Due: Monday, Mar 1, 2010. 11:59pm (no extensions).

What to submit: Upload a single ASCII file with your answers.

This exercise is intended to reinforce the content of the lectures relating to optimizations. For this exercise, please obtain your results on any of the machines on the rlogin cluster. Do not use a McB 124 machine.

1. Benchmarking combine. In the first part of the exercise, you will reproduce the results for the sum/prod, integer/float combiners on the Intel Netburst architecture used in our current lab machines. The directory `cs3214/code` contains a simplified and cleaned up copy of the code samples published at [http://csapp.cs.cmu.edu/public/1e/code.html](http://csapp.cs.cmu.edu/public/1e/code.html). Start by making a copy of that directory. (Use `cp -R cs3214/code ...`). Then build the benchmarks in the `opt` subdirectory by running the command `make`. This should build programs isbench, ipbench, fsbench, and fpbench. The benchmark harness contains more variations on unrolling and iteration splitting than were presented in class or in the book.

   (a) Complete the following table:

   | Integer | Float |
   | + * | + * |
   |---------------------------------+----+----+----+----|
   | Full abstraction -g | | | | |
   | Full abstraction -O2 | | | | |
   | Moving vec_len out of the loop | | | | |
   | Remove vector accessor method | | | | |
   | Accumulate in temporary | | | | |
   | Unroll 4x | | | | |
   | Unroll 16x | | | | |
   | Unroll 4x2 | | | | |
   | Unroll 8x4 | | | | |
   | Unroll 8x8 | | | | |

   Find the function corresponding to each row, and enter the measured CPE value. Your table should have 50 entries. If there is both an array and pointer code version, use the faster of the two.

   (b) The current code places the definitions of all methods of the ADT vec_rec into vec.c, which resides in a separate compilation unit. Rearrange the code such that these functions can be inlined. To accomplish this, the functions’ definitions now found in vec.c must be seen in the combine.c compilation unit. A possible way to ensure inlining is to place the functions as ‘static’ functions into the vec.h header file. You may need to remove vec.c from the Makefile to prevent it from being compiled and linked.

   Please verify that the functions are inlined, either using objdump -d on the executable or by compiling with -S and examining the assembly code. You may need to specify `-finline-functions`. gcc has a complicated set of rules that
guard which functions are inlined at the -O1, -O2, and -O3 optimization levels, based on whether the function is declared static or not, based on whether it is called only once, and based on whether it is “simple.” Please keep the -O2 optimization level.

Provide the CPE’s you’ve obtained for combine1, combine1l, combine2, combine3, and combine4 for all 4 cases (integer/float, sum/prod).

In your opinion, does the programmer pay a significant abstraction penalty after inlining is taken into account?


Find the published latency/throughput values of the arithmetic operation required for each vector element! (Intel uses the term ‘throughput’ to denote cycles/issue, that is, how many cycles must pass between injecting successive operations into the pipeline.)

For each of the 4 type/operation combinations, determine whether any of the combiners included in combine.c make optimal use of the throughput of the functional unit executing the arithmetic operation involved (i.e. addition or multiplication).

(d) **Compiler Unrolling.** gcc can perform loop unrolling, though it does not do so by default. Add `-funroll-loops` (on top of `-O2` and while preserving inlining!). Examine the resulting assembly code and answer the following questions:

How often did gcc unroll each loop?

Did it lead to the same performance gains as manual unrolling?

(e) **Auto Vectorization.** Recent versions of gcc include a feature where the compiler automatically determines opportunities for vectorization (rather than relying on the programmer to use `__attribute__((vector_size(...) ))` or SSE intrinsics as shown in lecture). Compile the inlined integer/sum benchmark with this feature by adding `-fno-vectorize -msse2` and, optionally, `-fno-vectorizer-verbose=5` to see detailed information about what was vectorized and what was not.

You should see a performance improvement for combine1l and combine4.

How many CPE does the auto-vectorized code require for integer/sum in this case?

2. **Common Subexpressions.** Consider these two examples:
int sum_neighbors(int *a, int n, int i, int j)
{
    int up = a[n*(i-1) + j];
    int down = a[n*(i+1) + j];
    int left = a[n*i + j - 1];
    int right = a[n*i + j + 1];
    return up + down + left + right;
}

int sum_neighbors1(int *a, int n, int i, int j)
{
    int up = a[n*i - n + j];
    int down = a[n*i + n + j];
    int left = a[n*i + j - 1];
    int right = a[n*i + j + 1];
    return up + down + left + right;
}

Is gcc -O2 able to recognize the common subexpression \(n \times i + j\) in each case? Show your work!

3. Optimizing Away Procedure Calls. In lecture we had discussed compilers must faithfully preserve the semantics of the program even for inputs the programmer may consider pathological. Consider this example from lecture:

void lower1(char *s)
{
    int i;

    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

Consider how strlen() is implemented, and assuming that strlen() is defined in the same compilation unit, could a compiler move the call to strlen() out of the loop? Assume that the compiler is infinitely sophisticated, but is not allowed to produce code that would yield different results for any input, even unlikely input.

If no, say why it is impossible, given the constraint to preserve the semantics of the program at all times.

If yes, explain how the compiler could prove such a transformation does preserve the semantics of this program.

Hint: it’s not sufficient to just argue about strlen(), e.g., saying that “strlen() doesn’t change the string” is not a sufficient answer.

4. Array Bounds Checking Detractors sometimes criticize Java for the potential performance penalty imposed by mandatory array bounds checking. Proponents claim that array bounds checks can be optimized. Compile the following Java class with gcj, the GNU Java compiler. Use -O2. Use the version on the lab machines.
public class array {
    int sum(int []a) {
        int s = 0;
        for (int i = 0; i < a.length; i++)
            s += a[i];
        return s;
    }

    int sum_range_checked(int []a, int from, int to) {
        if (from < 0 || to < from || to >= a.length)
            throw new ArrayIndexOutOfBoundsException();
        int s = 0;
        for (int i = from; i < to; i++)
            s += a[i];
        return s;
    }
}

Hint: to decode the mangled labels you’ll see, such as
_ZN4java4lang30ArrayIndexOutOfBoundsExceptionC1Ev, use the program c++filt like
so:

echo _ZN4java4lang30ArrayIndexOutOfBoundsExceptionC1Ev |
c++filt -s java

Which array bounds check(s) can this compiler optimize away?
Which array bounds check(s) would an infinitely sophisticated compiler be able to optimize away?