UNIT III

EMBEDDED PROGRAMMING

SOFTWARE PERFORMANCE OPTIMIZATION

ANALYSIS AND OPTIMIZATION OF EXECUTION TIME, POWER, ENERGY, PROGRAM SIZE.

SOFTWARE PERFORMANCE OPTIMIZATION

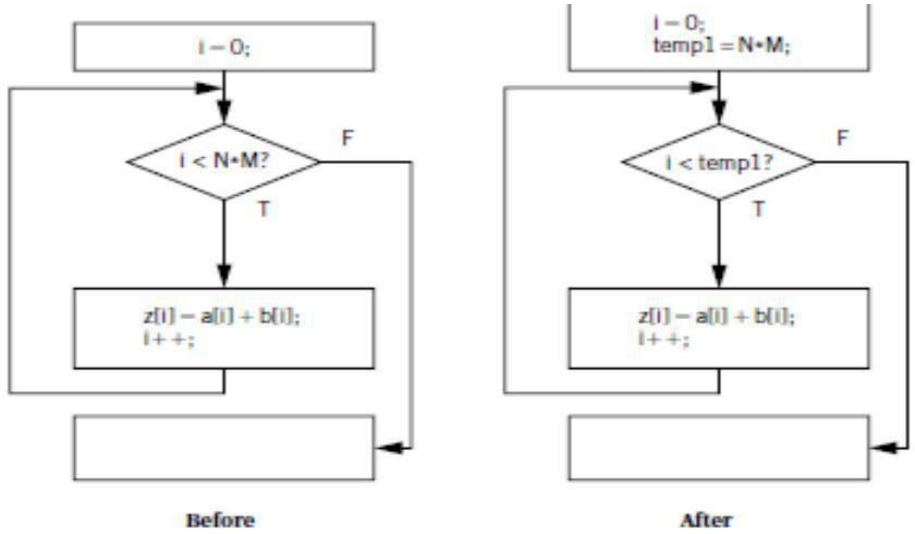
Loop Optimizations:

Loops are important targets for optimization because programs with loops tend to spend a lot of time executing those loops. There are three important techniques in optimizing loops: **code motion, induction variable elimination,** and **strength reduction**. Code motion lets us move unnecessary code out of a loop. If a computation's result does not depend on operations performed in the loop body, then we can safely move it out of the loop. Code motion opportunities can arise because programmers may find some computations clearer and more concise when put in the loop body, even though they are not strictly dependent on the loop iterations. A simple example of code motion is also common.

```
for (i = 0; i < N*M; i++) {
    z[i] = a[i] + b[i];
    }
```

- The code motion opportunity becomes more obvious when we draw the loop's CDFG .The loop bound computation is performed on every iteration during the loop test, even though the result never changes. We can avoid N *_M _*1 unnecessary executions of this statement by moving it before the loop, as shown in the figure.
- An **induction variable** is a variable whose value is derived from the loop iteration variable's value. The compiler often introduces induction variables to help it implement the loop. Properly transformed, we may be able to eliminate some variables and apply strength reduction to others. A nested loop is a good example of the use of induction variables. Here is a simple nested loop

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        z[i][j] = b[i][j];
```



Before After

Cache Optimizations

- A **loop nest** is a set of loops, one inside the other. Loop nests occur when we process arrays. A large body of techniques has been developed for optimizing loop nests. Rewriting a loop nest changes the order in which array elements are accessed. This can expose new parallelism opportunities that can be exploited by later stages of the compiler, and it can also improve cache performance

ANALYSIS AND OPTIMIZATION OF EXECUTION TIME, POWER, ENERGY, PROGRAM SIZE.

- The memory footprint of a program is determined by the size of its data and instructions. Both must be considered to minimize program size.
- Data provide an excellent opportunity for minimizing size because the data are most highly dependent on programming style. Because inefficient programs often keep several copies of data, identifying and eliminating duplications can lead to significant memory savings usually with little performance penalty.
- Buffers should be sized carefully rather than defining a data array to a large size that the program will never attain, determine the actual maximum amount of data held in the buffer and allocate the array accordingly. Data can sometimes be packed, such as by storing several flags in a single word and extracting them by using bit-level operations.

- A very low-level technique for minimizing data is to reuse values. For instance, if several constants happen to have the same value, they can be mapped to the same location. Data buffers can often be reused at several different points in the program. This technique must be used with extreme caution, however, since subsequent versions of the program may not use the same values for the constants.

- A more generally applicable technique is to generate data on the fly rather than store it. Of course, the code required to generate the data takes up space in the program, but when complex data structures are involved there may be some net space savings from using code to generate data.
- Minimizing the size of the instruction text of a program requires a mix of high-level program transformations and careful instruction selection.
- Encapsulating functions in subroutines can reduce program size when done carefully. Because subroutines have overhead for parameter passing that is not obvious from the high-level language code, there is a minimum-size function body for which a subroutine makes sense.

- Architectures that have variable-size instruction lengths are particularly good candidates for careful coding to minimize program size, which may require assembly language coding of key program segments. There may also be cases in which one or a sequence of instructions is much smaller than alternative implementations for example, a multiply-accumulate instruction may be both smaller and faster than separate arithmetic operations.
- When reducing the number of instructions in a program, one important technique is the proper use of subroutines. If the program performs identical operations repeatedly, these operations are natural candidates for subroutines.
- Even if the operations vary somewhat, you may be able to construct a properly parameterized subroutine that saves space. Of course, when considering the code size savings, the subroutine.

Thank you