Optimization Techniques in C

Fall 2012

1 Writing Cache-Friendly Code

Cache-friendly code is code whose organization and design utilizes the machine's cache in the most efficient way possible.

Caches, as you have seen in lecture, store *lines* containing bytes that are located consecutively in main memory. This design is based on the principle of *spatial locality* — that is, those memory regions that are physically closer together are more likely to be accessed within a short time of one another than those spread more thinly in RAM. In order to write cache-friendly code, a programmer must respect this principle, by writing programs that primarily access closely-grouped memory locations sequentially or simultaneously.

The following is an example of cache-*un*friendly code:

```
for (j = 0; j < NUM_COLS; j++) {
    for (i = 0; i < NUM_ROWS; i++) {
        array[i][j] += 1;
    }
}
```

This code visits every entry in a two-dimensional array and adds 1 to each. Recall that two-dimensional arrays in C are actually stored as one-dimensional arrays, with rows intact and listed one after the other, such that the underlying index of an element can be computed by `row * rowlen + col` (In technical terminology, this is called *row-major order*). When we iterate through the array in the above example, however, we iterate in *column-major order*[1]. Unless our cache lines are very large, or the array is very small, this poses a problem, because sequential memory accesses will each be separated from one another by `NUM_COLS`. Thus, if `NUM_COLS` is greater than the length of a cache line, and `NUM_ROWS` is greater than the number of lines in the cache, we run into the following problem:

On each of the first n iterations (where n is the number of lines in the cache), the program suffers a cache miss, since the next byte needed is not in any of the cache lines that are present in memory. This is not a huge problem because we started with a cold cache, and some number of misses were bound to be required in order to warm it up. However, on the next iteration, we again suffer a miss, and since the cache is now full, one of the current cache lines must be evicted to make room for the new line. Since each read requests an address that is on a different line and there are not enough lines to store the entire array in the cache, a miss occurs on every iteration, slowing the program down considerably. This is unacceptable — if we're to have a miss each time we access memory, we might as well not have the cache at all. Luckily, there is a simple fix: reverse the order in which the rows and columns of the array are traversed:

[1] Which is just like row-major order, except that the columns are intact and listed one after another.

```
for (i = 0; i < NUM_ROWS; i++) {
    for (j = 0; j < NUM_COLS; j++) {
        array[i][j] += 1;
    }
}
```

Now we are iterating through the columns on the inside loop, and the rows on the outside. Turning our attention back to the cache, we see a different pattern: The first memory access is a miss, since the cache is cold. However, the next $n - 1$ accesses are hits, since the requisite values were loaded into the cache with the first as part of the cache line. Only then do we encounter another miss. This pattern continues, and misses become much less frequent than before.

When an array must be iterated over in both row-major and column-major order, *code blocking* is a good way to reduce cache-misses. Code blocking involves dividing the array up into blocks which have dimensions that are no larger than the size of a cache line. In this way, a part of several rows will be loaded into the cache during the row-major traversal, and then all of their columns will be processed during the column-major traversal. The process is then repeated on the next block. This leads to only one miss per row per block.

2 Loop Unrolling

Loop unrolling, another optimization technique, entails repeating the code that will be executed multiple times in a loop, so that fewer loop iterations need to be made. The concept is simple if counterintuitive, since programmers are used to using loops to avoid exactly this sort of repetition (which is why this type of optimization is usually done by the compiler, rather than by humans).

Here is a simple example. In the unoptimized program, a `while` loop is used to call a certain function some finite, but large, number of times. Loops like this one are very common, and you have surely written numerous similar code snippets over the course of your CS career:

```
int i = 0;
while (i < num) {
    a_certain_function(i);
    i++;
}
```

There is a problem here, however: loops incur a certain amount of overhead, especially when each iteration does only a small amount of work (such as calling a single function). The conditional branch at the top of the loop, and the unconditional return jump at the bottom, are both operations that take a non-trivial number of cycles to complete, and thus slow the program down. Thus, assuming that `num` is a multiple of 4, the following optimized code might be preferable to the original:

```
int i = 0;
while (i < num) {
    a_certain_function(i);
    a_certain_function(i+1);
    a_certain_function(i+2);
    a_certain_function(i+3);
    i += 4;
}
```

In this version, although the actual code visible to the user (and the binary) is longer, the program will take less time to run, because it need only loop `num`/4 times to compute the same results as the original version.

Loop unrolling does have one major drawback: it makes programs significantly longer (in terms of lines of code and size of the compiled binary) than they would be otherwise. As in many cases in computer science, the decision of whether to unroll or not to unroll is a tradeoff between time and program size. For some applications, the added binary length might be worth the decreased runtime; however, there are also applications for which a smaller binary is more important than a shorter runtime (for example, if one were coding for a machine with very little RAM, or were writing a program that was already very large).

3 Inlining

Inlining is the process by which the contents of the inlined function are effectively copied and pasted into the function which calls it, replacing the traditional call to that function. This form of optimization avoids the overhead of function calls, by eliminating the need to jump, create a new stack frame, and reverse this process at the end of the function.

For example, consider the following piece of code:

```
int get_random_number(void) {
    return 4; // chosen by fair die roll
              // guaranteed to be random
}

int main(int argc, char **argv) {
    int a = get_random_number();
    printf("%d\n", a);
        return 0;
}
```

Making a whole function call just to retrieve the number 4 seems a bit inefficient, and `gcc` would agree with you on that. To optimize this function, the code inside of `get_random_number()` (which happens to just be the integer 4) would be substituted for the function call in the body of `main()`. Thus, after optimization, the function would look like this:

```
int main(int argc, char **argv) {
    int a = 4;
    printf("%d\n", a);
        return 0;
}
```

A smart compiler might even eliminate the temporary variable `a`, although for the sake of this example let us assume that it does not. The resultant program does not make any function calls; however, it retrieves the same result as weas provided in the original version. This program will run much faster than the original, given that it doesn't have to jump, create a new stack frame, and undo all of that time-consuming computation just to get the number 4.

Inline substitution, like loop unrolling, may seem counterintuitive at first, since conventional wisdom is that one should always try to separate out code into distinct functions. However, inlining is akin to treating functions as macros, which doesn't violate the principle of separation of functionality per se. Also, since inlining is almost always performed by the compiler, separation from the programmer's perspective is maintained.

Once inlining is brought into the picture, the distinction between "functions" and "macros" may seem a bit fuzzier. However, there remain important semantic differences between the two. Most importantly, a function always defines its own *scope* — that is, local variables defined within a function may only be accessed from within that function (in C, this is true of any block of code surrounded by curly braces). When inlining more complex functions with lots of local variables, things get much more complex. We won't get into the specifics here (this is a systems course, not a semantics course), but let it suffice to say that the process is a bit more complicated than simply copying and pasting the contents of one function into another.

4 Writing Pipeline-Friendly Code

Optimized programs must take into account how the machine's pipeline will affect their performance. As you have seen in lecture, the nature of a pipeline makes it such that, after some instructions (such as conditional branches), it is impossible to predict with perfect accuracy which instruction will be executed next. However, since knowing the following instruction, even some of the time, helps with efficiency, processor designers make every effort to anticipate the next instruction as often as possible in these cases.

Branch prediction, however, only gets it right so often, so it's up to the programmer to make sure that it is easy for the processor to predict branches. There are a number of different branch prediction schemes, which are detailed in your book and in lecture.

However, as was mentioned in the section on loop unrolling, the best solution is to avoid branching as much as possible, since mispredicted branches are fairly expensive, and it is difficult to know the branch prediction scheme of the machine on which your code will be run. In many processors, the branch predictor is good enough that there is very little the programmer can do to help — thus, while you shouldn't go to too great lengths to avoid branches, you should be careful not to overuse them when there is a cheaper alternative (such as the "ternary operator", `<cond> ? <true> : <false>`.

5 Code Motion

Code motion involves identifying bits of code that occur within loops, but need only be executed once during that particular loop. A good example of such an operation is the declaration of a variable — assuming that the variable is properly initialized (if it needs to be) each time around. A variable declared outside of a loop is still in scope inside the loop, so the code will behave the same way.

For example, the following code reverses a string, using a temporary variable that is re-declared each time around:

```
void reverse(char *rry) {
    int i = 0, len = strlen(rry);
    while (i < len) {
        char tmp;
        len--;
        tmp = rry[len];
        rry[len] = rry[i];
        rry[i] = tmp;
        i++;
    }
}
```

However, there is no reason to declare this variable on every iteration of the loop. Thus, we can optimize the function by moving the declaration of the variable `tmp` out of the loop:

```
void reverse(char *rry) {
    int i = 0, len = strlen(rry);
    char tmp;
    while (i < len) {
        len--;
        tmp = rry[len];
        rry[len] = rry[i];
        rry[i] = tmp;
        i++;
    }
}
```

The new version of the function only needs to allocate memory on the stack for the temporary variable once, saving time in its execution.[2] Since it takes time to allocate memory for a variable, it is less efficient to have a variable continuously move in and out of scope. Conversely, if a variable is declared and then stays in scope until it is no longer needed, stack space can be allocated for it and other variables in one fell swoop at the beginning of the function, bringing down the number of operations required to perform the computation.

[2] In practice, in a function that short, `tmp` would probably be assigned to a register. However, there might still be performance gains from keeping it in scope, rather than having it go out of scope and come back in repeatedly.