Optimistic Integrated Instruction Scheduling and Register Allocation

Gergö Barany and Andreas Krall (Faculty Mentor)

Institute of Computer Languages, Vienna University of Technology
{gergo, andi}@complang.tuwien.ac.at

Context: Instruction Scheduling and Register Allocation

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. 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 available to a subsequent scheduling pass. Thus, both possible orderings of independent scheduling and register allocation phases can have an overall negative effect on the program's execution time.

Scheduling Heuristics

Various scheduling techniques have been proposed in the literature, each with its own advantages and drawbacks.

Latency-based scheduling attempts to minimize the number of unused processor cycles by masking them with other instructions if possible. This can produce fast code but lengthen live ranges of values. Register allocation may become difficult, spill code may have to be introduced.

Scheduling for register pressure shortens live ranges to minimize the number of registers used and spilled. It can result in long unmasked pipeline delays.

Integrated Prepass Scheduling (IPS) [1] estimates register usage based on a global liveness analysis. It then makes a trade-off between scheduling for minimal latencies and minimal register pressure.

Our approach also results in a trade-off between latency and register pressure. It starts with a schedule for minimal latencies and modifies it during register allocation, based on actual register use information.

Idea: Optimistic Scheduling and Rescheduling

We propose to resolve the phase-ordering problem of instruction scheduling and register allocation by implementing a *rescheduling* register allocator. The proposed integrated instruction scheduler and register allocator first performs aggressive, *optimistic* scheduling to mask instruction latencies. If some scheduling decision turns out to be suboptimal, the register allocator can reschedule instructions to optimize register usage.

The approach can use an existing instruction scheduler to compute the initial schedule. It may be beneficial to modify the scheduler to compute meta-information about the ranges of positions where each instruction can be scheduled. The rescheduling register allocator builds on a linear scan register allocator [2]. It visits the instructions in a basic block in a single linear sweep, assigning registers and inserting spill/reload code where necessary.

Our extension gives the register allocator the chance to avoid spill code in certain situations: If two conflicting live ranges overlap, the register allocator may attempt to *reschedule* instructions to eliminate the conflict. If the register allocator manages to hoist the end of one live range or sink the beginning of the other live range far enough to avoid the overlap, both live ranges may be assigned to the same physical register without introducing spill code.

The freedom to choose between spilling and rescheduling means that the register allocator can undo overly optimistic scheduling decisions exactly where needed. It can leverage all of its heuristics for trading off the costs of spilling different live values, and compare the estimated costs and benefits to those of changing the order of the instructions.

We expect this to result in good code generation because scheduling and register allocation decisions can be made in an integrated fashion depending on the program context. For example, hot nested loops can be scheduled very aggressively with maximal register utilization, preferring to spill values from outer loops; conversely, register use in large basic blocks can be reduced to a point that is just the appropriate trade-off between schedule length and spilling overhead.

Comparison with Other Approaches

In contrast to latency-based scheduling our approach can schedule instructions to shorten live ranges where profitable. This can ease register allocation and avoid costly spilling. Spilling is avoided at the expense of unused processor cycles to fill pipeline hazards.

In contrast to register pressure based scheduling our approach prefers to schedule to mask instruction latencies. It optimistically assumes that the register allocator will reschedule to reduce register pressure exactly where needed.

In contrast to IPS our approach is *optimistic* in that it does not prematurely avoid spilling at all costs, based on an approximation of possible register usage. Rather, we make a decision to rearrange instructions or spill a register at register allocation time, when exact register usage information is available. This should be significantly more profitable especially for nested inner loops. Here, IPS schedules conservatively to reserve registers for values that may only be live through, but not used in, the innermost loop.

Implementation Status and Evaluation

The rescheduling register allocator will be implemented by extending the linear scan register allocator provided by the LLVM compiler framework [3]. We will evaluate the performance of the generated code and compare it against other scheduling techniques including IPS. At the time of printing of this poster, implementation is at a very early stage. *Ask me in person about the current status!*

Challenges and Research Directions

In addition to implementing and evaluating the integrated code generator described here, we have identified a number of challenges and directions for further research.

Choice of Rescheduling Heuristics

The possibility of rescheduling raises two essential issues. First, how should costs and benefits of rescheduling be weighted against estimated spill costs? A promising heuristic may be to schedule aggressively within loops, allowing values from outside the loop to be spilled, but as far as possible to avoid spilling of values used within the loop.

Second, how should one choose the position where a rescheduled instruction is inserted? Our first implementation will hoist instructions to exactly the point where the register allocator would otherwise choose to spill a value. However, it may be profitable to hoist instructions to even earlier points. We need to investigate the implications of such eager code motions.

Formal Characterization of Code Quality

During manual exploration, we have found that our approach tends to result in the same code as IPS if we attempt to avoid all spilling. However, as discussed above, we believe that some spills may be profitable if they allow other code to be scheduled more aggressively. We would like to investigate this more thoroughly and give a precise characterization of the conditions under which our heuristics generate code that is either equivalent to IPS or strictly better than IPS, respectively.

Integration with Different Register Allocators

We intend to investigate how to integrate the rescheduling approach with register allocators that are not based on the linear scan algorithm. For graph coloring register allocators, rescheduling would amount to breaking edges in the conflict graph. We need to find a way of identifying situations in which this transformation is profitable; further, the effects of rescheduling on the rest of the graph must be investigated.

Finally, we are interested in integrating our method with optimal or near-optimal heuristic register allocators based on techniques such as integer linear programming or PBQP (Partitioned Boolean Quadratic Problem) solving. If an appropriate way of combining these approaches can be found, we might obtain a novel near-optimal integrated code generation algorithm.

Example

We present our integrated instruction scheduling and register allocation approach using an example due to Goodman and Hsu [1]. The example code is first scheduled for minimum latency using standard techniques; then, our modified linear scan register allocator assigns physical registers to virtual registers.

In order to avoid inserting spill code where it is deemed too expensive, the register allocator may instead choose to reschedule the instruction stream. It can do so by hoisting instructions that end live ranges, thereby freeing physical registers for use by other values.

I. Example basic block using virtual registers, prescheduled to maximize pipeline utilization and minimize runtime delay cycles:

```
 4  load  VR4 ← c
 5  load  VR5 ← d
 7  load  VR7 ← e
 1  load  VR1 ← a
 2  load  VR2 ← b
 6  add   VR6 ← VR4 + VR5
 8  add   VR8 ← VR1 + VR7
 3  mul   VR3 ← VR1 * VR2
 9  mul   VR9 ← VR6 * VR8
10  add   VR10 ← VR3 + VR9
11  store h ← VR10
```

→

II. Start linear-scan register allocation; assume three registers are available or can be freed by spilling other values (not shown).

```
 4  load R1 ← c
 5  load R2 ← d
 7  load R3 ← e
 1  load VR1 ← a
```

Instruction 1 cannot be allocated a physical register for VR1 without spilling because all three registers are already in use. The register allocator decides to *reschedule* to free a register: It hoists an instruction that ends some live ranges.

→

III. Hoisted instruction 6 to free a register; register allocation was able to continue and assign a physical register for instruction 1.

```
 4  load R1 ← c
 5  load R2 ← d
 7  load R3 ← e
 6  add  R1 ← R1 + R2
 1  load R2 ← a
 2  load VR2 ← b
```

Again, no physical register can be assigned to VR2 without spilling. *Reschedule* again, hoisting instructions 8 and 9, which free registers.

→

IV. Finish register allocation without spilling, but still keeping to a schedule that is as aggressive as possible:

```
 4  load  R1 ← c
 5  load  R2 ← d
 7  load  R3 ← e
 6  add   R1 ← R1 + R2
 1  load  R2 ← a
 8  add   R3 ← R2 + R3
 9  mul   R3 ← R1 * R3
 2  load  R1 ← b
 3  mul   R1 ← R2 * R1
10  add   R1 ← R1 + R3
11  store h ← R1
```

References

[1] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 2nd international conference on Supercomputing*, pages 442–452, New York, NY, USA, 1988. ACM.

[2] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21:895–913, September 1999.

[3] The LLVM compiler infrastructure. http://www.llvm.org.

Acknowledgments

This work is 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/.