19MCE304- DESIGN OF EMBEDDED SYSTEMS

BASIC COMPILATION TECHNIQUES:

It is useful to understand how a high-level language program is translated into instructions. Since implementing an embedded computing system often requires controlling the instruction sequences used to handle interrupts, placement of data and instructions in memory, and so forth, understanding how the compiler works can help you know when you cannot rely on the compiler.

Next, because many applications are also performance sensitive, understanding how code is generated can help you meet your performance goals, either by writing high-level code that gets compiled into the instructions you want or by recognizing when you must write your own assembly code.

The compilation process is summarized in Figure 2.19. Compilation begins with high-level language code such as C and generally produces assembly code. (Directly producing object code simply duplicates the functions of an assembler which is a very desirable stand-alone program to have.)

The high-level language program is parsed to break it into statements and expressions. In addition, a symbol table is generated, which includes all the named objects in the program. Some compilers may then perform higher-level optimizations that can be viewed as modifying the high-level language program input without reference to instructions.

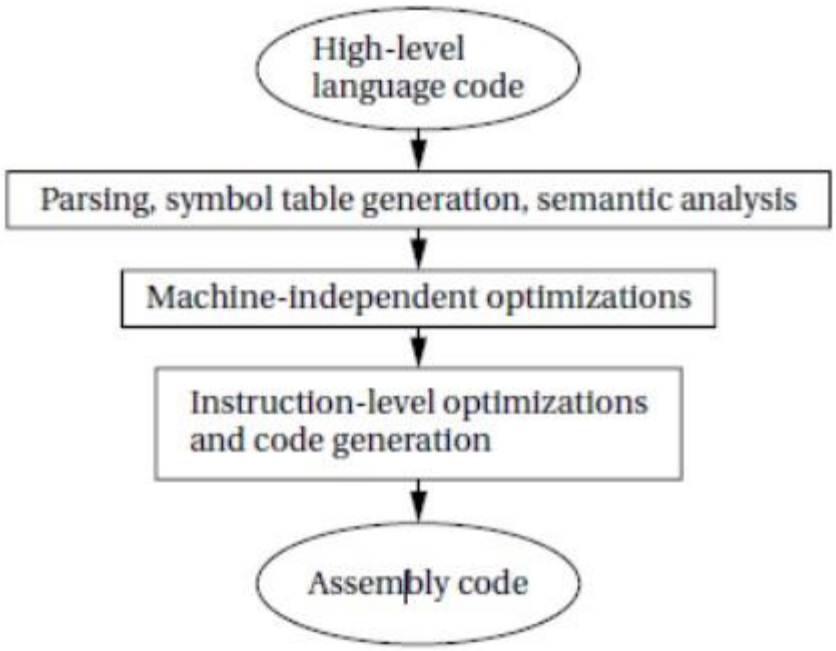

Fig 2.19 The compilation process.

Simplifying arithmetic expressions is one example of a machine-independent optimization. Not all compilers do such optimizations, and compilers can vary widely regarding which combinations of machine-independent optimizations they do perform.

Instruction-level optimizations are aimed at generating code. They may work directly on real instructions or on a pseudo-instruction format that is later mapped onto the instructions of the target CPU. This level of optimization also helps modularize the compiler by allowing code generation to create simpler code that is later optimized. For example, consider the following array access code:

x[i] = c*x[i];

A simple code generator would generate the address for x[i] twice, once for each appearance in the statement. The later optimization phases can recognize this as an example of common expressions that need not be duplicated. While in this simple case it would be possible to create a code generator that never generated the redundant expression, taking into account every such optimization at code generation time is very difficult. We get better code and more reliable compilers by generating simple code first and then optimizing it.

Program Optimization Techniques

Program optimization focuses on improving the efficiency and performance of executable programs after compilation. Key techniques include:

1. **Algorithmic Optimization**:
    - **Purpose**: Improves the efficiency of algorithms used in the program.
    - **Examples**: Using more efficient sorting algorithms (e.g., quicksort instead of bubblesort), reducing the complexity of operations.
2. **Data Structures Optimization**:
    - **Purpose**: Selecting appropriate data structures to enhance data access and manipulation efficiency.
    - **Examples**: Using hash tables for fast lookups, arrays for direct access, and balanced trees for efficient searching.
3. **Loop Optimization**:
    - **Purpose**: Improves the efficiency of loops by reducing iteration overhead.
    - **Examples**: Loop unrolling, loop fusion, loop-invariant code motion.
4. **Memory Optimization**:
    - **Purpose**: Reduces memory usage and improves memory access patterns.
    - **Examples**: Data alignment, memory pooling, minimizing memory leaks.
5. **Parallelization and Concurrency**:
    - **Purpose**: Utilizes multiple CPU cores or threads to execute tasks concurrently, improving performance.
    - **Examples**: Multithreading, multiprocessing, vectorization (SIMD - Single Instruction Multiple Data).
6. **Compiler-Specific Optimizations**:
    - **Purpose**: Takes advantage of compiler features and optimizations tailored to specific programming languages or platforms.
    - **Examples**: Compiler flags (e.g., -O3 for GCC), platform-specific optimizations.
7. **Profiling and Analysis**:
    - **Purpose**: Identifies performance bottlenecks and areas for optimization by analyzing program execution.
    - **Tools**: Profilers, performance counters, and debuggers.

Example of Optimization Techniques in Action

Consider optimizing a sorting algorithm in C++:

cpp
Copy code

```
#include <algorithm>
#include <vector>
#include <iostream>

int main() {
    std::vector<int> nums = {5, 2, 8, 1, 6};

    // Sort using the standard library's quicksort
    std::sort(nums.begin(), nums.end());

    // Output sorted numbers
    for (int num : nums) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

- **Algorithmic Optimization**: Choosing the appropriate sorting algorithm (e.g., quicksort) based on data characteristics.
- **Data Structures Optimization**: Using efficient data structures (e.g., vector) for storage and access.
- **Compiler Optimization**: Enabling compiler optimizations (-O2 or -O3 flags for GCC/Clang) to generate optimized machine code.