Purple Dragon book - Chapter 10.3 & 10.4 (Basic Block Scheduling & Global Code Scheduling)

Abstract

Jan Görig, xgorig01@stud.fit.vutbr.cz

Tomáš Ocelík, xoceli00@stud.fit.vutbr.cz

Many modern high-performance processors can execute more instructions per machine cycle. Such processors have more computing units of some kind - for example, multiple *Arithmetic logic units*. The aim of modern compilers for such architectures is the maximum utilization of available parallel computing units. In that case it is necessary to keep data and control dependencies between instructions. Otherwise, execution of a program would lead to unexpected results.

First, the article describes data dependencies between instructions and their representation by a *data-dependence graph*. Besides showing dependencies between operations, the graph shows the delays between them. For example, delay two means that the second instruction can be done no earlier than 2 cycles after the first instruction, on which is *data-dependent*.

Second part of the article deals with the *basic blocks* scheduling. The algorithm works with a dependence-graph introduced in the previous section. The simplest approach is to visit each node of the graph according to *prioritized topological order*. Since the graph can have no cycles, there is always at least one topological order for the nodes. However, among the possible topological orders, some may be more preferable than others.

The following section describes the *List-scheduling algorithm*. The algorithm visits each node of the data-dependence graph according to the selected order. For each node it computes the earliest time the node can be executed regarding to data dependencies of the previously scheduled nodes. Then, the resources needed by the node are checked against a *resource-reservation table*. The node is scheduled at the earliest time the resources are available.

Next chapter deals with the actual selection of a suitable prioritized topological order for the List-scheduling. The algorithm visits each node only once (does not backtrack), so *heuristic priority function* is used to choose among the nodes that are ready to be scheduled next. If there are no constraints on resources, the shortest schedule is given by the longest path through the dependence graph. On the other hand, if all the operations are data-independent, scheduling is constrained only by available resources. Then, the most critical operations may be given the highest priority. The third option is to first schedule operations which appears in the source program earlier.

Scheduling within the basic blocks can leave a lot of resources unused. Therefore following section deals with the *global scheduling* to make use of machine resources better. Instructions can be moved from one basic block to another. Now besides data dependencies, we also must take *control dependencies* into account. It is necessary to ensure that all instructions in the original program must be executed in the optimized program and while the optimized program may execute extra instructions speculatively, these instructions must not have any unwanted side effects. The concept of *dominance relation* is introduced. We say that a block B *dominates* block B' if every path from the entry of the graph to B' goes through B. Similarly, block B *post-dominates* block B' if every path from B' to the exit of the graph goes through B. If B dominates B' and also B' post-dominates B, we say that B and B' are *control-equivalent*. Based on this relation, situations that may arise and their solutions are presented. For example, sometimes it is necessary to insert an instruction being processed to more branches as *compensation code* if any of that branches can be executed. Then it may happen that one branch will run slower than before optimization. So optimization will improve program execution only if the optimized paths are executed more frequently than the slower ones.

Another part of the article deals with the global scheduling algorithms themselves. Generally, over 90% of a program's execution time is spent on less than 10% of the code. Thus, we should aim to make the most frequent executed paths run faster, while possibly making the least frequent paths run slower. The most frequent paths can be found either by estimation - for example, the inner loops are performed more frequently than those outside – or another option is to use profiling with a representative sample of data to measure which paths are executed the most frequently.

The *Region-based* scheduling algorithm is introduced. It moves operations up to control-equivalent block and speculative moves operations up one branch to a dominating predecessor.

Another technique is *Loop Unrolling*. In the Region-based scheduling, operations from one loop iteration cannot overlap with the operations in the next iteration. The principle is to unroll the loop small number of times before code scheduling.

The last described algorithm called *Neighborhood Compaction* uses also techniques with compensation code. Region-based scheduling is followed by a simple *pass*. In this pass, we can examine each pair of basic blocks that are executed one after one and eventually move some operations up or down to improve performance. Finally, thesis introduces some advanced techniques and describes *dynamic scheduling*.