Global Instruction Scheduling — a Practical Approach

Sebastian Schmidt

Computer Science Department
Friedrich-Schiller-University, D-07740 Jena, Germany
sc@iaxp01.inf.uni-jena.de

Abstract. The subject of this paper is the design and implementation of a framework, implementing a method for global instruction scheduling. It is based on the Program Dependence Graph as a central data structure. In contrast to other global scheduling methods, like Trace Scheduling, it does not rely on regular structures in the program. It has therefore the potential to be useful for control intensive programs, too. First results of the exploration of this method are presented and ideas for enhancement are derived from the experience.

1 Introduction

In the last years extensions of the Reduced Instruction Set Computer (RISC) architecture have evolved to enhance performance by increasing parallel processing. Typical examples of such architectures are superscalar [9], superpipelined and Very Long Instruction Word (VLIW)[3] processors.

For these machines with their increased parallelism instruction scheduling at the basic block level is no longer sufficient to utilize machine resources[4]. Instructions have to be moved across basic block boundaries. Beside other approaches like *Trace scheduling*[2] or *Enhanced percolation scheduling*[5], Bernstein and Rodeh[1] proposed a method which uses the Program Dependence Graph (PDG) to perform global instruction scheduling.

2 Design and Implementation of the Scheduler

The GNU C compiler GCC[6] was chosen as starting point for our experiments. It implements traditional optimizations like common subexpression elimination and loop optimizations as well as local instruction scheduling [7][8]. To achieve portability all informations needed by the code generator and machine specific optimizations are encapsulated in machine description files.

The global scheduler was split into the following modules:

1. Control flow analysis and construction of the CSPDG.
2. Data dependence analysis.
3. Instruction Scheduling.
4. Instrumentation and Configuration.

Global scheduling is done by the same algorithm which is used for the local scheduler. Since all restrictions imposed by global data dependencies are considered by data dependence analysis, there are only minor changes in the implementation of this algorithm. Instead of a single basic block it works on linear sequences of basic blocks now.

While useful scheduling is always done on the longest possible sequence of equivalent blocks, we currently experiment with three different variants of speculative scheduling:

1. The sequences are formed by pairs of basic blocks for which speculative scheduling is allowed according to the PDG.
2. The set of pairs from the first variant is restricted by using branch prediction. Only if the probability of a branch to be taken is higher than a given threshold, the corresponding edge in the PDG is selected for speculative scheduling. Currently this probability information is achieved by branch profiling.
3. Longer sequences of basic blocks are formed by branches with high probability.

There are some extensions made to the compiler to gather additional information at compile and runtime of the program:

- Branch profiling.
- A simple way to measure Instruction Level Parallelism.

To get some flexibility while experimenting with different scheduling variants, the scheduler reads a file that allows to configure it's behavior at compiler runtime.

3 Results

The following numbers are preliminary results of first tests for three different variants of global scheduling. Since the work concentrates on control intensive integer programs, only this type of programs was tested. `qsort`, `bubsort`, `minmax`, `knight` are short programs implementing simple algorithms. `GCC` (C compiler), `gofer` (interpreter for a functional language) and `gzip` (data compression) are real world applications. They were run on a DEC 3000 model 600 (processor alpha 21064 / 175 MHz). All other optimizations except loop unrolling were turned on (switch `-O2`). Table 1 shows the improvement in runtime for useful scheduling only and for speculative scheduling according to the first two variants as mentioned in section 2.

The bad results for `qsort` may result from the recursive nature of this program. Obtaining better results in this case would require transformations on an interprocedural level. These results match with the results published in [1].

Our current effort is to find out why the scheduler fails at large programs like gcc and gzip. To do so, they were analyzed with a profiling tool, to locate the places in the code where they spend most of their time. For this purpose we included some additional control intensive tools in our test suite: Bison (a parser

program	improvement in %		
	useful	speculative	
		variant 1	variant 2
qsort	-0.2	-0.8	0.0
bubsort	7.0	15.1	15.1
minmax	4.1	4.0	4.0
knight	-0.7	2.3	2.6
gcc	-1.7	-0.5	-0.1
gofer	4.9	6.8	7.9
gzip	0.2	-0.1	3.5

Table 1. Improvement in Runtime

generator), grep (searching a file for regular expressions) and diff (compare two files for differences).

With one notable exception it was found that the "hot spots" in all programs were while-loops with small bodies (only few source code instructions). The only exception mentioned above was the gofer system which spent most of the time in two functions: the intermediate code interpreter and the expression evaluator. Both of them consist of loops with large bodies and allow to form big regions.

Therefore the next task will be to collect these loops in a benchmark suite similar to the Livermore Loops for numeric programs. This way it will be easier to study the effects of different optimization variants on the sample programs.

4 Future Work

Loop handling: Since the scheduler fails on the input code mentioned above (while-loops with small bodies) the enhancements planned for the near future will concentrate on methods to handle this kind of loops.

The simplest way to enlarge the number of basic blocks available to the scheduler is *loop unrolling*, i.e. to multiply the code of the loop body. While unrolling for-loops may lead to larger basic blocks, unrolling while-loops only increases the number of basic blocks in the loop body. But this is exactly what we want.

Branch Prediction: One crucial thing when doing speculative scheduling is to find the "right" branch, i.e., the branch that is taken with the greatest probability. The easiest way to find this information is to use profile information from sample runs of the input program. This is the way we currently follow. Another way is to do static branch prediction at compile time using heuristics.

Other Enhancements:

- The heuristics currently used by the list scheduler are designed for local scheduling. It should give better results if at least their parameters are tuned.
- The current implementation uses useful and speculative scheduling only. With the information provided by the PDG it should also be possible to perform code duplication.

5 Conclusion

This paper describes design and implementation of a method for global instruction scheduling for control intensive programs. It is centered on the Program Dependence Graph (PDG) as the central data structure. The implementation is based on the GNU C compiler (GCC) and has been tested on a machine with the superscalar Alpha Architecture. First experiences, using small algorithms as well as large real world programs as samples, are presented. The results show that it is possible to improve the performance even of control intensive programs, using global instruction scheduling. They promise the possibility for further enhancements of this method. The reason why this optimization fails for some programs, is it's lack of ability to handle while loops with small bodies. Possibilities to overcome these disadvantages are shown and will be the subject of future work.

References

1. D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. *SIGPLAN Notices*, 26(6):241–255, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*
2. J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.
3. J.A. Fisher. The VLIW machine: A multiprocessor for compiling scientific code. *IEEE Computer*, pages 45–53, July 1984.
4. N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *SIGPLAN Notices*, 24(5):272–282, April 1989.
5. T. Nakatani K. Ebcioglu. A new compilation technique for parallelizing regions with unpredictable branches on a VLIW architecture. In *Proc. of the Workshop on Languages and Compilers for Parallel Computing*, August 1989.
6. R. M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Cambridge MA, 02139, June 1989.
7. M. D. Tiemann. The GNU instruction scheduler. course report CS343, Stanford University, June 1989.
8. A. Unger, S. Schmidt, and E. Zehendner. Anordnung von Instruktionen. Berichte zur Rechnerarchitektur Vol. 2, No. 4, Friedrich-Schiller-University Jena, 1996.
9. T. Ungerer. *Mikroprozessortechnik: Architektur und Funktionsweise superskalarer Mikroprozessoren.* Number ISBN 3-8266-0130 in Thomson's aktuelle Tutorien. Internat. Thomson Publ., Bonn, 1995.