Code Scheduling for Instruction-level Parallel Processors

The Hardware-Software Interface

Tópico 5

Instruction-level Parallel Processing

Tópico 2 : Exploiting Instruction-level Parallelism (ILP) – An Overview.

Tópico 3 : Instruction-level Parallel Processors How architectures exploit ILP.

Tópico 4 : Processing Control Transfer Instructions.

Tópico 5 : *Code Scheduling for Instruction-level Parallel Processors.*

Code Scheduling for ILP Processors

- Introduction
- Basic block scheduling
  - List scheduling
- Loop scheduling
  - Loop unrolling
  - Software pipelining
- Global scheduling
  - Trace scheduling and Finite Resource Global scheduling

Introduction

Smart code scheduling can boost the performance of ILP-processors significantly, since they perform, at most, dependency checking and resolution, but do not carry code optimisation for parallel execution.

The detecting and resolution of dependencies can be achieved either statically or dynamically, or in a concerted way both statically and dynamically.

In the case of static scheduling, the compiler has to deliver dependency-free and parallel optimised code (early pipelined processors and VLIWs). In contrast, under dynamic scheduling the ILP compiler behaves as a performance booster (superscalar processors).

Introduction$_{(cont)}$

Performance-greedy ILP processors expect parallel optimisation from the compiler as well. During parallel optimisation, the compiler identifies independent instructions and reorders code such that independent instructions become executable as early as possible. Thus hardware resources are better utilised and program execution is speeded up. Evidently parallel optimisation can be carried out only if dependencies among instructions have been previously identified. Thus, dependency detection and resolution is a prerequisite for parallel optimisation.

The term *code scheduling* is used to cover dependency detection and resolution as well as parallel optimisation.

Traditional Compilers

- Traditional non-optimising compilers typically consist of two major parts:
  - The front-end performs scanning, parsing and semantic analysis of the source program and produces an intermediate representation.
  - The back-end, in turn, generates the object code.
- Traditional optimising compilers speed up sequential execution and reduce the required memory space mainly by eliminating redundant operations.

ILP-Compilers

ILP-compilers integrate traditional compilation and code scheduling in one of two ways:

- *Pre-pass scheduling* completely integrates scheduling into the compilation process, following the traditional sequential optimiser.

- The other approach is to use a traditional (sequentially) optimising compiler and carry out code scheduling afterwards (*post-pass scheduling*).

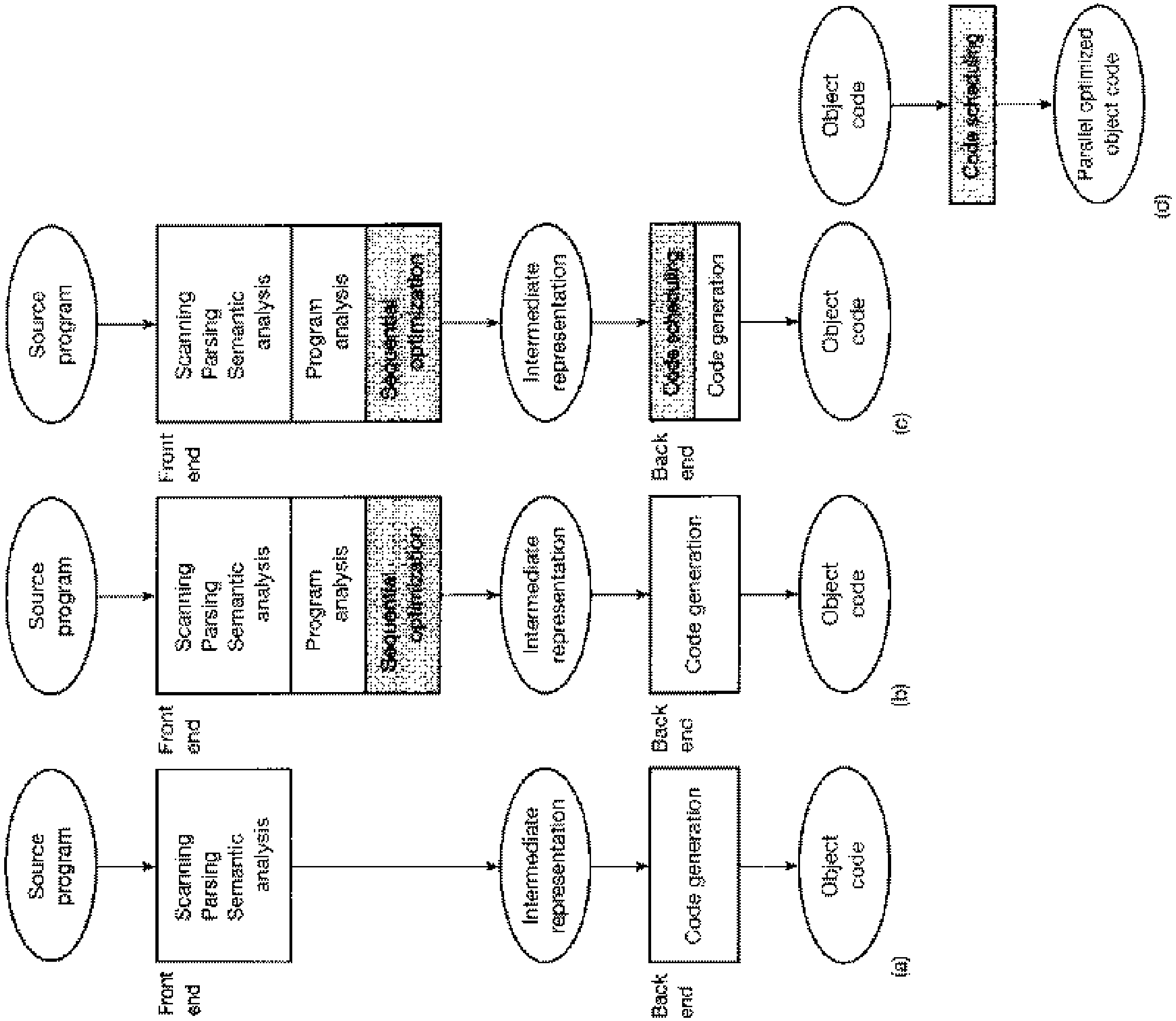

Figure 1 Typical layout of (a) a traditional non-optimizing compiler, (b) a traditional optimizing compiler, (c) an ILP-compiler performing pre-pass parallel optimization, and (d) an ILP-compiler with post-pass parallel optimization.

Traditional and ILP-compilers

There are two significant differences between traditional compilers (TCs) and ILP-compilers (ICs):

- During sequential execution, no WAR or WAW dependency can occur, therefore TCs do not pay attention to false dependencies during register allocation.
  - TCs and ICs have contradictory criteria with respect to register allocation. TCs try to reuse registers as much as possible to reduce the number required. ICs attempt to avoid register reuse if it results in a false dependency.
- TCs do not handle potential data dependencies associated with memory references.

Code Scheduling for ILP Processors

Code scheduling may be performed at three different levels:

1. The easiest way to code schedule is to do it at *basic block* level, separately for each block, one after the other.

2. At the *loop* level consecutive iterations of a loop can usually be overlapped, resulting in considerable speed-up.

3. The most effective way to schedule is to do it at the highest possible level using *global* scheduling techniques.