Summary of the paper: 
Register Allocation with Instruction Scheduling: a New Approach.

INTRODUCTION
The goal of this paper is to combine register allocation and instruction scheduling in such a way that during register allocation no possibilities to parallelize is removed, i.e., no false dependencies are introduced. This can be useful in e.g. a superscalar machine.

What to do it with
The algorithm begins with the output from the code selection with symbolic registers (Example in Fig. 2)

Again we see here the output from the code selection, again with symbolic registers, s1-s9

In this example we also will assume a few possible machine dependencies:
• There are only one arithmetic unit handling floating-point and …
• … one handling fixed-point.
• Only one fetching unit.

How to do it
First we will find a graph representing relations between instructions that's NOT possible to schedule in parallel.
• Create the schedule graph (Fig. 3)
• Produce the transitive closure of the schedule graph.

We can realize that the edges in this graph is relations between instruction that is not possible to execute in parallel. After this is made we have no further interest in the directions of edges and therefore removes them. (Fig 4a)

To complete the work finding NOT parallelizable instructions we add the machine dependencies to this graph. We can see that s3, s4 both needs the fixed-point arithmetic unit and therefore cannot execute in parallel. The fetch instructions s1, s2, s6 and s7 all have machine dependence according to our model. This results in a graph like Fig 4b. So, this is the graph representing all relations between instructions that's NOT possible to schedule in parallel.

Next thing to realize is that any dependence introduced by the register allocator that is NOT a part of this graph is a false dependence. Thus, we create a "false dependence graph" by, 'inverting' the graph we just produced. (Fig. 5)

This papers claims that:
IF we keep the symbolic registers defined in the two instructions connected to a false dependence in different registers we avoid introducing the false dependence.
To take advantage of this fact we look upon the false dependence in the same way as the interferencies in the interference graph.

The interference graph is shown in Figure 6. As I said, false dependencies can be looked upon as interferences. Therefore we introduce edge from false dependence graph into the interference graph. The result is called the "parallelizable interference graph" (Fig. 7).

This graph presents all relations where different symbolic registers need to use different registers. Of course this graph is now used for COLORING! The coloring of the parallelizable interference graph provides a register allocation where no false dependencies are created, which let the scheduling work as parallel as possible.

Sadly this algorithm of course might need more registers. If fewer registers are available than there are colors needed to color the graph, we need to sacrifice something. This is done by removing edges in the parallel interference graph. The big question is which edges to remove. The paper suggests a few suggestions:

- Avoid removing edges resulting from both the interference graph AND the false dependence graph. This might result in both spilling AND lost parallelization.
- First restrict parallelism, i.e. offer edges from false dependence graph first.

Another approach is to create a heuristic variable spilling, in short, to deciding which variable to spill create

\[ h(v) = \frac{\text{cost}(v)}{\sum w(\{u,v\})} \]

where cost is a measure on an instruction's nesting level and \( w \) is the weight of the edges from/to \( v \).

**Summary of summary**

The important parts of this paper are:

- Creation of a parallel interference graph.
- How to use it in register allocation to get a register allocation with no false dependencies introduced.
- Actions to take to when registers are too few.

The paper also gives approach for how to use similar technique for scheduling between basic blocks.