Performance Engineering of Software Systems September 14, 2010
Massachusetts Institute of Technology 6.172
Professors Saman Amarasinghe and Charles E. Leiserson Handout 5

Bit Hacks

Project 1

Last Updated: October 3, 2010

This project provides you with an opportunity to experiment with ***word-level parallelism****: the abstraction of a computer word as a vector of bits on which bitwise arithmetic and logical operations can be performed. Word-level parallelism — more colloquially called* ***bit hacks*** *— can have a dramatic impact on performance. This project will also give you the opportunity to develop and practice your C programming skills.*

Please read Handout 4, which describes the grading policy for projects.

Introduction

Upon arriving for your first day of work at Snailspeed Ltd., you are given two programs written by Snailspeed's former programmer, Harvey Crimson. Crimson was asked to leave Snailspeed because his programs did not measure up to Snailspeed's eponymous standard. (He is now pursuing a career in fried-potato engineering.) Thus, it falls to you to improve and maintain the two programs.

Upon examining Harvey's code, you discover that although these programs seem to be correct (that is, they produce the intended results), they are far from efficient. Cleaning up the code and making some small tweaks will probably result in a measurable performance gain, but you want to impress your boss. Your challenge is to exploit word-level parallelism to greatly speed up Harvey's programs.

Do not perform multithreaded parallelization or hardware-specific optimization, which will come later in the course. Please code in standard C — for example, no in-line assembly directives to the compiler. Furthermore, please don't use compiler intrinsics or libraries that use assembly or intrinsics to achieve the same effect. We want to see you use standard C. The x86 machines on which you will be running are little endian. You need not make your code portable to big-endian machines.

Code structure

The two programs you are responsible for improving are in the `everybit` and `pentominoes` directories. The timing and testing system resides in `main.c`, which calls your routines. Do not make modifications to `main.c`, as it will be replaced with a fresh copy when the staff runs your code. By the same token, please do not remove or change the signatures of any of your functions that `main.c` calls. In short, the provided `main.c` should always compile against your code and execute correctly!

Building the code

In each program subdirectory, you can build the code by typing

```
% make
```

Note that to build with debugging symbols and assertions (useful for debugging with gdb), you must build with

```
% make DEBUG=1
```

After building, you can then, for example, run the everybit binary with

```
% ./everybit
```

Most of the binaries require a set of parameters to run. If you fail to specify the required parameters, the program prints out a "usage" statement.

Testing

Testing is a fundamental component of good software engineering. You will find that having a regression suite for your projects speeds your ability to make performance optimizations, because it is easy to try something out and localize the bug, rather than spending hours trying to figure out where it is or — worse — never realize that you have a bug in your code.

The staff has provided you with a basic framework for writing tests, as well as a few rudimentary test cases. These tests do not provide adequate coverage, and we will be looking for you to add more test cases. Your goal with testing is to find bugs not only in your own code, but in the code written by others in the class. In particular, your regression suite will be run against other teams' projects. If another team's buggy program passes all the tests in your regression suite, you will lose points. The harder it is to find a bug, the more points the bug is worth, so your goal should be to find as many corner cases as possible.

Write tests for all the edge cases in each of the programs. Include some general tests, but don't go overboard. Also, avoid nondeterministic test cases, such as generating randomized input – it unnecessarily complicates debugging and reproducing test failures.

Whenever you find a bug in your own code that passes all the tests in your current regression suite, it is a good idea to add a test case for that bug, so you can immediately catch it if it shows up again. In fact, many software projects require that every bug fix submission be accompanied by a test case for the bug!

You can run the test suite for any project by running

```
make test
```

in the project directory. Add additional test cases to the `tests.c` file, and check them in with your code using `git add`. If any of your test cases require input files to work, put them in a `tests/` directory.

Please refer to Handout 4 for guidelines on the submission and grading of test suites.

1 Every bit counts

Many programs operate under tight constraints, and getting respectable performance under these circumstances can be especially challenging. Part of Snailspeed Ltd.'s product portfolio targets mobile and embedded devices such as cell phones. One such application requires that various operations on bit strings be performed within a large data buffer. Since the buffer is deliberately as large as possible, only a small amount of auxiliary memory is available to implement of these functions. In particular, your implementation should use only a constant amount of auxiliary memory, independent of the size of the data buffer and of other parameters, such as the rotation amount.

Take a look at `bitarray.h` and `bitarray.c`. This code implements the functions needed to allocate, access, and process large strings of bits while using a minimum of memory. In the implementation, bits are packed 8 per byte in memory, but can be accessed individually through the public `bitarray_get()` and `bitarray_set()` functions.

Your job is to improve the `bitarray_rotate()` and `bitarray_count_flips()` functions programmed by Harvey. For the following tasks, do not use `malloc()` or other memory-allocation functions, and do not call `bitarray_new()` from within your implementations. You can allocate small buffers on the stack or in the BSS section (e.g., global arrays). Do not use more than a constant amount of additional memory for each of the tasks below.

Your implementations of `bitarray_rotate()` and `bitarray_count_flips()` will be considered correct if the contents of the bitarray, as accessed through `bitarray_get_bit_sz()` and `bitarray_get()`, is the same as after running Harvey's implementation.

1.1 Rotating a bit string

The function `bitarray_rotate()` rotates a string of bits within a bit array by some amount to the left or right. See the documentation in `bitarray.h`. Harvey's implementation is slow, however, performing lots of one-bit rotations over and over again until the correct degree of rotation is achieved. If you try to run

```
./everybit -r
```

you will see that a couple of rotations on even a small buffer can take several minutes to run. Your task is to come up with a more efficient implementation for `bitarray_rotate()`, given the rules stated earlier. Your write-up and documentation should explain clearly and succinctly how your method works.

The natural way you might think to perform a k-bit circular left rotation of a string of length n, while using only a constant amount of auxiliary memory, is to save the 0th bit, and then copy kth bit to index 0, the $2k$th bit to index k, the $3k$th bit to index $2k$, etc., all mod the length n of the string being rotated, until you return to 0, at which point you store the saved 0th bit. If k and n are relatively prime, a single loop suffices. If not, however, the code becomes more complicated.

There is a more clever way, however, which involves moving every bit twice, but which is simpler to code. The string to be rotated can be considered to be of the form ab, where a and b are bit strings and a has length k. We wish to transform ab to ba. Observe the identity $(a^R b^R)^R = ba$, where R is the operation that reverses a string. The "reverse" operation can be accomplished using only constant storage. Thus, with 3 reversals of bit strings, the string can be rotated.

1.2 Counting bit flips

The function `bitarray_count_flips()` counts the number of bit transitions from 0 to 1 and vice versa in a string of bits within a bit array. For example, the bit sequence 0000 has no transitions, the sequence 0001 has one transition, and the sequence 0010 has two transitions. (See the documentation in `bitarray.h`.) As you can see from the code, Harvey's implementation is slow. You can run

```
./everybit -f
```

and see a bunch of flip count operations that take several minutes to run on a large buffer.

Improve the performance of `bitarray_count_flips()`. Explain how your method works.

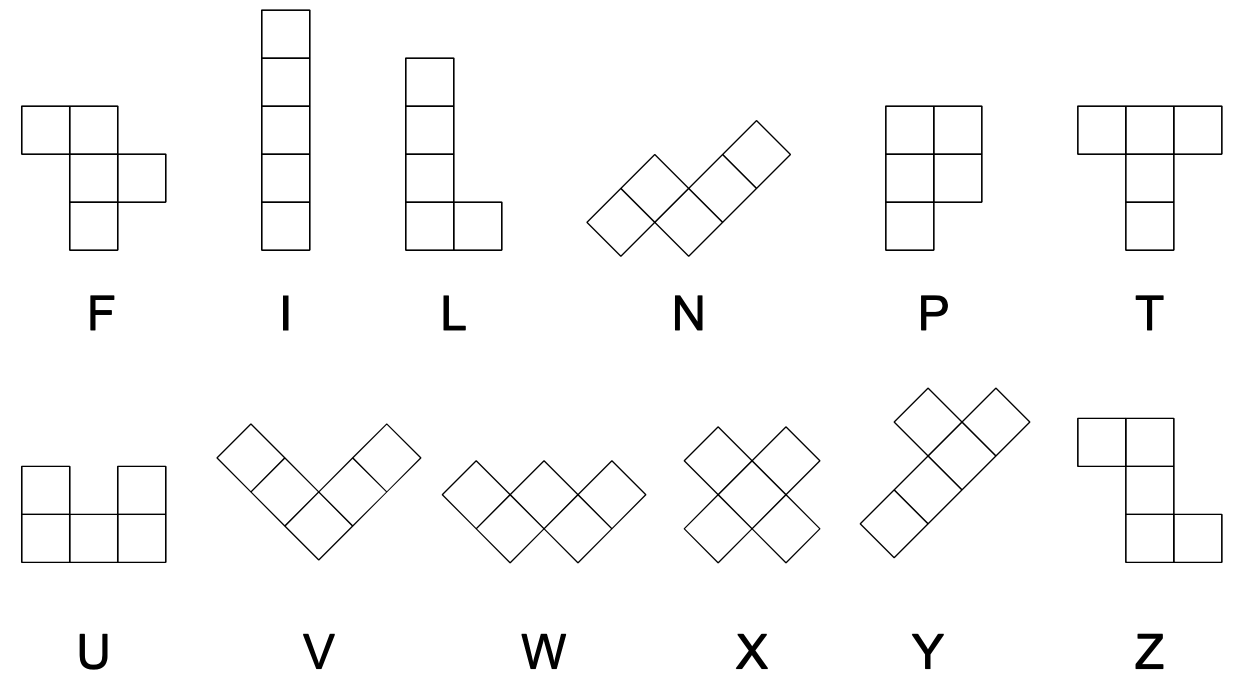


Figure 1: The 12 pentominoes and the letters designating their names.

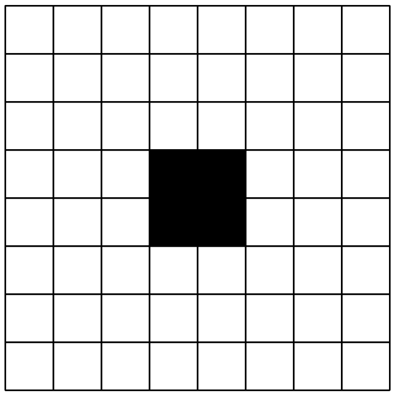
2 Tiling a torus with pentominoes

Snailspeed, Ltd., sells game and puzzle applications for mobile devices. One such puzzle involves ***pentominoes***. A pentomino, the name of which was coined by Solomon W. Golomb in 1953 and popularized by Martin Gardner in his “Mathematical Games” column in *Scientific American*, is a figure made of 5 connected squares, as shown in Figure 1. It is conventional to identify each pentomino by a capital letter, as shown in the figure. The classic ***pentomino puzzle*** involves tiling an 8-by-8 board with these 12 pentominoes, using each pentomino exactly once. Pentominoes can be rotated and flipped, but they cannot overlap. Since 12 pentominoes only cover 60 out of the 64 squares of an 8-by-8 board, the board starts with 4 arbitrarily filled squares. Figure 2(a) shows a solution to a pentomino puzzle in which the four middle squares are filled. For more background information on pentominoes and pentomino puzzles, consult `http://en.wikipedia.org/wiki/Pentomino`.

A ***pentomino torus puzzle*** is similar to a pentomino puzzle, except that pieces “wrap around” the edges of each row and column. Figure 2(b) shows a solution to a pentomino torus puzzle which has no solution as an ordinary pentomino puzzle. Harvey has implemented a program to solve pentomino torus puzzles. It takes as input the coordinates of the 4 initially filled squares, and it prints out either the number of solutions or the board configuration of every possible solution, depending on the arguments you pass. Harvey’s solution uses a recursive backtracking-search algorithm. You can read more about his implementation in his commented code.

You can also play with `http://godel.hws.edu/java/` for an interactive demonstration of a pentomino puzzle solver. Sadly, Harvey’s program provides a much less interactive interface. You can specify the locations of the four initially filled squares using a command like

```
./pentominoes -b "7,1 7,2 6,3 7,6"
```


(a)

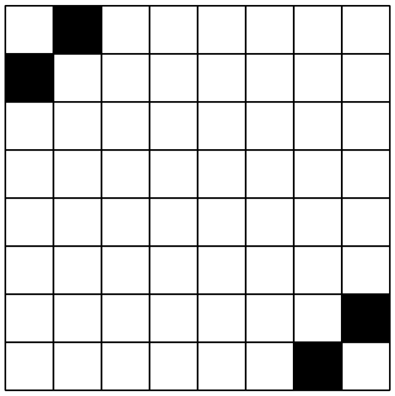

(b)

Figure 2: (a) A solution to a pentomino puzzle. (b) A solution to a pentomino torus puzzle.

Then, the program prints out the 8-by-8 board showing where the four squares are located:

```
--------
........
........
........
........
........
........
...#....
.##...#.
--------
```

The program then measures how long it takes to find a solution:

```
---- RESULTS ----
pieces filled: 0002097152, solutions found: 00000000
Solutions found: 1
---- END RESULTS ----
Elapsed execution time: 1.488306 sec
```

To print out the solution board, add `-p` to the command. You may find for simpler boards, it takes a trivially short amount of time to find the first solution. You can pass `-a` to find all solutions to the board. This option has the opposite problem: it might take *days* for Harvey's program to solve certain board configurations. Until you improve the program's performance, a good middle ground might be to find 1000 solutions and then exit. You can pass the option `-n 1000` to do so.

2.1 Optimize the pentomino torus puzzle

Improve the performance of Harvey's program. Describe the optimizations you make, and how they impact performance. In particular, investigate how operations on boards might be improved by using a board representation that supports bit hacks. Then optimize the implementations of `fill_piece()` and `can_fill_piece()`.

Your solution will be considered correct if it finds the same solutions and the same number of solutions as Harvey's. You can list solutions in any order, as long as you list them all. Do not, however, change the output format of the solution. The code in `main.c` calls your `print_board()` function. Make sure its output is identical to Harvey's.