Abstract — Instruction scheduling and register allocation are two fundamental operations used in an optimizing compiler’s back-end. There is a well-known phase ordering problem between these two stages: Performing either stage first can result in intermediate code that forces the other stage to make suboptimal decisions.

We propose an optimistic integrated approach in which scheduling is performed before register allocation, but where not all scheduling decisions are final; rather, the register allocator may use meta-information computed by the scheduler to rearrange the order of certain instructions to find a good match of register usage to the number of actually available registers.

We outline our intended approach and goals in this very early work-in-progress.

I. INTRODUCTION

Two of the central activities in a compiler’s code generating back-end are instruction scheduling and register allocation. The instruction scheduling phase takes a partial order of machine instructions and attempts to compute a total order that minimizes total execution time by overlapping computations that can be executed in parallel on a processor with multiple functional units. For example, a long memory load may be overlapped with a number of quicker integer arithmetic instructions.

Register allocation attempts to map the values used by the program into CPU registers for very fast access. A central notion in register allocation is that of liveness: A value is said to be live in the interval from the point of its definition until its last use. Values that are live in overlapping intervals and not known to be equal must be allocated to different registers; if at some program point more values are live than can be accommodated in the registers provided by the CPU, some of them must typically be spilled to memory and reloaded at appropriate points later in the program. As memory accesses can incur very long delays, the register allocator should be careful to minimize the number of spills executed by the program.

The two stages exhibit a well-known phase ordering problem: Performing scheduling before register allocation may lengthen live ranges. This increases the number of conflicts and makes expensive spilling more likely. Conversely, register allocation may assign the same register to two otherwise independent instructions; this limits the choices a later scheduling pass may make. Thus, both possible orderings of independent scheduling and register allocation phases can have an overall negative effect on the program’s execution time.

II. INTEGRATED APPROACHES

Integrated approaches attempt to find a good trade-off between the needs of instruction scheduling and register allocation; a prominent example is integrated prepass scheduling (IPS) [1]. In IPS, the scheduler attempts to estimate the number of live registers after each scheduled instruction and lets this estimate guide its choice of whether to schedule for latencies or registers. If the number of live registers becomes too low, IPS will switch to a mode that prefers to schedule instructions that free registers; if such scheduling steps succeed in increasing the number of free registers above some threshold, the scheduling mode is switched back to latency-oriented scheduling.

IPS makes a good trade-off between scheduling to minimize latencies and register usage; it is able to produce considerably more efficient code than a non-integrated approach. However, it may be more ‘pessimistic’ than necessary: When a value may be live in a basic block, IPS always assumes that it will be allocated in a register. It may therefore commit to a schedule with very low register usage. However, it might be more profitable to use more registers, if some values can be spilled outside the block.

III. OPTIMISTIC SCHEDULING

We propose an optimistic integrated scheduling and register allocation approach. Rather than prematurely committing to an overly conservative schedule, the optimistic scheduler assumes that the primary objective is to mask instruction latencies. It computes an appropriate schedule, but it also generates meta-information to be used by the register allocator to reschedule certain instructions if this can help it to avoid spilling.

The register allocator in the optimistic scheme is a linear scan allocator [2] which passes over the block in scheduled order, allocating registers for each instruction.
When all machine registers are used up and the following instruction needs an additional register, the usual allocator must spill one of the conflicting intervals to memory. At this point, the optimistic allocator adds another option: Neither live range need be spilled if it is easy to reschedule instructions to avoid the overlap of live intervals. If this rescheduling succeeds, the non-overlapping live intervals can both be allocated to the same machine register without conflict.

The register allocator can thus override certain scheduling decisions by more register-friendly alternatives if they are needed, but it may also keep to the optimistic efficient schedule if actual register use is not as high as estimated by the preceding scheduling phase. We expect this scheme to be profitable and produce even better code than pure IPS if we can find good heuristics to enable rescheduling.

The planned optimistic instruction scheduler computes the following meta-information during the preliminary scheduling phase:

Possible scheduling points. The scheduler computes for each instruction the earliest point where it could be scheduled, given the partial schedule determined up to that point. This is the point in the schedule where the instruction became available because all of its predecessors in the dependence DAG had been scheduled. The instructions having an earliest scheduling point that precedes their position in the actual preliminary schedule are candidates for motion to an earlier point. An instruction’s latest scheduling point is approximated by taking the point just before its first scheduled successor; the instruction may be delayed up to this point.

Register usage. For each movable instruction, we compute the number of live intervals that start at the instruction and the number of live intervals that end there.

Once the preliminary schedule and meta-information are computed, the modified rescheduling linear-scan register allocator is started. At any point where no more registers are available, it attempts to reschedule instructions as follows:

Candidate selection. The register allocator identifies all candidates for motion to free a register for use at the current point. These are either instructions that free at least one register and can be moved to a point before the allocator’s current position, or instructions that start live ranges and can be delayed to a point past the current position. If there is more than one candidate, one is chosen heuristically, taking into account each candidate’s latency and register use.

Rescheduling. The chosen candidate is moved in the appropriate direction, either to an earlier point to end a live interval, or to a later point to delay the start of a live interval. We attempt to find a most profitable position, balancing the number of registers freed by the instruction against the latencies of the preceding instructions. The earliest scheduling points of later instructions must be updated to preserve correctness.

Reallocating. Since moving the instruction to its new position changes the number of available registers after that point, the register allocator may have to backtrack by reverting all register assignments after that point and starting again from there. To avoid uncontrolled backtracking, we advance a barrier pointer to ensure that future reschedule operations do not hoist code past the restart point.

If the register allocator does not have any profitable instructions to reschedule, it chooses a virtual register for spilling. This operation is the same as in the unmodified linear scan allocator.

IV. IMPLEMENTATION AND EVALUATION

The rescheduling register allocation approach described above is currently only at the planning stage. We intend to implement it using the LLVM\(^1\) compiler framework. The allocator’s compile time performance and the performance of the generated machine code will be compared against LLVM’s existing scheduler/allocator combinations and a version of IPS, of which we have implemented a prototype. We expect the code generator based on the rescheduling allocator to have competitive performance and to generate better code than the other approaches. By tuning the heuristics for selecting which instructions to reschedule where, we will be able to investigate the costs and benefits of optimal or near-optimal integrated scheduling and register allocation.

ACKNOWLEDGMENTS

This work is partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P21842, Optimal Code Generation for Explicitly Parallel Processors, http://www.complang.tuwien.ac.at/epicopt/.

REFERENCES


\(^1\)http://www.llvm.org