Dependence-Conscious Global Register Allocation

Wolfgang Ambrosch M. Anton Ertl Felix Beer Andreas Krall

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
{anton,fbeer,andi}@mips.complang.tuwien.ac.at
Tel.: (+43-1) 58801 {4459,3036,4462}

Abstract. Register allocation and instruction scheduling are antagonistic optimizations: Whichever is applied first, it will impede the other. To solve this problem, we propose dependence-conscious colouring, a register allocation method that takes the dependence graph used by the instruction scheduler into consideration. Dependence-conscious colouring consists of two parts: First, the interference graph is built by analysing the dependence graphs, resulting in fewer interference edges and less spilling than the conventional preordering approach. Second, during colouring the register selection keeps dependence paths short, ensuring good scheduling. Dependence-conscious colouring reduces the number of interference edges by 7%–24% and antidependences by 46%–100%.

1 Introduction

Global register allocation and instruction scheduling are two standard compiler techniques. Register allocation reduces the traffic between the processor and memory by keeping frequently-used variables in registers. Instruction scheduling reduces the number of pipeline stalls (wait cycles) by reordering instructions.

However, these techniques are antagonistic: Instruction scheduling tends to move dependent instructions apart. This lengthens the lifetimes of the values and increases register pressure, which in turn may cause more memory traffic. On the other hand, the register allocator can assign the same register to two different temporary values. This can reduce the opportunities for reordering instructions and it can increase the number of pipeline stalls. So, the technique that is applied first will reduce the effectivity of the other technique. This problem is especially important for pipelined and superscalar implementations of register-starved architectures, e.g., the Pentium and the 68060.

As an example, consider Figure 1: Conventional register allocation before scheduling can introduce dependences that cause bad scheduling. The last instruction stalls for two cycles waiting for the result of the multiply. Scheduling before allocation uses more than the four available registers (the other registers hold global values) and causes spilling to memory.

To solve this dilemma, Goodman and Hsu developed DAG-driven register allocation, a local register allocation algorithm that avoids introducing additional dependences if possible [GH88]. Inspired by DAG-driven register allocation, we created dependence-conscious colouring, a global register allocator that takes its effect on

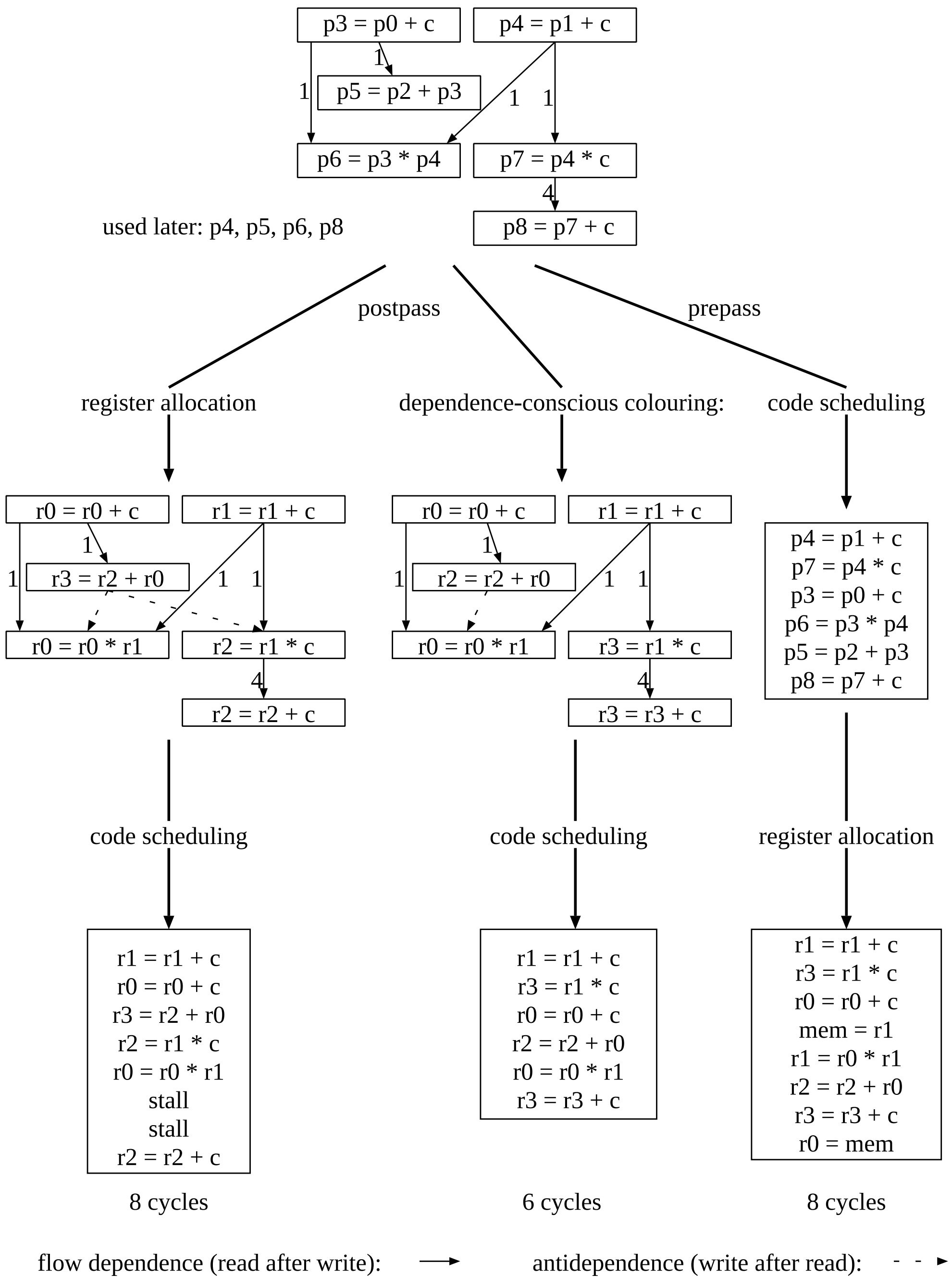


Fig. 1. Postpass scheduling vs. dependence-conscious colouring vs. prepass scheduling

scheduling into account. Figure 1 (middle) shows how this method can avoid inefficiencies: The register allocator is aware of the dependence graph and avoids introducing expensive dependences by selecting the right registers. Note that, while this example considers only one basic block, dependence-conscious colouring is a global register allocator.

The rest of the paper explains the data dependence graphs used in scheduling (Section 2), graph colouring register allocation (Section 3) and the two innovations of dependence-conscious colouring: the minimal interference graph (Section 4) and dependence-conscious register selection (Section 5). Finally, preliminary results are presented and dependence-conscious colouring is compared with other work.

2 The Data Dependence Graph

Current RISC processors achieve their high performance by exploiting parallelism through pipelining. As a consequence, the results of previous instructions are sometimes not available when the next instruction can be executed. If the next instruction uses the result, it has to wait and the pipeline stalls. The problem of arranging the instructions in a way that reduces the number of wait cycles is known as instruction scheduling. In this paper we consider only instruction scheduling within basic blocks.

The basic data structure for instruction scheduling is the dependence graph [GM86]. Figure 1 shows (several variations of) a dependence graph. An edge from instruction a to instruction b indicates that a must be executed before b to preserve the correctness of the overall program. Dependence edges must be drawn from writes to reads of the same register or memory location (flow dependences), from reads to writes (anti dependences), and from writes to writes (output dependences). The dependence graph is essentially the expression evaluation graph (drawn up-side-down), with some edges added due to dependencies between memory accesses. Register allocation can add antidependences (write-after-read dependences) by allocating the same register to several live ranges.

Path lengths of the dependence graph play an important role in instruction scheduling: There can be no schedule that is shorter than the critical path length. The edge length of a flow (read-after-write) dependence is the latency of the parent instruction. The length of an antidependence is zero or one cycle. However, if it connects two long paths, an antidependence can increase the critical path length and the execution time. Therefore, antidependences should be avoided or placed well.

3 Graph Colouring Register Allocation

The compiler front end and the optimizer can use an infinite number of live ranges (also known as pseudoregisters) for the variables and temporary values of the program. The task of register allocation is to map these live ranges onto a finite register set. The standard approach to register allocation is graph colouring [CAC$^+$81, Cha82, BCKT89, CH90, Bri92b]. As the basis for dependence-conscious colouring, we used the algorithm presented in [BCKT89]. Figure 2 presents the phases of a graph colouring register allocator.

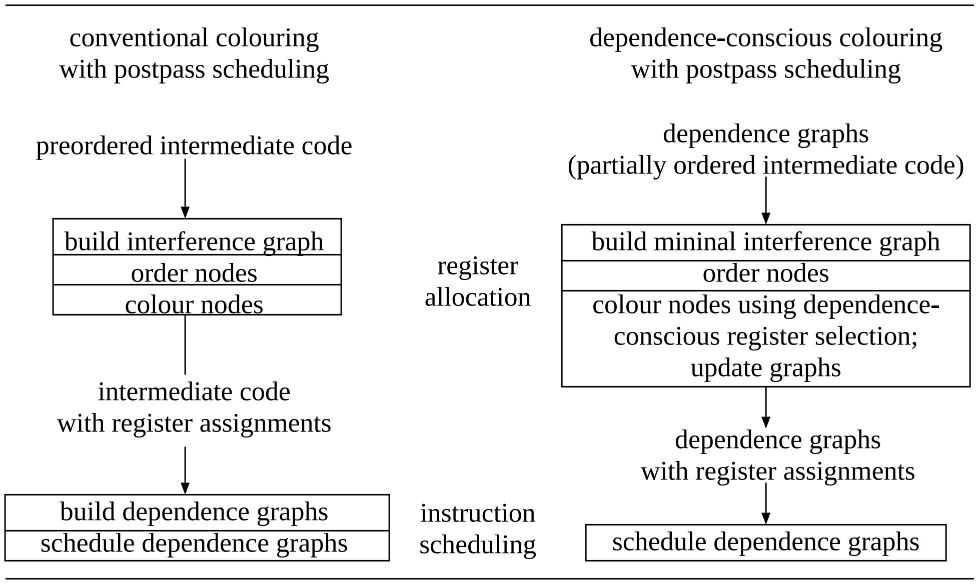

conventional colouring
with postpass scheduling
dependence-conscious colouring
with postpass scheduling
preordered intermediate code
dependence graphs
(partially ordered intermediate code)
build interference graph
order nodes
colour nodes
register
allocation
build mininal interference graph
order nodes
colour nodes using dependence-
conscious register selection;
update graphs
intermediate code
with register assignments
dependence graphs
with register assignments
build dependence graphs
schedule dependence graphs
instruction
scheduling
schedule dependence graphs

Fig. 2. Conventional graph colouring and dependence-conscious colouring

3.1 The Interference Graph

The basic data structure used by graph colouring register allocators is the interference graph. Every live range is represented by a node. There is an edge between two nodes if the live ranges interfere, i.e. if they must not stay in the same register, because they overlap. So, the problem of assigning registers to live ranges is mapped into the problem of "colouring" the nodes of the interference graph with registers such that directly connected (i.e., interfering) nodes do not get the same "colour" (register).

3.2 Colouring

Once the interference graph is complete, registers can be allocated in the following way: select an uncoloured node and give it a register that differs from the registers of the neighbours. There are several open points in this algorithm:

- In what order are the nodes coloured? Since dependence-conscious colouring does not differ from other colouring register allocators in this respect, the ordering will not be discussed here.
- What register is given to the live range? For register allocation purposes, it does not make much of a difference, which of the legal registers is selected for colouring a node [Bri92a]. However, for scheduling it makes a big difference (see Section 5).
- What happens if there is no register available for a live range? The live range has to be spilled into memory. Dependence-conscious colouring does not differ from other colouring register allocators in this respect.

4 The Minimal Interference Graph

The right side of Figure 2 shows the overall structure of dependence-conscious colouring.

In conventional graph colouring, the interference graph is computed from totally ordered code. In a postpass approach, this ordering usually is the coincidental result of some earlier phase. It could also be produced by a register-friendly scheduler. In any case, the ordering will cause some interference edges that need not be valid for the final schedule. These edges needlessly restrict register allocation.

To avoid this problem, dependence-conscious colouring computes the interference edges of a basic block from its dependence graph. I.e., the notions of "before" and "after" in a totally ordered basic block are replaced by the data dependence relation, which is a partial ordering.

4.1 Building the Minimal Interference Graph

The conventional method of computing live information and computing the interferences from that cannot be generalized straightforwardly to dependence graphs. Therefore we go back to the roots: Two live ranges interfere, if one is defined before the other is used and the other is defined before the first one is used. Formally:

$$\mathit{interfere\text{-}with}(l) = \mathit{used\text{-}later\text{-}out}(\mathit{definition}(l)) \cap \bigcup_{i \in \mathit{uses}(l)} \mathit{defined\text{-}earlier\text{-}in}(i)$$

interfere-with(l) is the set of live ranges that interferes with the live range l in the basic block; *definition*(l) is the node of the dependence graph where l is defined; *uses*(l) is the set of nodes where l is used. The *used-later* and *defined-earlier* information can be computed by applying data flow analysis techniques to the data dependence graph (Figure 3 shows an example):

$$\begin{aligned}
\mathit{used\text{-}later\text{-}out}(i) &= \bigcup_{j \in \mathit{successors}(i)} \mathit{used\text{-}later\text{-}in}(j)\\
\mathit{used\text{-}later\text{-}in}(i) &= \mathit{used\text{-}later\text{-}out}(i) \cup \mathit{use}(i)\\
\mathit{defined\text{-}earlier\text{-}in}(i) &= \bigcup_{j \in \mathit{predecessors}(i)} \mathit{defined\text{-}earlier\text{-}out}(j)\\
\mathit{defined\text{-}earlier\text{-}out}(i) &= \mathit{defined\text{-}earlier\text{-}in}(j) \cup \mathit{def}(i)
\end{aligned}$$

This method is only used for computing the interferences within basic blocks. Conventional data flow techniques are used for computing the global interferences, i.e., interferences between variables that are live at the same control flow graph edges. In order to get correct interference edges for non-local live ranges that become live or dead in the basic block, we insert into every dependence graph a top node $\top$ that is the definition point of all live ranges accessed in the basic block that are defined before the start of the basic block, and a bottom node $\perp$ that is a use point of live ranges accessed in the basic block that are used after end of the basic block.

The interference computation given above is only correct if the live range is contiguous within the basic block (otherwise it computes too many interferences).

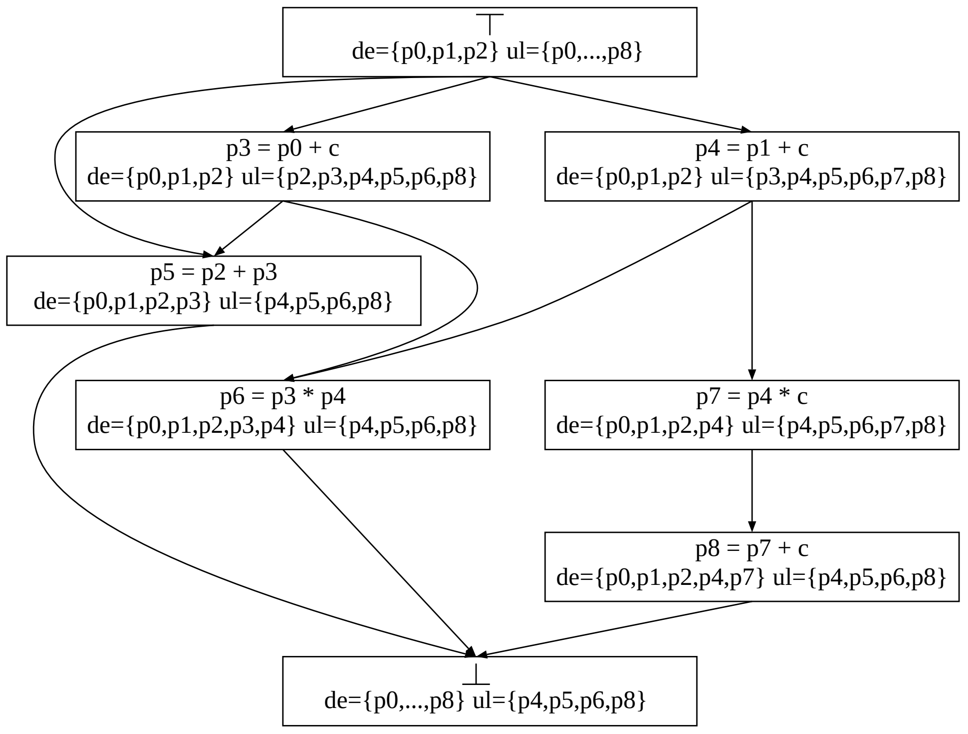
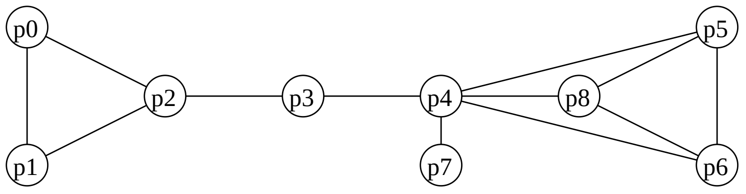

⊤
de={p0,p1,p2} ul={p0,...,p8}
p3 = p0 + c
de={p0,p1,p2} ul={p2,p3,p4,p5,p6,p8}
p4 = p1 + c
de={p0,p1,p2} ul={p3,p4,p5,p6,p7,p8}
p5 = p2 + p3
de={p0,p1,p2,p3} ul={p4,p5,p6,p8}
p6 = p3 * p4
de={p0,p1,p2,p3,p4} ul={p4,p5,p6,p8}
p7 = p4 * c
de={p0,p1,p2,p4} ul={p4,p5,p6,p7,p8}
p8 = p7 + c
de={p0,p1,p2,p4,p7} ul={p4,p5,p6,p8}
⊥
de={p0,...,p8} ul={p4,p5,p6,p8}


Fig. 3. Dependence graph of Figure 1 with *defined-earlier-in* and *used-later-out* sets and the resulting interference graph

Therefore, two definitions should be treated as belonging to two separate live ranges, even if they belong to the same global live range. There is one exception: If the second definition is in the same instruction as a use of the live range, the live range is contiguous in the basic block and should be treated as one live range.

Using this method, we get all interference edges that will be valid for every schedule and only those. The complexity for computing all interferences of a basic block is $O(e + u)$, where e is the number of dependence edges and u the number of register uses in the basic block.

4.2 Maintaining the Minimal Interference Graph

During colouring, the same register may be assigned to several live ranges in the basic block. Similarly, during coalescing[1] separate live ranges can be united. These actions may cause additional interferences:

Assigning the same register to different live ranges introduces antidependence edges in the dependence graph. If the antidependences are not redundant[2], new *defined-earlier* and *used-later* information may be propagated through them, possibly resulting in new interference edges. In other words, due to the antidependences the scheduler has less freedom to adapt to the register allocation, therefore the register allocator also has less freedom. In the example of Figure 3, allocating `p8` to the same register as `p2` causes an antidependence from `p5=p2+p3` to `p8=p7+c`, which causes an interference between `p5` and `p7`. Our prototype implementation handles these cases by recomputing the interference edges for the whole basic block, but we are investigating incremental recomputation. Fortunately, our register selection algorithm (see Section 5) avoids introducing non-redundant antidependences if possible.

Coalescing two live ranges introduces interferences between the united live range and live ranges that before coalescing could be assigned the same register as either of the coalesced live ranges. This can also occur if the same register is assigned to two live ranges and the definition of the second live range is in the same instruction as a use of the first. The interferences for the combined live range can be computed by

$$\textit{interfere-with}(l_{12}) = \textit{used-later-out}(\textit{definition}(l_1)) \cap \bigcup_{i \in uses(l_2)} \textit{defined-earlier-in}(i)$$

where l_{12} is the combined live range, l_1 is the first one and l_2 is the second.

Note that Chaitins original algorithm cannot handle adding interference edges during colouring, because it performs spill decisions before colouring. However, Briggs' modification [BCKT89] can handle it.

5 Dependence-Conscious Register Selection

For register allocation, it makes little difference, which of the available registers is selected [Bri92a]. But for the instruction scheduler the difference is important: Antidependences introduced by colouring can produce long dependence paths, which result in bad scheduling.

Therefore, dependence-conscious colouring carefully selects the registers. Colouring a live range with a register should introduce no antidependences or only redundant ones. If everything else fails, the introduced antidependences should connect only short paths. The problem is complicated by the fact that several basic blocks have to be considered at the same time: all basic blocks where the live range is born or dies. For all allowed registers the cost of the dependences introduced by selecting

[1] Coalescing is an optimization that is performed immediately after interference graph construction. It eliminates copies.

[2] A dependence between two instructions is redundant, if the ordering between the instructions is already enforced by other dependences.

the register is computed over all basic blocks; the register with the lowest weighted sum of costs is selected. The weights are the expected execution frequencies of the basic blocks.

The cost of a dependence d is the expected increase in the execution time of the basic block due to adding the dependence. It is 0 for redundant dependences. For non-redundant dependences we use the following cost function:

$$\mathit{cost}(d) = \max(\frac{\mathit{path\text{-}length}(d)}{\mathit{expected\text{-}time}}, \mathit{path\text{-}length}(d) - \mathit{expected\text{-}time})$$

$\mathit{path\text{-}length}(d)$ is the length of the longest path containing the dependence d. It can be computed in constant time, if the earliest issue and finish times [GH88] are precomputed. *expected-time* is the expected execution time of the basic block before adding the dependence:

$$\mathit{expected\text{-}time} = \max(\mathit{cycles}, \mathit{critical\text{-}path\text{-}length})$$

where *cycles* is the number of cycles the basic block would need if its instructions were independent (i.e., the naive expected execution time), and *critical-path-length* is the critical path length of the dependence graph of the basic block before adding the antidependence.

The cost of of an edge d is quite low as long as $\mathit{path\text{-}length}(d)$ is smaller than *expected-time*, but is high otherwise.

E.g., consider the basic block in Figure 3 after assigning registers to `p4`, `p5`, `p6` and `p8`. Figure 4 shows the costs of antidependences introduced by selecting a register for `p3`. Since `p3`s lifetime is restricted to that basic block, these are the total costs and `r1` or `r2` is selected for `p3`.

register	antidependences	path lengths	costs
r0 (p4)	–	–	interferes
r1 (p5)	p6=p3*p4 → p5=p2+p3	2	0.33
r2 (p6)	p5=p2+p3 → p6=p3*p4	2	0.33
r3 (p8)	p5=p2+p3 → p8=p7+c, p6=p3*p4 → p8=p7+c	2, 2	0.67

Fig. 4. The costs of selecting a register for p3

This selection process may seem to be expensive. But graph colouring register allocation is dominated by the time for building the interference graph (e.g., 90% of the register allocation time in [BCKT89]), so making the colouring slower does not make much of a difference for the whole algorithm.

Register selection is independent of the interference graph building method described in Section 4. Either method can be used separately.

6 Preliminary Results

We implemented a prototype dependence-conscious colouring allocator by modifying the register allocator of a C compiler for the Mips R3000. Since the C compiler was under development, the results are preliminary. They are shown in Table 1. We are currently reimplementing dependence-conscious colouring in the finished compiler.

We compiled two programs, a fast Fourier transform (FFT) and the Dhrystone integer benchmark (Dhry). We compared a conventional colouring register allocator ([BCKT89]) to dependence-conscious colouring (DCC). In both cases the compiler schedules after the register allocation (postpass scheduling). The programs consist of several procedures that are compiled one at a time. The presented data is cumulated.

program	FFT		Dhry	
register allocator	[BCKT89]	DCC	[BCKT89]	DCC
initial interference edges	1024	948	1383	968
additional interference edges		0		81
all antidependences	407	0	713	382
redundant antidependences		0		23

Table 1. Results

As expected, dependence-conscious colouring produces fewer interference edges while building the interference graph (initial interference edges). During colouring, dependence-conscious colouring inserts additional edges in the interference graph (additional interference edges), but there are still fewer interferences than from preordered code. Both allocators do not produce spill code. The conventional register allocator introduces a considerable number of antidependences. Dependence-conscious register allocation produces no antidependences for FFT and halves Dhry's antidependences. This means that for FFT dependence-conscious colouring achieves the best behaviour possible: no spilling and full scheduling freedom. Unfortunately we do not have speedup numbers, since the compiler back end was still under development and did not produce fully functional code.

7 Related Work

The standard approach to the problem is to more or less ignore it. Scheduling is performed either before register allocation (prepass, [AH82]) or afterwards (postpass, [HG83]). With the prepass approach, scheduling has to be repeated after register allocation to schedule spill code. Prepass schedulers usually employ a register-saving heuristic, but only as low-priority secondary heuristic. Our approach is postpass scheduling, but our register allocator takes scheduling into consideration.

The Harris C compiler uses postpass scheduling, but reallocates registers during scheduling to remove harmful dependences [Beu92]. In contrast, our register allocator tries to do it right the first time.

In [GH88] two techniques for integrating local register allocation and instruction scheduling are introduced: Integrated prepass scheduling switches between scheduling for pipelining and scheduling for register allocation based on the number of available registers. DAG-driven register allocation tries to avoid introducing long paths into the dependence graph. The performance of both methods is about equal, with DAG-driven register allocation being simpler. Dependence-conscious colouring can be seen as the global version of DAG-driven register allocation.

[BEH91] presents integrated prepass scheduling in a global register allocation setting and introduces register allocation with schedule estimates (RASE). In RASE, the global register allocator leaves a number of registers to the local allocator, which also performs instruction scheduling. The number of registers left for local allocation is determined by the estimated costs of scheduling with x registers. The estimates are computed from practice runs of the scheduler. RASE and integrated prepass scheduling are about equal in code quality; integrated prepass scheduling is simpler. Like RASE, dependence-conscious colouring makes the register allocator aware of scheduling. But dependence-conscious colouring does all of the register allocation, including local allocation. It directly sees the data dependence graphs and the effects of allocation decisions on it instead of just heeding a register limit.

[Pin93] proposes using postpass scheduling with a modified register allocator. The register allocator uses an interference graph that contains all interference edges that could be introduced by scheduling. I.e., even more interference edges and more spilling than in a prepass scheduling approach. Pinter proposes heuristics for removing edges from that interference graph to avoid excessive spilling, but does not give results. In contrast, dependence-conscious colouring uses a minimal interference graph to minimize spilling and preserves scheduling freedom through its register selection heuristics.

[PF91] gives an optimal algorithm. Unfortunately it solves a very limited and unrealistic problem: scheduling and stupid register allocation for binary expression trees with single-delay-slot loads at the leaves. I.e., no unary operators, no constants or register variables, no common subexpression elimination and the algorithm is restricted to one expression. In contrast, dependence-conscious colouring does not have these restrictions and performs global register allocation.

In [FR91] instruction scheduling and register allocation are performed tracewise[3], starting with the most frequently executed trace. This approach is similar to coagulation [Mor91]. In contrast to dependence-conscious colouring, the first phase does not consider the needs of the second; instead, the phases are interleaved, so possible problems are pushed into low-frequency code.

[RLTS92] discusses register allocation for globally scheduled loops. In contrast, dependence-conscious register allocation can handle general control structures, but is restricted to basic block instruction scheduling.

8 Conclusion

Dependence-conscious colouring is a global register allocation method based on graph colouring, that takes the needs of instruction scheduling into account. It consists of two independent techniques:

[3] Traces are parts of possible execution paths. You can think of them as multiple-entry, multiple-exit basic blocks.

– The *minimal interference graph* is built by analysing dependence graphs of basic blocks instead of preordered code. This eliminates unnecessary interference edges, making colouring easier.
– *Dependence-conscious register selection* is aware of the antidependences that it can introduce. It keeps dependence graph path lengths short by selecting the right registers for assignment. This provides for good instruction scheduling.

Dependence-conscious colouring reduced the number of interference edges by 7%–24% and antidependences by 46%–100% in the two programs we measured.

Acknowledgements

We want to thank the referees, Bjarne Steensgaard, and Franz Puntigam for their helpful comments on earlier versions of this paper.

References

[AH82] Marc Auslander and Martin Hopkins. An overview of the PL.8 compiler. In SIGPLAN '82 [SIG82], pages 22–31.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275–284, 1989.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 122–131, 1991.

[Beu92] Paul Beusterien. Personal communication, 1992.

[Bri92a] Preston Briggs. Personal communication, 1992.

[Bri92b] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, 1992.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):45–57, 1981. Reprinted in [Sta90].

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In SIGPLAN '82 [SIG82], pages 98–105.

[FR91] Stefan M. Freudenberger and John C. Ruttenberg. Phase ordering of register allocation and instruction scheduling. In Robert Giegerich and Susan L. Graham, editors, *Code Generation — Concepts, Tools, Techniques*, Workshops in Computing, pages 146–170. Springer, 1991.

[GH88] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, pages 442–452, 1988.

[GM86] Phillip B. Gibbons and Steve S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, 1986.

[HG83] John Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[Mor91] W. G. Morris. CCG: A prototype coagulating code generator. In SIGPLAN '91 [SIG91], pages 45–58.

[PF91] Todd A. Proebsting and Charles N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. In SIGPLAN '91 [SIG91], pages 256–267.

[Pin93] Shlomit S. Pinter. Register allocation with instruction scheduling: A new approach. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 248–257, 1993. SIGPLAN Notices 28(6).

[RLTS92] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 283–299, 1992.

[SIG82] *SIGPLAN '82 Symposium on Compiler Construction*, 1982.

[SIG91] *SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

[Sta90] William Stallings, editor. *Reduced Instruction Set Computers.* IEEE Computer Society Press, second edition, 1990.

This article was processed using the LaTeX macro package with LLNCS style