Scheduling Instructions by Direct Placement

Robert Griesemer
griesemer@inf.ethz.ch

Institute for Computer Systems
Swiss Federal Institute of Technology (ETH Zurich)
Switzerland

Abstract. This paper presents a simple and efficient method for instruction scheduling within basic blocks. An implementation proved to be extremely small while producing results comparable to other more complicated techniques. The algorithm is of quadratic complexity in the number of instructions but a linear run-time is achieved in practice. Because no (code) look-ahead is needed, the algorithm is even suitable for one-pass compilers.

1 Introduction

A well-known technique to improve the execution speed of pipelined computers is the reduction of *pipeline interlocks* due to data dependencies. This is achieved by *instruction scheduling*, i.e. the reordering of instructions such that interlocks are avoided but data dependencies are preserved.

We present a new scheduling heuristic (to the best of our knowledge) operating on basic blocks, which is much simpler than other techniques [Gro83, Hen83, Gib86] while performing comparably well. Optimal code ordering and optimal register allocation are conflicting goals and normally *phase ordering* problems occur. Recent work tries to improve the situation by using information of the scheduling phase to drive register allocation [Bra91]. We circumvent the sea of troubles by doing *global register allocation for variables* in a first pass but *not for temporaries* occurring during expression evaluation. Despite this common and in some way "naive" approach, good results are obtained when enough registers are available. We directly generate the final code for basic blocks, except that the exact register numbering is unknown. During this code generation phase, a simple method allows us to *directly place* an instruction at a *good position* from the scheduling point of view without using a DAG as in traditional compilers. Local register allocation is done in a second pass, thereby avoiding artificial dependencies: This is in contrast to postpass approaches, which try to improve the code after compilation [Gro83, Hen83, Gib86, Dav84, Dav86].

We have implemented the scheduler as a part of a simple cross-compiler for the Cray Y-MP for an experimental language. Although the algorithm is $O(n^2)$, in practice a linear run-time in the number of instructions was measured.

2 A Simple Machine Model

In the sequel we consider basic blocks only, i.e. a restricted instruction set is sufficient (no jump or call instructions are of importance). Conditional jumps depend directly on the value of a register (specified with the jump), thus no condition code concept has to be introduced. A few instruction types will help us to demonstrate the scheduling algorithm using a simple example (Fig. 1).

Type	*Instruction*	*Operation*
1	Ri := Rj op Rk	binary operation op
2	Ri := f(Rj)	unary operation f
3	Ri := c(val)	constant operation c with immediate argument val
4	Ri := M[addr + Rj]	load from memory (aliasing possible)
5	M[addr + Rj] := Ri	store to memory (aliasing possible)
6	Ri := var	load from memory or register file (no aliasing possible)
7	var := Ri	store to memory or register file (no aliasing possible)

Fig. 1

We distinguish between two different load/store instruction classes for optimization reasons: General memory access is done by the *Ri := M[addr + Rj]* and *M[addr + Rj] := Ri* instructions (Fig. 1, Type 4 and 5), where *aliasing* is taken into account (i.e. two load or store instructions may access the same memory location although its arguments are different). In all other cases we use the load/store instructions *Ri := var* and *var := Ri* (Fig. 1, Type 6 and 7). This mainly has consequences for instruction scheduling (dependencies).

In the following, an unbounded number of registers is assumed. Hence, it is possible to assign a new (result) register number to each computation. We even may identify the current instruction count with the result register number (i.e. these register numbers need not be stored). Thus, a possible code sequence for the expression *A[i, j] + A[i, k]* might look as follows:

PC	*Instruction*	*A[i, j] + A[i, k] with A: ARRAY 100, 100 OF INTEGER and i, j, k: INTEGER*	
0	R0 := i	i	
1	R1 := 100	100	
2	R2 := R0 * R1	i*100	
3	R3 := j	j	
4	R4 := R2 + R3	i*100 + j	
5	R5 := M[addr(A) + R4]	A[i, j]	*no index check generated*
6	R6 := k	k	
7	R7 := R2 + R6	i*100 + k	*common subexpression eliminated*
8	R8 := M[addr(A) + R7)	A[i, k]	
9	R9 := R5 + R8	A[i, j] + A[i, k]	

Fig. 2

3 Instruction Scheduling

During code generation, new instructions are simply appended to the already existing code stretch. We modify this code emitting scheme such that most interlocks are prevented: Assuming that an instruction is issued every clock cycle (without interlocks), the instruction count may be interpreted as *current clock*. Hence, instead of appending consecutive instructions, they are *directly placed* in the code array at the *earliest position in time* where no interlock is possible any more (because its operands are available by then). This leads to a surprisingly simple method for scheduling instructions, while observing dependencies. The following example will illustrate the situation:

Let *I(a, b)* be an instruction depending on the contents of registers *a* and *b* (i.e. the results of the instructions starting at time *a* and starting at time *b* in our encoding scheme, Section 2). When

E(a) and *E(b)* are the execution times of these instructions, *a+E(a)* and *b+E(b)* are the *times* when each result is available. Consequently, both results are available at *Max(a+E(a), b+E(b))*. In our array expression example (Fig. 2), the computation *R2 + R3* depends on *R2* and *R3* and the earliest issue time is *6 = Max(2+4, 3+1)*, using the execution times in Fig. 3. Instead of simply appending the instruction as before, it is placed at position 6 (remember that the result register number equals the instruction position). The resulting *free instruction slots* at position 4 and 5 may be thought of beeing filled with no-ops.

PC	*Instruction*	*Computation*	*Execution time*	*Result available at*
0	R0 := i	i	1	1
1	R1 := 100	100	1	2
2	R2 := R0 * R1	i*100	4	6
3	R3 := j	j	1	4
4	*no-op*			
5	*no-op*			
6	R6 := R2 + R3	i*100 + j	2	8

Fig. 3

Hence, whenever a new instruction has to be appended, its earliest issue time is determined based on its data dependencies and the instruction is then placed at the calculated position. Placing an instruction past the end of the current code block extends the block. When placing an instruction in the middle of the code block, the instruction replaces the *next* free instruction slot (possibly at the end of the code block). Continuing this way, our array example leads to the code sequence shown in Fig. 4.

PC	*Instruction*	*Computation*	*Execution time*	*Result available at*
0	R0 := i	i	1	1
1	R1 := 100	100	1	2
2	R2 := R0 * R1	i*100	4	6
3	R3 := j	j	1	4
4	R4 := k	k	1	5
5	*no-op*			
6	R6 := R2 + R3	i*100 + j	2	8
7	R7 := R2 + R4	i*100 + k	2	9
8	R8 := M[addr(A) + R6]	A[i, j]	14	22
9	R9 := M[addr(A) + R7]	A[i, k]	14	23
10	*no-op*			
...	...	...	...	...
22	*no-op*			
23	R23 := R8 + R9	A[i, j] + A[i, k]	2	25

Fig. 4

The placing rules are a little bit more complex, when store instructions are involved: To preserve the semantics of a code sequence, the relative ordering of consecutive loads and stores must remain unchanged. Hence, it is necessary to know where previous memory accesses occured. Such informations can be kept e.g. in a table for each instruction type. We do not explain these aspects further.

The code generated this way is good, however not optimal. The main reason for this is that

instructions drift forward in code if no suitable instruction slot is available. Sometimes it would be better to choose a (locally) suboptimal instruction position taking an interlock cycle into account. Assume a time consuming instruction *I* to be dependent on instruction *2* in our example (Fig. 4). The earliest possible position is *6* which is already occupied. Hence *I* drifts down until a free instruction slot is found, that is at position *10*. An instruction *J* depending on *I* is therefore positioned at *10+E(I)*. If *J* is the last instruction in the basic block, it would have been better to insert *I* at position *5*, taking one interlock cycle into account, and thereby gaining five interlock cycles. Nevertheless, if basic blocks are sufficiently large the algorithm works well: most free instruction slots are filled up with subsequent instructions.

With the representation suggested in Section 2, only a relatively simple array data structure is necessary. The length of the instruction sequence directly represents the time used for its execution. When transforming the intermediate code into target machine code, free instruction slots may be translated into no-ops for architectures with no hardware interlocks (e.g. MIPS, [Kan87]). Further, the scheduling algorithm may be combined with a *common subexpression elimination* algorithm: Before trying to place a new instruction in the code array, the already generated code is searched for this instruction. If the instruction is found, no code is emitted, but a reference to the found instruction is remembered instead. The run-time complexity of the scheduler is $O(n^2)$: In the worst case all instructions are independent and a free instruction slot must be searched from the beginning of the code block.

4 Target Code Generation

After the intermediate code for a basic block has been generated, it must be translated into the target machine code. That is, for each intermediate instruction one or several *target machine instructions* have to be selected and *local register allocation* must be done. When register pressure is too high, eventually *spill code* has to be inserted. The techniques described here are well-known and mentioned only to sketch the environment of the instruction scheduling algorithm.

Target instruction selection: When the intermediate machine instructions differ from the target machine instructions only in the unbounded number of registers used, this part of the translation process is obviously trivial. Hence, we propose to choose the intermediate code instructions to match *exactly* the target code instructions (despite the unlimited number of registers). Although this model is quite simple, it covers several popular machine architectures [Cra88, Dec92, Kan87, Sun87]. Irregularities in the instruction set may be fixed by introducing special intermediate instructions which must be translated into several target instructions. However, it will probably be difficult to support irregular architectures like the i860 CPU or the 80x86 family.

Register allocation: Note that at this level only registers for temporaries occurring during expression evaluation have to be allocated. Hence, a set of general purpose registers must be *reserved* for expression evaluation. During translation of the intermediate code, whenever a new result is computed (i.e. basically for every instruction except for stores), a new real register is allocated out of this set using a simple stack or round-robin algorithm. The real register number is remembered together with the corresponding virtual register. Whenever the virtual register is referenced again, the real register number is used instead. However, at some point, the virtual register is not referenced any more and the corresponding real register should be released. This situation is easily detected using reference counts. A similar technique was described by Freiburghouse [Fre74].

Spill code generation: When straight-forward code is generated for expressions, only a few registers are usually necessary to hold temporaries. However, instruction scheduling may significantly raise *register pressure*, i.e. the number of registers which are alive at any one time. Excessive register demand may require to spill currently unused registers to memory. When spilling to memory, reload instructions have to be inserted which may impose new interlocks.

In our context (Cray Y-MP) spilling is relatively cheap: because of the large number of

temporary registers (64 registers for each of two main register classes), a good portion of the register files can be reserved for this purpose. Load and store accesses to temporary registers execute in one cycle, therefore no additional interlocks are introduced. Spilling to main memory would be impractical due to the large interlocks introduced (14 cycles [Rob89]) as long as no additional scheduling pass is involved. However, on machines with a single cycle delay for memory access this might be acceptable.

5 Measurements

We have integrated our instruction scheduler in a compiler for an experimental subset of the Oberon language [Wir88]. Compilation is performed in three phases using a well-known technique: first, during lexical analysis an abstract syntax tree is built, semantic checks and minor optimizations are made [Cre90]. At the same time, a weighted usage count is computed for each variable. Using this information, variable allocation is done in a second phase, allocating registers for the statically most frequently used variables. It is hoped that the static measurements reflect the dynamic variable usage. Note that temporaries are not considered in this phase. Thirdly, the syntax tree is traversed and intermediate code is generated using the instruction scheduling method described above. The intermediate code differs from the real machine code only in the use of unbounded register numbers, and is quite good without any further optimizations (as long as no interlocks are concerned). This in contrast to other compilers (e.g. [Aus82]), where in a first pass relatively simple code is generated, which *must* be improved during a lot of optimization steps.

The Compiler generates code for the Cray Y-MP, but in our sample programs no attempt for vectorization is made. The Cray Y/MP is a heavily pipelined machine and hence, despite its small clock cycle time of 6.5ns, simple scalar programs perform poorly without instruction scheduling because of long delays. Integer operations have an execution time of 2-3 cycles (depending on the operation), 64-bit real operations need 6-7 cycles and reciprocal approximation as well as scalar memory access take 14 cycles [Kan87].

	Reduction of: Execution time (Cray)				*NOPs (Cray)*			*NOPs (Hyp. M.)*		
Program	cf77	IS -	IS +	Red.	IS -	IS +	Red.	IS -	IS +	Red.
MatrixMul	0.07821s	0.40670s	0.32225s	21%	101	70	31%	32	7	78%
MatrixInv	0.24289s	0.49889s	0.35505s	29%	482	317	34%	90	13	86%
Gradient	0.67267s	1.65865s	1.07755s	35%	296	171	42%	56	15	73%
Fractal	4.42827s	7.43025s	4.70283s	37%	-	-	-	-	-	-
Quicksort	-	0.11618s	0.10821s	7%	165	136	18%	16	12	25%
Sieve	0.39802s	0.83557s	0.77295s	7%	83	64	23%	7	7	0%
Bubble	-	0.69936s	0.60650s	13%	268	247	8%	23	15	35%
Fibonacci	-	2.91846s	2.81848s	3%	67	60	10%	5	4	20%
Average				19%			24%			45%

Table 1 Table 2 Table 3

Tables 1 to 3 show static and dynamic aspects for some characteristic programs. The first three routines operate on 100 by 100 matrices, Fractal calculates a 400 by 290 pixel Mandelbrot picture; all make heavy use floating-point arithmetic. The last four programs are integer benchmarks. Table 1 shows the absolute execution times (IS-) and the reduction (Red.) thereof gained by instruction scheduling (IS+). As expected, the number crunching applications offer more potential for improvements due to larger basic blocks. Overall, the instruction scheduling optimization achieves about 19% shorter execution times. When concentrating on floating-point intensive

applications, the improvement is 30% in average (without extending basic blocks, e.g. by loop unrolling). The dynamic improvements are comparable to the results described in [Gro83] for the reorganizer of pipeline constraints.

The corresponding Fortran programs (Table 1, cf77 column) perform much better; although vectorization is disabled. Massive DO loop unrolling [Don79] leads to this improvements, as can be seen when analyzing the corresponding cf77 code. However, loop unrolling is an orthogonal optimization which we have not supported in our compiler (unfortunately it seems to be impossible to disable unrolling done using cf77 5.0). Where this optimization is not possible (Fractal), our code is only about 6% slower than the cf77 code. Vice versa, when doing loop unrolling by hand, about the same speedups are gained (e.g. a factor of four for the MatrixMul routine).

Table 2 shows how many interlocks have been found statically due to data dependency conflicts before and after instruction scheduling. For this mix of basic blocks a static reduction of about 24% is achieved. Again, for the applications with long basic blocks (MatrixMul, MatrixInv and Gradient) a better reduction of 35% is reached in average.

Table 3 shows the same static interlock analysis for a hypothetical machine with shorter pipelines. We have assumed a one-cycle delay for memory accesses, floating-point addition and subtraction, and a two-cycle delay for floating-point multiplication and division. The static reduction of 45% is much better, however not optimal.

For the purpose of estimating the compile-time behavior when enabling or disabling the instruction placer, we have modified the code generation phase such that after building the syntax tree, the code generator is activated 100 times for every program, thereby not writing out the object file. The measurements did not indicate a significant slow-down of the compilation process due to scheduling (compile-time differences less than 5%). When compiling a single huge basic block, doubling the length of the basic block raised the compile-time by only little more than a factor of two. The second pass for target code generation is a simple sweep and is of complexity O(n). By that, a linear run-time of the instruction scheduling algorithm can be expected for practical cases.

The absolute compile times range between 0.54s (Fibonacci) and 0.88s (MatrixInv) on a Ceres-2 workstation ([Hee88], NS32032 CPU, 25MHz clock; more than 10 times slower then the Cray Y-MP in scalar mode). The absolute compile times for cf77 range between 0.21s user time (Fractal) and 1.17s (!) (MatrixInv) on the Cray Y-MP. Although the two compilers are not directly comparable, this indicates that our compiler would translate corresponding programs by at least an order of magnitude faster than cf77 on the Cray.

6 Conclusions

We have described an instruction scheduling heuristic for pipelined computer architectures. The algorithm may be implemented as an extension of already existing optimizations like common subexpression elimination. While speed gains produced by our method are comparable to other techniques, our method seems to be much simpler; e.g. no complicated data-structures like dependency DAGs need be maintained. An implementation of the scheduler within a cross-compiler for the Cray Y-MP proved to be fast and small (including a common subexpression elimination algorithm about 400 lines of around 4000 lines for the complete compiler). Because no form of look-ahead during scheduling is necessary, the method is adequate even for one-pass compilers without introducing much complexity.

We have not analyzed how the method is related to more sophisticated scheduling techniques (e.g. Circular Scheduling). Another issue we have not investigated is the extension of the scheduling scheme across basic block boundaries. However, simple extensions towards this goal seem to be relatively straight-forward: e.g. instructions after a conditional jump may be placed before the jump as long as no side-effects occur (and as long as there are free instruction slots).

This would not increase the run-time when the jump is taken, but decrease it when the jump is not taken.

There is always a tradeoff between reaching very high and "reasonable good" performance. The former often causes long compile times, the latter may cost expensive execution time. For machines like a Cray one obviously has decided to pay nearly everything for fast code and it remains questionable if a "reasonably good" but not "optimal" scheduling method might be acceptable. However, the small effort for our instruction scheduling algorithm may improve simple compilers on more conventional architectures considerably without introducing much complexity.

Acknowledgements

We would like to thank H. Mössenböck for his valuable comments and C. Szyperski for reading earlier versions of this paper.

References

[Aus82] Auslander, M., Hopkins, M., "An Overview of the PL.8 Compiler", *Proc. ACM SIGPLAN '82 Symposium on Compiler Construction*, June 1982.

[Bra91] Bradlee, D.G., Eggers, S.J., Henry, R.R., "Integrating Register Allocation and Instruction Scheduling for RISCs", *Proc. ACM Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[Cra88] *CRAY Y-MP Computer Systems Functional Description Manual*, Cray Research Inc., Publication HR-4001, 1988.

[Cre90] Crelier, R., *OP2: A Portable Oberon Compiler*, Computersysteme ETH Zürich, Tech. Report No. 125, Feb. 1990.

[Dav84] Davidson, J.W., Fraser, C.W., "Register Allocation and Exhaustive Peephole Optimization", *Software - Practice and Experience*, (14) 9, Sept. 1984.

[Dav86] Davidson, J.W., "A Retargetable Instruction Reorganizer", *Proc. ACM SIGPLAN '86 Symposium on Compiler Construction*, June 1986.

[Dec92] *Preliminary Alpha Architecture Handbook* (Special Announcement Edition), Digital Equipment Corporation, Feb. 1992.

[Don79] Dongarra, J.J., Hinds, A.R., "Unrolling Loops in FORTRAN", *Software - Practice and Experience*, (9) 3, March 1979.

[Fre74] Freiburghouse, R.A., "Register Allocation Via Usage Counts", *Communications of the ACM*, (17) 11, Nov. 1974.

[Gib86] Gibbons, P.B., Muchnick, S.S., "Efficient Instruction Scheduling for a Pipelined Architecture", *Proc. ACM SIGPLAN '86 Symposium on Compiler Construction*, June 1986.

[Gro83] Gross, T., *Code Optimization of Pipeline Constraints*, Tech. Report No. 83-255, Computer Systems Lab, Stanford Univ., Dec. 1983.

[Hee88] Heeb, B., *Design of the Processor-Board for the Ceres-2 Workstation*, Computersysteme ETH Zürich, Tech. Report No. 93, Nov. 1988.

[Hen83] Hennessy, J., Gross, T., "Postpass Code Optimization of Pipeline Constraints", *ACM Transactions on Programming Languages and Systems*, (5) 3, July 1983.

[Kan87] Kane, G., *MIPS R2000 RISC Architecture*, Prentice Hall, 1987.

[Rob89] Robbins, K.A., Robbins, S., *The Cray X-MP/Model 24 - A Case Study in Pipelined Architecture and Vector Processing*, Lecture Notes in Computer Science No. 374, Springer 1989.

[Sun87] *The SPARC Architecture Manual*, Sun Microsystems Inc., 1987.

[Wir88] Wirth, N., "The Programming Language Oberon", *Software - Practice and Experience*, (18) 7, July 1988 and Computersysteme ETH Zürich, Tech. Report No. 143, Nov. 1990.