Journal of Programming Language **4** (1996) 21–38

Register allocation with instruction scheduling: a new approach

SHLOMIT S. PINTER

Department of Electrical Engineering, Technion — Israel Institute of Technology, Haifa 32000, Israel

Received September 20 1995

A new framework is presented in which considerations of both register allocation and instruction scheduling can be applied uniformly and simultaneously. In this framework an optimal colouring of a graph, called the *parallelizable interference graph*, provides an optimal register allocation and preserves the property that no false dependences are introduced, thus all the options for parallelism are kept for the scheduler to handle. For this framework heuristics are provided for trading off parallel scheduling with register spilling.

Keywords: register allocation, instruction-level parallelism, instruction scheduling

1. Introduction

Register allocation is an important task of every compiler. The problem of efficiently allocating values to registers was investigated extensively for sequential processors. The most widely used solution to the problem is by colouring the interference graph [1] which represents conflicts between live ranges of values; the chromatic number of the graph equals the number of registers needed and a colouring specifies an allocation. Values that cannot be accommodated in registers are spilled into memory. Many heuristics for this global allocation problem (which in general is NP-complete) have been investigated and implemented over the years.

Recently, with the development of new processors which offer instruction level parallelism, an optimal colouring of the interference graph does not necessarily correlate with good machine utilization. To take advantage of the parallel capabilities, instruction reordering is carried out by the instruction scheduler. When instruction reordering is carried out after register allocation the selection of registers may limit the possibilities to reorder instructions due to false dependences that are introduced with the reuse of registers. On the other hand, when reordering instructions precede register allocation the span of live values is increased, implying longer register lifetimes. Thus, more registers are needed and more spills may be introduced. This register pressure is increased further with long pipelines and the use of loop unrolling techniques. In addition, in some cases register allocation must precede instruction scheduling since the exact register assignment is needed by the scheduler [2].

The above tradeoff is manifested in the following short example. Program (b) of Example 1 is a representation of the code fragment generated for (a) with symbolic registers. In (c) a possible

allocation of values to physical registers is provided where `r1` and `r2` are reused. In this case only three registers are needed but a false dependence is introduced between the second and fourth instructions forbidding the parallel execution (scheduling) of the two instructions.[1]

```
Example 1:
                    s1 := load z          r1 := load z
                    s2 := i               r2 := i
x := a[i]           s3 := a[s2]           r3 := a[r2]
y := z + z          s4 := s1 + s1         r2 := r1 + r1
z := x * 5 + z      s5 := s3 * 5 + s1     r1 := r3 * 5 + r1
  (a)                   (b)                   (c)
```

In view of the tradeoff between good register allocation and good instruction scheduling in some compilers, like those for the MIPS processors, register allocation precedes instruction scheduling [3], whereas in others, like the one for the IBM RISC S/6000 [4], instruction scheduling is carried out first.

Register allocators and instruction schedulers each use a graph model of the input program. Since the meanings of the nodes and edges in those graphs are different and partially conflict (as we shall see), a simple combination of the graphs is impossible. In this work we provide a simple common framework for representing the input program for both tasks. In this framework a register allocation using the minimum number of registers (without spilling) does not decrease the parallelism embedded in the program. We also investigate the case when spilling occurs.

The relation between code scheduling and register allocation has been considered in a few studies. A method for alternating between two scheduling techniques in which one tries to schedule instructions so as to minimize the number of registers required whereas the other schedules for pipelined processors is presented in [5]. Another solution in [5] modifies the dependence graph whenever a register must be reused (by adding the related dependences considering all possible registers as candidates), and chooses one register. In [6] three strategies are studied. In the first, a limit is put on the number of registers to be used in every basic block; in the second, some scheduling cost estimates are employed by the register allocator; and in the last method a pre-stage is used for estimating the limits to be used in the first method. Their basic algorithm maintains both graphs and alternates between running the restricted versions of both the allocator and the scheduler. In [7] some registers are coloured during the optimization phase and the final colouring procedure is called from the scheduler.

Our approach is different from all of the above in that it provides a single framework in which optimal utilization can be achieved when there are enough registers and both heuristics of the scheduler and the allocator may be applied when the number of registers is small. In particular, the *parallelizable interference* graph which takes into account scheduling constraints is generated first. With this graph register allocation is carried out; we prove that an optimal colouring of this graph provides a register allocation which does not generate false dependences. In the case of a need for spill, heuristics for both scheduling and register allocation are performed on the graph at

[1] There is a way to allocate three registers and not generate the false dependence between the second and the fourth instructions (using the mapping `s1-r1, s2-r2, s3-r2, s4-r3, s5-r2`); in this case instructions 2 and 4 can be executed simultaneously.

the same time. The scheduling itself takes place after the register allocation; nevertheless, the relative order of the non-constrained (independent) statements need not be the one used during the register allocation process. We present some new heuristics and show how to implement existing ones in this framework. The effect of the scheme is presented on the SPEC92 benchmark.

We first provide the necessary background and introduce some definitions in Section 2. Then, in Section 3 we present our framework and prove its properties. Section 4 includes our algorithms (heuristics) for the case when the number of registers is too small and spills may be needed. Conclusions and experimental results are presented in Section 5.

2. Definitions

In this section we define the general setting for both the register allocation and the instruction scheduling problems. A simple model of a superscalar machine is provided, followed by a symbolic register level based description of a program. These models are used for defining our problem and algorithms. Although the results are presented for a superscalar machine model, they are all applicable to regular single-issue, pipelined uniprocessors as well.

2.1 *Machine model*

In the description of the model we assume a RISC type processor (memory reference instructions are only `load` and `store` while computations are done in registers) with finitely many registers. An instruction level parallel processor is a RISC type processor comprising a collection of functional units, each one of them can potentially execute one instruction in the same machine cycle. Examples of such processors are the MIPS R3000 and the IBM RISC S/6000 comprising three functional units: fixed point, floating point and branch units.

The source code of the program is first translated into instruction level register based intermediate code where an infinite number of symbolic registers is assumed (one symbolic register per value)[2] [18], or one can use static single assignment form [9]. In what follows we use data dependences between instructions of the intermediate code. We define the *data dependences* between instructions as follows:

(1) Let u and v be two instructions. A data dependence from u to v exists if one of the following holds:
   (a) *Data flow dependence*—the register (value) defined in u is used in v.
   (b) *Data anti-dependence*—a register used in u is later reassigned (redefined) in v (destroying the value used in u).
   (c) *Data output dependence*—the register defined in u is redefined in v (destroying the value defined earlier in u).

[2] During this translation some obvious machine related transformations deviate from this assumption by reusing registers (e.g. incrementing induction variables).

2.2 *Instruction scheduling for superscalar machines*

Instruction scheduling is the process of moving instructions so that they can be scheduled to the different units of the processor such that the total execution time is minimized. Instructions are moved in a way that preserves the program's semantics on one hand, and provides a better utilization of the machine on the other hand.

The constraints to the scheduler are presented by a *scheduling graph*, $G_s = (V_s, E_s)$, constructed as follows:

(1) Every node $v \in V_s$ corresponds to an instruction of the intermediate code.
(2) There exists a directed edge $(u, v) \in E_s$, from u to v, if u must be executed before v. This happens in one of the following three cases: (i) there is a data dependence from u to v (i.e. v is data dependent on u), (ii) there is a control dependence from u to v, (iii) there is a machine constraint that enforces the precedence of u over v.

Control dependences exist only between basic blocks where the corresponding edges are derived from the program's flow graph (following `if` branches etc.). The machine-based constraints imposed on the program may forbid the use of the same functional unit within a short time interval (resource contention), or the simultaneous access to the same memory address, or similar constraints. In general, machine constraints that are not of a precedence type (e.g. forbidding the simultaneous use of resources) are not present in the graph; we consider those constraints in a later stage.

A legal scheduling must preserve the execution order described by the edges of the graph. Thus, adding precedence edges to G_s may restrict the potential for parallel scheduling. Traditional instruction scheduling is done per basic block. The precedence constraints in such a case are the data dependence based constraints and the machine based constraints.

2.3 *Register allocation*

Register allocation is the process of mapping symbolic registers into physical ones in a way that values stay in registers as long as they are live (if at all possible). For this mapping an *interference graph*, $G_r = (V_r, E_r)$, representing the interactions between live values, is constructed as follows:

(1) Every node $v \in V_r$ represents a live range of a symbolic register. A live range is the union of the regions in the program in which the symbolic register is live. Typically it is a connected component that begins at definition (assignment) points and terminates at last uses.[3]
(2) There exits an (undirected) edge $\{u, v\} \in E_r$ if and only if live ranges u and v interfere (intersect) with each other; that is, a definition of one range occurs at a point where the other range is live (its value is used or subsequently used).

In practice, in many compilers an end point of the live range of a symbolic register (i.e. the statement corresponding to its last use) is not considered part of the range; this, later on, enables the reuse (redefinition) of a register in the same statement that last uses it (e.g. the increment of a register).

[3] We further discuss this issue when we use the interference graph.

A colouring of a graph is *legal* if no two nodes that are incident upon the same edge have the same colour and it is *optimal* if it uses the smallest colouring set over all legal colourings. An optimal colouring of the interference graph induces an *optimal register allocation* (if the number of registers in the machine is not smaller than the chromatic number of the graph) in which every value is in a register as long as it is alive. In addition to the fact that the minimum colouring problem is NP-complete, in general the number of registers is small. Thus, in practice a *spilling* stage is carried out in which the values of some symbolic registers are temporarily stored in memory. The register allocation problem is to find a register mapping in which the cost of spilling is minimum.

As discussed in the introduction, register allocation may restrict the capabilities to schedule instructions in parallel for superscalar machines; the issue raised is whether the capabilities to schedule instructions in parallel can be defined together with register allocation so that one does not restrict the other.

3. Register allocation for superscalar machines

As seen from the definitions in the previous section, the vertices in the two graphs do not have the same meaning, thus, a simple combination of the graphs is impossible. In this section we provide a simple common framework for representing the two frameworks.

We start with an example that demonstrates some of the main ideas of our technique. Consider an execution of the code in Example 2 on a processor with two arithmetic units (fixed-point and floating-point), where the `si` variables represent symbolic registers. The program and its dependence edges of the scheduling graph are presented in Fig. 1.

At this point when we have all the precedence type constraints we generate the transitive closure of the scheduling graph and remove the directions of the edges. To this new graph we now add all the machine related constraints that are not of a precedence type. For example, in our machine there is only one fixed-point unit, thus operations `s3` and `s4` cannot be executed simultaneously; this constraint will be represented by adding the edge `{s3, s4}`. In some scheduling algorithms such an edge is not present in the graph but instead the algorithm itself will take it into account by trying to schedule an operation on the floating-point unit following an operation on the fixed-point unit and vice versa (see for example [10]). In our framework we will make all such constraints

```
Example 2:
    s1 := load z (fixed)
    s2 := load y (fixed)
    s3 := s1 + s2
    s4 := s1 * s2
    s5 := s3 + s4
    s6 := load x (float)
    s7 := load w (float)
    s8 := s6 * s7
    s9 := s5 + s8
```

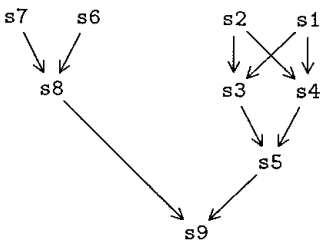


Fig. 1. A program and its dependence edges of the scheduling graph

explicit at this stage. We note that the more edges are present in this graph the better the results will be; this is so since we are actually going to use only the edges that are in the complement of the constructed graph. The edges in the complement graph present the actual parallelism available to our machine for the given program. To continue our example, since we have only one fetching unit we will also generate all the possible edges between the four load instructions `s1, s2, s6`, and `s7`. The only edges now in the complement graph of the example are between `s8` and each of the five statements `s1, s2, s3, s4, s5`, all the edges between `s7` and each of `s3, s4, s5`, and all those between `s6` and each of `s3, s4, s5`. Indeed, the paired nodes present the only operations that can be executed in parallel according to the definition of our machine. The next stage of our technique is to integrate those edges with the interference graph so that the register allocation algorithm will take this available parallelism into account. We will continue the example after the formal presentation of our framework.

The formal presentation starts with the definition of false dependences whose existence limits the potential parallelism implemented by the scheduler. When scheduling is done after register allocation the dependences between the registers provide the edges of the scheduling graph. The register allocation may cause the generation of dependences that were not present before.

We first define an augmentation of a scheduling graph (a directed graph) to include undirected edges that correspond to non-precedence type of constraints.

Definition 1 Let G_s be a scheduling graph. The *augmentation* of G_s is generated by adding to G_s all non-precedence based machine constraints as undirected edges.

Definition 2 Let G_s and G'_s be the scheduling graphs generated before and after register allocation, respectively. An edge $\{u, v\}$ or a directed edge (u, v) in the augmented G'_s is a *false dependence* if there is a potential to schedule u and v simultaneously prior to register allocation (i.e. $\{u, v\}$ is not in the augmented G_s and no directed path from either u to v or v to u exists in G_s).

The joint optimization problem is defined as follows:

Definition 3 A register allocation for a superscalar machine is *optimal* if it requires the smallest possible number of registers such that no two live ranges that co-exist at some point are allocated to the same register and no false dependence is introduced (in the augmented scheduling graph generated following the register allocation).

In general, the number of registers in a machine is smaller than the number of registers needed for the optimal solution. Since most found solutions may need even more registers the allocator has to introduce and trade off spilling and false dependences.

We next derive a new graph comprising exactly all the false dependences of a given program (that may be generated following the run of any legal register allocator). The edges of the graph will be used in the generation of the parallelizable interference graph whose optimal colouring produces an optimal register allocation for superscalar machines. The construction is first provided for a basic block and is generalized to a full program in Section 3.1. We point out that if only a local instruction scheduler is used by the compiler then there is no need to generate the global construction.

For a given basic block define the *false dependence* undirected graph, $G_f = (V_f, E_f)$, as follows:

let $G_s = (V_s, E_s)$ be the scheduling graph of the basic block prior to register allocation, i.e. instructions are presented with symbolic registers and all the precedence based constraints are present in the graph.

(1) $V_f = V_s$.
(2) Consider the set of edges in the transitive closure of G_s and define E_t to be this set after the removal of the directions of the edges. Add to E_t all the non-precedence based constraints that describe the restrictions on the machine capabilities[4]. The set of edges E_f comprises the pairs $\{u, v\}$ such that $u, v \in V_f$, $u \neq v$, and $\{u, v\} \notin E_t$.

It is clear that the scheduling graph, G_s, of the basic block has no output and anti data dependence edges (with symbolic registers no register is redefined during an execution of the basic block); thus, the set E_t contains exactly the real constraints on the scheduler. In the case that the false dependence graph is constructed after partial register allocation (some registers are reused), it is assumed that every anti or output dependence thus introduced is part of a single machine instruction (e.g. incrementing a register). Therefore, no real output or anti dependence is introduced.

Lemma 1 Let G_s and G'_s be the augmented scheduling graphs generated before and after register allocation, respectively. A dependence edge between u and v in G'_s is a false dependence iff $\{u, v\} \in E_f$.

Proof. From the construction we know that E_t contains exactly all the pairs (of instructions) which may not be scheduled simultaneously. Since E_f contains the pairs in the complement of E_t, then $\{u, v\} \in E_f$ implies that u and v may be scheduled simultaneously with respect to G_s. Thus, by definition, the dependence (constraint) edge between u and v in G'_s is a false dependence edge. For the other direction: if $\{u, v\} \notin E_f$ then $\{u, v\} \in E_t$ thus u and v may not be scheduled simultaneously with respect to G_s, implying that the edge between u and v in G'_s is not a false dependence edge. ■

Since anti and output dependences are generated only when a memory location (or a register) is redefined they are often considered as non-real dependences. Indeed, when an unbounded number of symbolic registers is used, no redefinition is needed and these dependences need not distract parallelism. This leads to the idea that if we keep in different registers the two definitions corresponding to a false dependence edge, then no false output dependence will be introduced (similarly for non-precedence type of constraints); we will also show that our technique does not generate false anti dependences.

In Fig. 2(a) we present only the data dependence edges of the scheduling graph, G_s, corresponding to Example 1(b). The instructions (vertices) are marked with the defined symbolic registers. The graph in Fig. 2(b) presents the edges in the set E_t; note that the edges `{s1,s3}` and `{s4,s5}` are machine-dependent constraints. Lastly, Fig. 2(c) shows the interference graph, G_r, of the same example.

[4] The constraints modelled at this stage are of the form: these two instructions may not be scheduled in parallel. Positive forms of non-precedence constraints can be introduced on the complemented graph which is generated next (e.g. by removing edges or nodes).

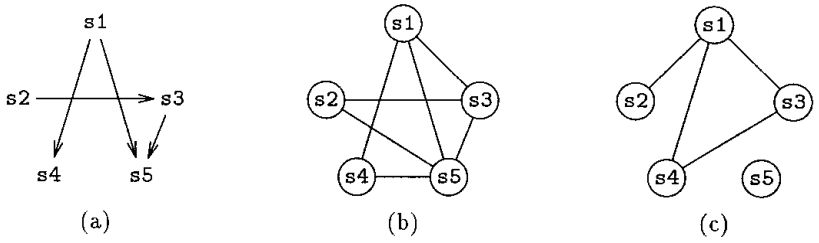


Fig. 2. A scheduling graph (a), the edges in the set E_f (b), and the interference graph (c) of Example 1(b)

From Fig. 2(b) we see that the only false dependence edges are `{s1,s2}`, `{s2,s4}` and `{s3,s4}`; indeed, the operations in every pair of a false dependence edge can be scheduled on the same cycle (according to our example).

With the false dependence graph we next define the parallelizable interference graph. It basically includes both edges of the false dependence graph and those of the interference graph. The following observation and claim are used in the construction:

In the interference graph of a basic block a definition (assignment of a value) marks the beginning of a live range (*live interval* in this case) and also potentially defines a new symbolic register; thus, a node may correspond to the defining instruction and to the symbolic register.

Claim 1 Let G_s and G_r be the scheduling and interference graphs respectively of a basic block. Then $V_r \subseteq V_s$.

Proof. Every definition corresponds to a single instruction. ■

Note that only branch instructions are not directly represented in the interference graph. This can be relaxed somewhat since the predicate of the branch condition is computed separately and thus is present in the interference graph. A call instruction is changed to be a multiple register assignment, a load defines a new value and a store instruction indicates the end of a live interval (it may also be considered a spill). Also, the right number of names analysis and coalescing, see [1], is used to discover the live ranges and unify those that share the same use. Within a basic block this may have an effect only in some redefinitions in the body of a loop; it is simple to see that Claim 1 is preserved in this case. The modifications needed for generating the global interference graph are discussed in Section 3.1.

Note that in general there may be edges in E_s such that their undirected counterparts are not in E_r and vice versa. For example, an interference edge in E_r which is present due to the sequential ordering of the input code may not be present in E_s (live ranges are approximated with def-use chains that are computed for some sequential ordering of the code); for the other direction: machine-based constraints may not be present in E_r, or a data flow dependence edge $(u, v) \in E_s$ may not be present in E_r since it corresponds to the last use of u (which is not considered in the live range of the symbolic register).

At this point we are ready to construct the parallelizable interference graph. In such a process we need all the edges in E_r even though some of them only present interferences due to the

sequentiality of the input code (but not due to real dependences or machine constraints). Thus, our solution (register allocation) is relative to the order of instructions in the input code. Note that the only need to preserve sequentiality arises from the way in which live ranges are computed by current register allocators. In contrast, the constraints used for generating the false graph include only the data dependences in the program and those of the machine. In some compilers pre-scheduling is used to somewhat improve the input ordering.

Define the *parallelizable interference graph*, $G = (V, E)$, of a basic block (the definition will later be extended to more complex control structures) as follows: let G_r be the interference graph of the block and G_f its false dependence graph.

(1) $V = V_r$.
(2) $E = E_r \cup \{\{u, v\} : \{u, v\} \in E_f \text{ and } u, v \in V\}$.

In Fig. 3 we see the parallelizable interference graph of Example 1 and a possible register allocation.

Note that the parallelizable interference graph may be expanded to include extra nodes, like stores (see Claim 1), and extra edges in order to represent more information for the instruction scheduler. Nevertheless, those parts of the expanded graph do not take part in the colouring (register allocation) algorithm.

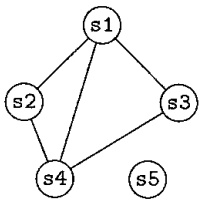
A possible *expanded parallelizable interference graph* (for a basic block) is:

(1) $V = V_s$.
(2) $E = E_r \cup \{\{u, v\} : \{u, v\} \in E_f \text{ and } u, v \in V\}$.

The nodes in this graph were changed and an edge between two nodes means that the two operations may be scheduled at the same cycle or the two nodes represent live ranges that are not disjoint. Thus, for each node v all the nodes paired with v comprise a list of available instructions (with v) similar to the one used in list scheduling algorithms such as in [11].

Theorem 1 Every optimal colouring of the parallelizable interference graph provides an optimal register allocation for the given machine.

Proof. Given a program fragment; let G_f, G_r, G and G'_s be the false dependence graph, the interference graph, the parallelizable interference graph, and the augmented scheduling graph built with the colours (of G) specifying physical registers, respectively. The colouring of the nodes adjacent to the edges in $E_r \subseteq E$ ensures that live ranges that co-exist at some point are allocated to different registers. From Lemma 1, a dependence edge between u and v in G'_s is false if and only if $\{u, v\}$ belongs to E_f (subset of E). From the colouring of G no two vertices incident upon an edge in E_f




```
r1 := load z
r2 := i
r2 := a[r2]
r3 := r1 + r1
r2 := r2 * 5 + r1
```

Fig. 3. The parallelizable interference graph of Example 1 and a possible register allocation

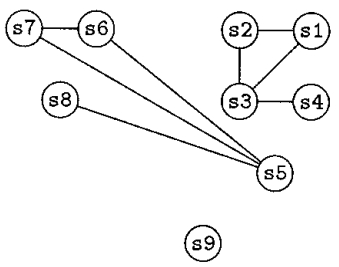
are allocated to the same register. This ensures that no false output dependence nor false non-precedence type constraint is generated. The edges of E_r ensure that no false flow dependences appear. To complete the proof, we next prove that no false anti dependence is generated. Let $\{u, v\} \subseteq V$ be such that v has the same colour (register) as one of the values used in u and the definition of v kills that value (using the colours to specify registers). We have to show that (u, v) in G_s' is not a false anti dependence edge. Let x be the value (register) used in u and killed by v and let node x, that precedes u, represent the definition of that value. Clearly, from the construction of the interference graph G_r, v is not in the live interval of the definition of x. Thus, the colour (register) of x may be reused in v. Now u and v can potentially be scheduled simultaneously since the fetch in u will precede the generation of the value in v. The rest of the proof is derived directly from the construction of the parallelizable interference graph and Lemma 1. ■

Going back to Example 2, if we assume that no value is live on the entrance and exit from the code fragment then from the interference graph in Fig. 4(a) we see that only three registers are needed; but there is no restriction to assign the same register, for example, to operations `s8` and `s3` or to operations `s8` and `s2`, thus preventing the possible parallel scheduling of these operations. Such an assignment is impossible with the parallelizable interference graph; remember that the complement graph generated following the transitive closure and the addition of non-precedence machine constraints includes all the edges between `s8` and each of `s1, s2, s3, s4, s5`, and thus they are in the parallelizable interference graph. With the parallelizable interference graph four registers are needed. One of the possible assignments is provided in Fig. 4(b).

The next theorem proves the optimality of our construction with respect to the input graph and the machine in use; i.e. it shows the necessity of all the edges in the general case.

Theorem 2 Let $G = (V, E)$ be a parallelizable interference graph of a basic block and $G' = (V, E')$ where $E' = E - \{u, v\}$ for some edge $\{u, v\} \in E$. Every colouring C, such that $C(u) = C(v)$, provides for the basic block a register allocation in which a spill is introduced or the resulting scheduling graph has a false dependence.

Proof. If $\{u, v\} \in E_r \subseteq E$ then a spill is introduced. Otherwise $\{u, v\} \in E_f$ thus from Lemma 1



(a)

```
Example 2:
   s1   r1 := load z (fixed)
   s2   r2 := load y (fixed)
   s3   r3 := r1 + r2
   s4   r2 := r1 * r2
   s5   r3 := r3 + r2
   s6   r1 := load x (float)
   s7   r4 := load w (float)
   s8   r4 := r1 * r4
   s9   r1 := r3 + r4
```

(b)

Fig. 4. (a) The interference graph of Example 2 and (b) a possible register assignment

u and v could be scheduled simultaneously according to the scheduling graph for the code when presented with symbolic registers. ■

The consequence of Theorems 1 and 2 is that optimal colouring of the parallelizable interference graph is an optimal register allocation for superscalar machines with respect to the order of the instructions used for defining the live ranges and their interference. It provides the assignment of a minimum number of registers that are needed in order not to generate a spill for a live value and it keeps all the parallelization options for the scheduler (no false dependences in the resulting graph). The case in which spills happen, i.e. when the number of registers is small, is presented in Section 4. Since the initial possible parallelism is prevalent and only part of it may be implemented (the scheduler can exploit even a smaller part) one can think that the extra edges may only spoil the register allocation. As we shall see in Section 4 this is not the case. During the register allocation process (especially when the potential need for spilling arises) it is possible to remove some of the extra edges (giving up some option for parallelism) by reasoning simultaneously on the benefit of parallelism versus the possibility of a spill.

3.1 *Generalizing the parallelizable interference graph*

We next expand our framework to cover scheduling across basic block boundaries. In such scheduling, control dependence edges between basic blocks are used. In global code motion or percolation based scheduling [12, 13] the instructions are moved across branches and join points (inter-basic blocks); this is also the case in region scheduling based techniques [14] and in [15]. For these last two scheduling techniques moving instructions are possible only within a *region* which is a maximal acyclic fragment of code. The scheduling is done by logically ignoring the control dependence edges between two basic blocks that are considered as a single block for scheduling. There are a few criteria for selecting such two blocks and we say that the selected blocks are *candidates for simultaneous scheduling*. An example of such a case is when one block is executed if and only if the other one is also executed. This condition holds when one block dominates the other and the second one postdominates the first, and can be verified by observing the dominators tree [8] and the postdominators tree (constructed like a dominators tree when the edges in the program flow graph are reversed).

In the example of Fig. 5, blocks `b1` and `b4` are candidates for simultaneous scheduling. In order to keep open this prospect for parallelism during register allocation, the control edges are removed before the generation of the transitive closure (in the process of building the false dependence graph).

Some of the restrictions on global instruction schedulers concern instructions (or even blocks) that may never be scheduled together. Such is the case for the instructions in blocks `b2` and `b3` since they may never execute together. In order to take care of such scheduling constraints we add undirected edges between the nodes representing the statements in the two blocks. This is done at the same stage in which we add edges for representing the non-precedence based constraints on the machine capabilities. The result is that the parallelizable interference graph will not have edges between instructions in `b2` and `b3`. This is consistent with registers needs. For example, if each of `b2` and `b3` will have a statement that defines `x` then we need both definitions in the same register.

Let $G_s = (V_s, E_s)$ be the global scheduling graph of a region prior to register allocation (each

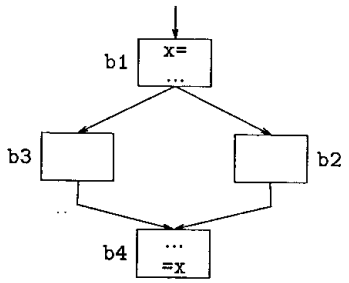


Fig. 5. Blocks b1 and b4 are candidates for simultaneous scheduling

node in V_s corresponds to a single instruction). The *global false graph*, $G_f = (V_f, E_f)$, is defined as follows:

(1) $V_f = V_s$.
(2) Remove all the control edges from G_s.
(3) Generate the transitive closure of G_s and remove the directions of the edges; denote the new set of edges by E_t.
(4) Add to E_t all the non-precedence based constraints that describe the restrictions on the machine capabilities. Add to E_t edges between instructions in every pair of blocks that are not candidates for simultaneous scheduling.[5]
(5) The set of edges E_f comprises the pairs $\{u, v\}$ such that $u, v \in V_f$, $u \neq v$, and $\{u, v\} \notin E_t$.

Note that optimizations for removing unreachable code prior to the generation of the global false graph improve G_f.

When generating the global interference graph, live ranges are no longer intervals. A definition may be live on two branches of a conditional statement (a possibility for multiple end points to the live range), or the value of a single use may depend on more than one definition (i.e. several def-use chains reach a single use; e.g. when coming from different branches of an if-then-else statement). See, for example, Fig. 6.

Although the combination of def-use chains into a single live range generates a node in the interference graph whose edge degree is higher than any of its constituents (each def-use chain), such a combination is desirable since we need the value in a single register and it has no negative implications on the scheduler; observe that such a combination does not decrease the options for parallelism since operations on two branches of a conditional may never execute simultaneously and in the other cases (anti or output) data dependence prevent parallel executions.

At this point our early observation that every live range corresponds to a definition (assignment statement) is no longer true. Nevertheless, we may view a node, v, in the global interference graph, G_r, as representing all the definitions of its live range. We later use the notation $v_i \in v$ where v_i is a node in the global scheduling graph, G_s, which is one of the definitions of the live range v. Figure 6

[5] If the analysis that reveals fragments of code that are candidates for simultaneous scheduling is done on instructions level (rather than blocks) then the extra edges to add at this stage will relate to instructions as well.

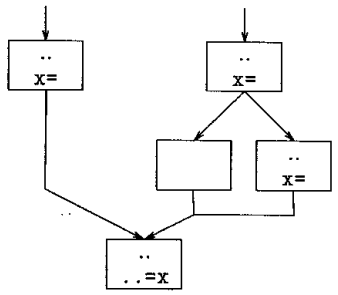


Fig. 6. The live range for x at the use point has three definitions

has three definitions for the live range of x. Formally we use $v_i \in v$ iff $v_i \in V_s$ is a definition statement of the value in the live range of $v \in V_r$.

For inter basic block scheduling (on regions) and global register allocation the *global parallelizable interference graph*, $G = (V, E)$, is constructed as follows:

(1) $V = V_r$ which is the set of vertices in the global interference graph.
(2) $E = E_r \cup \{\{u, v\} : \{u_i, v_j\} \in E_f \ , \ u_i \in u \ , \ v_j \in v$ and $u, v \in V_r\}$
where E_r is the set of edges in the global interference graph.

Claim 2 Let G_s and G_r be the global scheduling and global interference graphs respectively of a program segment. Let $v_i \neq v_j \in V_s$, $v_i, v_j \in v \in V_r$; then the corresponding instructions of v_i and v_j may never execute in parallel.

With Claim 2, Theorems 1 and 2 can be shown to be true for this generalization to regions, thus enabling the use of the technique also with region (or global) based schedulers. In general, there are three possibilities to combine global and local false or interference graphs; the choice depends on the types of the instruction scheduler and the register allocator in use.

With some caution one can also expand the global parallelizable interference graph similarly to the one for basic blocks (i.e. replacing V_r by V_s).

4. More on algorithms

From Theorem 1 we know that a minimum size colouring set of the parallelizable interference graph provides a legal register allocation, and from Theorem 2 we know that such a colouring set is the smallest one that will not generate false dependences and still keep live values in registers as long as needed. Since the number of registers may be smaller than the size of the minimum colouring set, or the set achieved, heuristics that are being used during either register allocation or scheduling may be applied on the parallelizable interference graph. We next present some heuristics and those properties of the graph that they use. The heuristics are embedded in a Chaitin based colouring algorithm but may be integrated in any register allocator.

The first type of heuristic eliminates edges from the graph. The question of which edge to eliminate may involve consideration of both the scheduler and the allocator. We note that all

the possible alternatives for parallelization are kept; since only some alternatives can be used (e.g. with two pipelines only two instructions may be scheduled at the same cycle), some alternatives can be removed. This involves scheduling considerations. We can also choose to preserve some edges that promise good parallelization and decide to remove interference edges which may lead to spills (a spill may be avoided only when two such nodes get different colours in spite of the absence of the interfering edge).

The following rules are used:

(1) Remove those edges of $E - E_r$ for which the parallel scheduling of the two instructions have the smallest priority for the scheduler. Such a priority may involve early scheduling of an instruction which is last on a critical path of the precedence edges (in E_s).
(2) Avoid the removal of edges in $E_f \cap E_r \subseteq E$. They are used by both the scheduler and the allocator.

Lemma 2 Allocating a single register for definitions u and v where $\{u, v\} \in E - E_r$ may not cause a spill but removes an option for parallelism.

Lemma 3 Allocating two registers for definitions u and v where $\{u, v\} \in E_f \cap E_r$ both may prevent a spill and enables the parallelization of the two definitions.

Another possible approach is to generalize the heuristic function used in many register allocators. It is customary to use the following spilling guidance function h for selecting which definition to spill:

$$h(v) = cost(v)/deg(v)$$

where $cost(v)$ is the cost of keeping the value v in a register rather than in memory, and $deg(v)$ is the degree of v in the interference graph. The live range v with the lowest $h(v)$ value is spilled first.

This function might also be improved by considering weights on the edges of the parallelizable interference graph. The weight function $w : E \rightarrow \mathcal{R}$ can be generated by using Lemmas 2, and 3 that distinguish between those edges that preserve parallelism and those that prevent spills. Each such type of an edge has its 'price' for any given machine.

(1) Let $N(v)$ denote the set of vertices that each share an edge with v. The new h guidance function can be:
$$h^*(v) = cost(v)/\sum_{u \in N(v)} w(\{u, v\})$$
(2) The *cost* function, in general, is a function of the instruction's nesting level and may not be changed.

In particular, if all the edges in $E - E_r$ have weight 0 and the rest are 1 then we get the traditional h function. Since for generating the live ranges for the interference graph a sequential ordering of the instructions is used, we add a preliminary scheduling heuristic for selecting one such order. The scheduling graph $G_s = (V_s, E_s)$ is first extended by adding to every node $v \in V_s$ a number $EP(v)$ representing the earliest possible time for scheduling v (in [16] *EP* stands for early partition). The *EP* numbers are computed from the scheduling graph; during this stage the delay numbers (on the edges) that are part of the scheduling graph (they were ignored in our presentation of G_s) may be used for generating more accurate *EP* numbers; see for example [17,18].

4.1 *A registers allocation algorithm*

Given a program segment represented with symbolic registers, the scheduling graph, G_s, is first generated. The mapping $EP : V \rightarrow N$ of the earliest possible time for scheduling is computed.

Scheduling heuristics—Traverse the scheduling graph by increasing *EP* numbers (smallest first). Whenever all the operations with the same *EP* number cannot be scheduled together (machine limitations) select the operations to be postponed; for each node in the postponed set increase the *EP* number and update the *EP* numbers on all the paths (in G_s) leaving the node. When this process terminates select a linear order which is consistent with the partial order of the new *EP* numbers and reorder the program segment accordingly.

Colouring procedure—Generate the parallelizable interference $G = (V, E)$ of the code fragment. Denote by r the number of registers in the machine and initialize a spill list to be `empty`.

$G' := G$
`while` G is not empty `do` ;;; simplify
  `while` there is a node $v \in V$ with degree smaller than r `do`
    delete v and update G
  `end` do
  `while` there is a node in V with degree smaller than r when only interference
    edges are considered `do`
      use scheduling considerations to remove from both G and G' a false dependence
      edge not in E_r (e.g. an edge $\{v, u\}$ for which scheduling u with v contributes
      the least), thus giving up some possible parallelization (this can be done
      globally or with respect to a selected node)[6];
      `while` there is a node $v \in V$ with degree smaller than r `do`
        delete v and update G
    `end` do
  end do
  `if` V is not empty `then` {
    choose $v \in V$ with min $h^*(v) = cost(v)/\sum_{u \in N(v)} w(\{u, v\})$;
    delete v and update G;
    place v on the spill list
    }
.`end` do
`if` the spill list is empty `then` ;;; select
  colour the nodes in reverse order of deletion (this is done by rebuilding G a node at
  a time using G')
`else` { ;;; spill
  spill each v in the spill list;
  repeat the colouring procedure
  }

The second while loop (in the simplify part) guarantees that the convergence property of the

[6] Currently, it is also advisable to use the knowledge of homogeneous multiple units if there are more than two of some type.

algorithm will be similar to the one proved for the original algorithm. It is pointed out that since we keep all the possible parallelism up to that stage, for most of the instructions there will be many scheduling possibilities, thus scheduling pruning is a must during the simplify stage.

5. Discussion and experimental results

We embedded part of our scheme in a gcc-based compiler for the Pentium processor. The Pentium has two non-symmetric pipelines, called U and V, that share some of the functional units. As a result there exists a collection of pairing rules that limits the simultaneous use of the two pipes. For example, instructions with an immediate operand and displacement operand cannot be paired. Following the pairing rules the instructions were divided into four sets: G1 includes all those instructions that must use the U pipe with no other instruction in the V pipe; this set is the largest of all four. G2 contains the 27 instructions that can execute in each of the two pipes. G3 includes all the instructions (27) that must use the U pipe, but at the same time the V pipe may be used as well (e.g. V can be used by instruction from G2). The last set, G4, contains all the 39 instructions that are used for changing control, and each of them can be executed as a pair only in the V pipe. With this classification we generated the proper edges in the false dependence graph.

The global register allocation in the gcc compiler uses a priority scheme which is based on the length of the live ranges of values, the number of uses of a value, and the number of registers needed for a value. In our modification we did not replace the allocator but rather checked if this priority scheme can take advantage of the parallelizable interference graph. This was done by introducing the number of neighbours in the interference (conflict) graph (for the base compiler), and the number of neighbours in the parallelizable interference graph (for the modified compiler), to the priority scheme when sorting the symbolic registers. In addition, we remove the local register allocator from the two compilers.

We use the compilers on the SPEC92 benchmark; the results are summarized in Table 1. Even with the very limited changes to the compiler (as described above) and without the use of the colouring algorithm or any of the new heuristics, the relative improvement (values in the original column minus those of the modified column divided by the last one) is nice for arithmetically oriented programs. This can be explained by the extra available parallelism in sets G2 and G3 (which include many of the arithmetic instructions). In the 022.li and 026.compress programs the differences were very small. In addition, the spills in the Pentium are relatively cheap. Therefore, a processor for which spills are more expensive might benefit more from the scheme, especially with a compiler that will embed more of the scheme (compared to the above experiment).

Table 1. Timing results for the two compilers

Program	*Original (user seconds)*	*Modified (user seconds)*	*Relative improvement*
052.alvinn	59.14	50.78	0.16
050.ear	28.35	27.05	0.05
023.eqntott	0.96	0.72	0.33
072.sc	0.54	0.48	0.13

The desire to allocate a small number of registers to be used at any point during the execution of a program is driven both by the limited number of registers and by the fact that temporary saving of registers' values, for example, when a procedure is called, is time consuming. With many functional units competing for registers (even more so with deep pipelining) and with new optimization techniques like unrolling, careful allocation of registers is still a central issue for current technology.

5.1 *Future work*

Encouraged by the preliminary results we plan to investigate further heuristics for removing edges and selecting values to spill. We plan to embed our technique in other types of register allocators and extend it to handle more than two homogeneous units. Finally, we would like to use the framework to measure machine dependent available parallelism.

Acknowledgements

The author would like to thank David Bernstein, Itai Nahshon and Nathan Srebro for fruitful discussions. Special thanks are due to Jacob Levin, Adi Greenberg, Ilan Hazan and Yuval Hamuz for their work on the implementation. The author would also like to thank the anonymous referees for their insightful reviews. This work was partially supported by a grant from Intel Israel Ltd. Early results appeared in [19].

References

1. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markst. Register allocation and spilling via graph coloring, *Computer Languages*, **6** (1981) 47–57.
2. J. Hennessy and T. Gross. Postpass code optimization of pipline constraints, *ACM Transactions on Programming Languages and Systems*, **5** (1983) 422–448.
3. F. Chow. M. Himelstein, E. Killiam and L. Weber. Engineering a RISC compiler system. In *COMPCON Proceedings*. IEEE, March 1986.
4. K. O'Brien, B. Hay, J. Minish, H. Schaffer, B. Schloss, A. Shepherd and M. Zaleski. Advanced compiler technology for the RISC System/6000 architecture. In *IBM RISC System/6000 Technology*, pp. 154–161. IBM SA23–2619, 1990.
5. J. R. Goodman and W. Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, pp. 442–452. ACM, July 1988.
6. D. G. Bradlee, S. J. Eggers and R. R. Henry. Integrated register allocation and instruction scheduling for RISCs. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, April 1991.
7. R. F. Touzeau. A Fortran compiler for the FPS-164 scientific computer. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Vol. 19, pp. 48–57, June 1984.
8. A. V. Aho, R. Sethi and J. D. Ullman. *Compilers—Principles, Techniques, and Tools* (Addison-Wesley, Reading, MA, 1986).

9. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman and F.K. Zadeck. An efficient method of computing static single assignment form. In *Sixteenth Annual Symposium on Principles of Programming Languages*, pp. 25–35. ACM, January 1989.
10. H. S. Warren. Instruction scheduling for the IBM RISC system/6000 processor, *IBM Journal Research and Development*, **34**, (1990). pp. 85–92.
11. P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Vol. 21, pp. 11–16, July 1986.
12. J. A. Fisher. Trace scheduling: A technique for global microcode compaction, *ACM Transactions on Computer Systems*, **C-30** (1981) 478–490.
13. A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *International Conference on Parallel Processing*, pp. 614–618. ACM and IEEE Computer Society, August 1985.
14. R. Gupta and M. L. Soffa. Region scheduling: an approach for detecting and redistributing parallelism, *IEEE Transactions on Software Engineering*, **16** (1990) 421–431.
15. D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 241–255. ACM, June 1991.
16. S. Davidson, D. Landskov, B. D. Shriver and P. W. Mallett. Some experiments in local microcode compaction for horizontal machines, *IEEE Transactions on Computers*, **C-30** (1981) 460–477.
17. D. Bernstein and I. Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle, *ACM Transactions on Programming Languages and Systems*, **11**(1) (1989) 57–66.
18. K. V. Palem and B. Simons. Scheduling time-critical instructions on risc machines, *ACM Transactions on Programming Languages and Systems*, **15** (1993) 632–658.
19. S. S. Pinter. Register allocation with instruction scheduling: a new approach. In *SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 248–257, ACM, June 1993.