Code Optimization and Scheduling Loop Parallelization

Code optimization

Code optimization is any method of code modification to improve code quality and efficiency. A program may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer input/output operations.

The basic requirements optimization methods should comply with, is that an optimized program must have the same output and side effects as its non-optimized version. This requirement, however, may be ignored in the case that the benefit from optimization, is estimated to be more important than probable consequences of a change in the program behavior.

Types and Levels of optimization

Optimization can be performed by automatic optimizers, or programmers. An optimizer is either a specialized software tool or a built-in unit of a compiler (the so-called optimizing compiler). Modern processors can also optimize the execution order of code instructions.

Optimizations are classified into:-

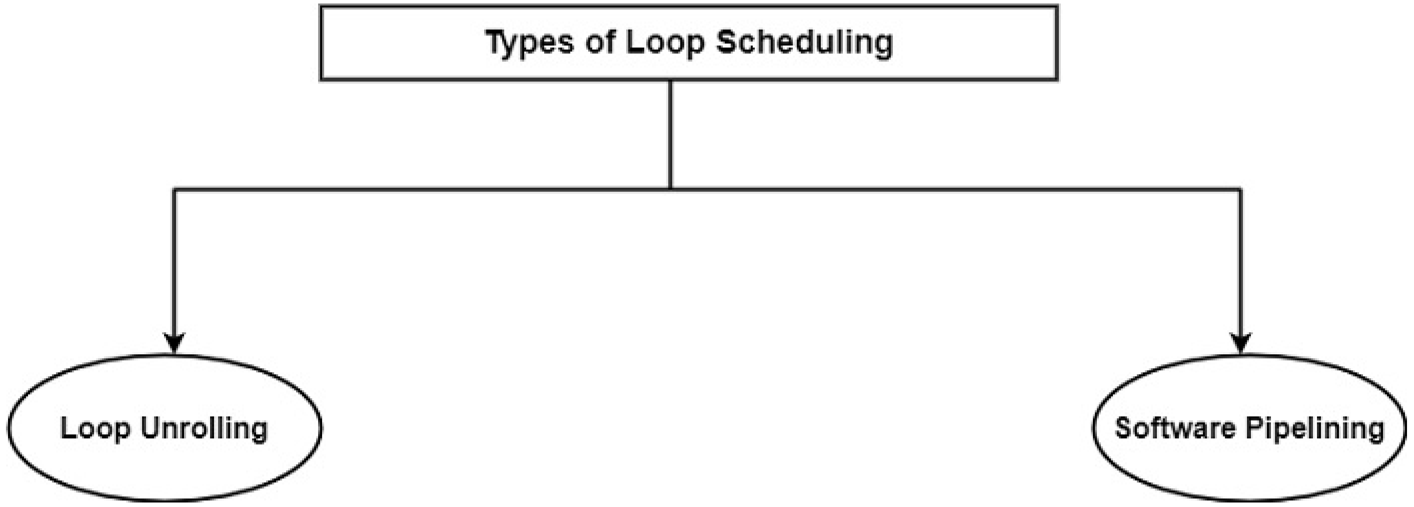
- High-level and
- Low-level optimizations.

High-level optimizations are usually performed by the programmer, who handles abstract entities (functions, procedures, classes, etc.) and keeps in mind the general framework of the task to optimize the design of a system. Optimizations performed at the level of elementary structural blocks of source code - loops, branches, etc. - are usually referred to as high-level optimizations too, while some authors classify them into a separate ("middle") level (N. Wirth?).

Low-level optimizations are performed at the stage when source code is compiled into a set of machine instructions, and it is at this stage that automated optimization is usually employed. Assembler programmers believe however, that no machine, however perfect, can do this better than a skilled programmer (yet everybody agrees that a poor programmer will do much worse than a computer).

Loop Scheduling

Loops are an important source of parallelism for **Instruction-level Parallelism** (ILP) processors. Therefore, the regularity of the control structure can speed up computation. Loop scheduling is a central point of instruction schedulers that have been advanced for highly parallel ILP-processors, including **Very long instruction word** (VLIWs).



Loop unrolling

The basic concept of loop unrolling is to repeat the loop body multiple times and to discard unnecessary inter-iteration code, including decrementing the loop count, verification for loop end, and branching back conditionally between iterations.

This will result in a shortened implementation time. Loop unrolling can be executed simply when the multiple iterations are already established at compile-time, which appears generally in ‘do’ and ‘for’ loops.

Loop unrolling stores performance time, at the cost of code length, in much the similar method as code inlining or traditional macro expansion. Code inlining is one of the standard compiler optimization approaches, used for short, occasionally used subroutines.

Software pipelining

Software pipelining is that consecutive loop iterations are implemented as if they were a hardware pipeline as displayed in the following table. Let us see cycles c+4, c+5, and c+6. These are the ones displaying the real advantage of software pipelining. The important point is that for these cycles the available parallelism between subsequent loop iterations is fully used. For instance, in cycle c+4 the parallel operations are as follows −

- It can be storing the result of iteration 1 (that is, a(1)), auto-incrementing the index.
- It can be decrementing the loop count, which is maintained in r200 by 1.
- It can be performing the floating-point multiplication with the operands belonging to cycle 4, that is (2.0 * b(4));
- It can be loading the operand for iteration 5 that is b(5).