  swc1      $f4,28(a2)                5
  swc1      $f3,24(a2)                6
  swc1      $f1,20(a2)                7
  swc1      $f0,16(a2)                8
  [...]
```

Since the **restrict** keyword is not recognized by many compilers, including older MIPS compilers, for portability you may want some construct like:

```
#if (defined(__sgi) && _COMPILER_VERSION >= 720)
#define RESTRICT restrict
#else
#define RESTRICT
#endif

process_data(float * RESTRICT in, float * RESTRICT out, float gain,
```

This construct will attempt to use **restrict** only in the presence of the MIPSPro 7.2 or later compilers.

The following table summarizes the results for this simple example using the four techniques discussed here: the original code; the hand-unrolled loop; with -OPT:alias=restrict; and with -LANG:restrict. We've included R10000 results for comparison. As mentioned earlier, the performance improvements for the R10000 are less dramatic; the code gets about 16% faster using **restrict** . The R5000 results used **-O3 -r5000 -mips4 -n32** ; the R10000 results used **-O3 -r10000 -mips4 -n32**.

	original	hand-unrolled	-OPT:alias=restrict	-LANG:rest
180MHz R5000	519	1454	1585	1596
195MHz R10000	1765	1891	2051	2051

From this, we can see that the **restrict** keyword is a clean, high-performance approach to writing number-crunching code.

Acknowledgments

Thanks much to **Raymond Lo** for his review, and to **Rohit Chandra** for precise descriptions of the compiler behavior and for some of the code examples.