CIS 872 Optimizing and Parallelizing Compilers **Fall 1995**

Course Information

Course:

Optimizing and Parallelizing Compilers.
MW 12:20-1:35 (3 hours) Smith 102A

Prerequisite: CIS 672: *Compiler Construction.*

Instructor:

Lori Pollock pollock@cis.udel.edu
406 Smith Hall (831-1953)

Office Hours:

Monday 2:00-3:00 PM
Thursday 1:00-2:00 PM
By appointment.

Course Objectives:

This course is intended to give the student a thorough understanding of the issues involved in building a highly optimizing compiler for various architectures. The course concentrates on sophisticated machine-independent static program analyses, optimizing program transformations, and advanced compiler techniques for improving the effective utilization of scalar, vector, and parallel machines.

Course Materials:

Recent journal and conference articles. Excerpts from (yet to be published) book by Kennedy and Allen (KA).

Recommended Supplemental Textbook:

Aho, Sethi, and Ullman, *Compilers, Principles, Techniques, and Tools*, Addison Wesley, 1986 (ASU).

Requirements and Grading:

- (15%) "The SUIF Experimental Compiler" exercise (due Friday, Sept 29).
- (30%) Project (due Dec 15).
- (25%) Written Summary/Discussion Report.
- (30%) Oral presentations.

Tentative Course Schedule (intended number of lectures in parentheses):

1. INTRODUCTION.

(1) Overview of parallel architectures, issues in compiling for these machines.

References: ASU-10, KA-1.

2. CLASSIC COMPILER OPTIMIZATION.

(2) CONTROL FLOW ANALYSIS: finding basic blocks, building the control flow graph, terminology of flow graphs, natural loops, strongly connected regions, identifying loops, dominator tree construction, finding back edges, the node stacking algorithm for finding loops, reducibility, detecting loops in reducible flow graphs, classes of edges, interconnectedness and nesting depth, control dependence

References: ASU-9,10, KA-7.

(3) INTRAPROCEDURAL DATA FLOW ANALYSIS: notions of safety of transformations and conservativeness in analysis, example data flow problems: (reaching defs, live vars, available expressions, constant propagation), forward vs backward, rep of sets, monotone data flow analysis frameworks, distributive vs nondistributive, overview of methods, iterative method, depth first order, notion of rapid problems, interval analysis, Allen/Cocke method, intuitive idea behind other techniques: T1/T2, Graham/Wegman (notion of fast problem), Tarjan, hybrid

References: ASU-10, KA-4.

Cytron, Ferrante, Rosen, Wegman, Zadeck, "Efficiently computing static single assignment form and the control dependence graph," TOPLAS Oct 1991.

(2) CLASSIC OPTIMIZING TRANSFORMATIONS: transformations, safety, profitability, how data flow analysis is used, constant folding, global and local common subexpression elimination, loop invariant code motion, code hoisting

References: ASU-10, KA-4.

Wegman and Zadeck, "Constant Propagation with Conditional Branches", TOPLAS April 1991.

Knoop, Oliver Ruthing, Steffen, "Lazy Code Motion," PLDI 1992.

Wolfe, "Beyond Induction Variables," PLDI 1992.

Knoop, Ruthing, and Steffen, "Partial Dead Code Elimination," PLDI 94.

Briggs and Cooper, "Effective Partial Redundancy Elimination," PLDI 94.

Click, "Global code motion/global value numbering," PLDI 95.

Hanxledon and Kennedy, "Give-N-Take - A Balanced Code Placement Framework," PLDI 94.

3. CODE GENERATION FOR SCALAR MACHINES.
(2) tree pattern matching, Graham-Glanville code generator, Davidson's retargetable instruction reorganizer, the BURS code generator generator.

References: ASU-9.

Glanville and Graham, "A New Method for Compiler Code Generation," POPL 1978.

Davidson, "A Retargetable Instruction Reorganizer," PLDI 1986.

Fraser and Wendt, "Automatic Generation of Fast Optimizing Code Generators," PLDI 1988.

Benitez and Davidson, "Target-specific Global Code Improvement: Principles and Applications," unpublished manuscript.

Proebsting, "BURS Automata Generation," TOPLAS, May 1995.

4. COMPILER MEMORY HIERARCHY MANAGEMENT.
(2) REGISTER ALLOCATION: local (Sethi/Ullman tree code generation, Aho/Johnson dynamic programming), graph coloring methods (Chaitin, Chow, Briggs, register windows), overview of others: clique separators, hierarchical, probabilistic reg alloc(pldi1992), load-store range analysis (pldi1993), Hendren's framework, interactions with scheduling.

References: KA-8.

Chaitin, Auslander, Chandra, Cocke,, Hopkins, Markstein, "Register Allocation via Coloring," Computer Languages 1981.

Chow and Hennessy, "The Priority-based Coloring approach to register allocation", TOPLAS, October 1990.

Briggs, Cooper, Torczon, "Rematerialization," PLDI 1992.

Hendren, Gao, Altman, Mukerji, "A register allocation framework based on hierarchical acyclic interval graphs," International Conference on Compiler Construction, 1992.

Gupta, Soffa, Ombres, "Efficient Register Allocation via Coloring Using Clique Separators," TOPLAS, May 1994.

Kurlander and Fischer, "Zero-Cost Range Splitting," PLDI 94.

Proebsting and Fischer, "Probabilistic Register Allocation," PLDI 1992.

Norris and Pollock, "Register Allocation over the Program Dependence Graph," PLDI 1994.

Burger, Waddell, Dybvig, "Register Allocation Using Lazy saves, Eager Restores, and Greedy Shuffling," PLDI 1995.

(2) CACHE MANAGEMENT:

References: KA-8.

D. Callahan, K. Kennedy and A. Porterfield, ‘‘Software prefetching," *SIGPLAN Notices, 26(4):40-52. Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* Santa Clara, CA, April 1991.

E. Gornish, E. Granston and A. Veidenbaum, ‘‘Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *Proceedings of the 1990 International Conference on Supercomputing,* 1990.

T. C. Mowry, M. S. Lam and A. Gupta, ‘‘Design and evaluation of a compiler algorithm for prefetching," *SIGPLAN Notices,* 27(9):62-75 September 1992, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.*

D. Callahan, J. Cocke, and K. Kennedy, ‘‘Estimating interlock and improving balance for pipelined machines," *Journal of Parallel and Distributed Computing,* 5:334-358, 1988.

S. McFarling, ‘‘Program optimization for instruction caches," *Third International Conference on Architectural Support for Programming Languages and Operating Systems,* Boston, pp 183-191, April, 1989.

D. Gannon, W. Jalby and K. Gallivan, ‘‘Strategies for cache and local memory management by global program transformations," in *Proceedings of the First International Conference on Supercomputing,* Springer-Verlag, Athens, Greece, 1987.

Carr, McKinley, and Tseng, "Compiler Optimizations for Improving Data Locality," ASPLOS 1994.

M.S. Lam, EE. Rothberg, and M.E. Wolfe, ‘‘The cache performance and optimization of blocked algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1991.

M.E. Wolf and M.S. Lam, ‘‘A data locality optimizing algorithm," in *Proceedings of the SIGPLAN ’91 Conference on Programming Language Design and Implementation,* June 1991.

M. Wolfe, ‘‘Iteration space tiling for memory hierarchies," in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

McCalpin and Smotherman, "Automatic Benchmark Generation for Cache Optimization of Matrix Operations," ACM SE Conference, 1995.

4. COMPILING FOR PARALLEL MACHINES.

(2) DATA DEPENDENCE ANALYSIS: dependence tests, dependence graphs (separability test, gcd test, Banerjee test, practical testing, omega test)

References: KA-2-3.

Ferrante, Ottenstein, Warren," The PDG and its Use in Optimization, TOPLAS July 1 987.

(1) PRELIMINARY TRANSFORMATIONS

References: KA-4.

(2) VECTOR CODE GENERATION: code generation algorithm, loop interchange, recurrence breaking transformations to increase opportunity for vectorization

References: KA-5.

(2) PARALLEL CODE GENERATION: parallel code generation algorithm, transformations to increase parallelization.

References: KA-6.

Bacon, Graham, Sharp, "Compiler Transformations for High-Performance Computing," ACM Computing Surveys, 1994, volume 26, No. 4.

(2) SCHEDULING: list scheduling, software pipelining.

References:

Gibbons and Muchnick, "Efficient instruction scheduling for a pipelined architecture," PLDI 1986.

Lam, "Software pipelining: An effective scheduling technique for VLIW machines," PLDI 88.

Lo and Eggers, "Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level parallelism," PLDI 1995.

5. MORE EXACT COMPILER ANALYSIS.

(2) ALIAS ANALYSIS: pointer alias analysis, subprogram-related analysis

Cooper, "Analyzing Aliases of Reference Formal Parameters," POPL 1985

Cooper and Kennedy,"Fast Interprocedural Alias Analysis," POPL 1989.

Ruf, "Context-insensitive Alias Analysis Revisited," PLDI 1995.

Landi and Ryder, "Pointer-induced aliasing: a problem classification," POPL 1991.

Wilson and Lam, "Efficient Context-sensitive Pointer Analysis for C Programs," PLDI 1995.

(2) INTERPROCEDURAL ANALYSIS: call graph construction, flow sensitive vs flow insensitive, aliases, interproc constant propagation, may-mod and may-use information, procedure linkage tailoring

Hall and Kennedy,"Efficient Call Graph Analysis," LOPLAS 1992.

Callahan, Cooper, Kennedy, Torczon, "Interprocedural Constant Propagation," SIGPLAN 1984.

Cooper, Kennedy, "Efficient Computation of Flow Insensitive Interprocedural Summary Information," SIGPLAN 1984.

Cooper and Kennedy, "Interprocedural Side-Effect Analysis in Linear Time," PLDI 1988.

Details on Course Requirements:

1. "The SUIF Experimental Compiler" exercise (due Friday, Sept 29):
This assignment consists of familiarizing yourself with the "use" of SUIF, the Stanford Experimental Compiler for optimizing and parallelizing research and education. Your task is do either of the following:

- Run SUIF with different optimizations turned on and off, using various C and Fortran benchmark programs, focusing on the loop-level parallelism enhancing optimizations and locality optimizations. Compare your results with the code generated without running through SUIF, and create a chart and short report discussing your findings. You can run your final code on the University's SGI 8 processor PowerChallenge. (gershwin.udel.edu)

- Run SUIF with different optimizations turned on and off, using various C and Fortran benchmark programs, focusing on the MIPS optimizer backend. Compare your results with the gcc generated code, and create a chart and short report discussing your findings. You should run your final codes on one processor of the SGI PowerChallenge or one of the CIS SGI machines.

To do this assignment, you need to start by reading the documentation on auriga in the /m/auriga/suif-1.0.1 directory structure.

2. Project (due Dec 15):

The focus of the projects is to study the interactions of various program transformations and/or their relative effects on different architectures. This can be done in a number of ways. To study the effect of a given transformation on a given architecture, you can run different applications on the same machine, using the provided compiler and turn optimization switches on and off. For source-to-source transformations, you can manually perform the transformations on the source code before running the compiler. However, you need to carefully determine when it appears to be useful to perform these transformations for a given application.

We have several different architectures available to us here: (1) the 8 processor SGI Powerchallenge, which has a source-to-source preprocessor and a Fortran compiler with software pipelining. It also has a C compiler that does some parallelizing and optimizing. (2) the 8 vector processor Cray J90 with the Fortran 77 and Fortran 90 compilers, and Fortran preprocessors. (3) the SUIF experimental optimizing and parallelizing compiler which generates either C code or goes down to MIPS code (can be run on an SGI machine). (4) the IBM RS6000 990's with optimizing Fortran 90 and Fortran 77 compilers.

A variety of Fortran and C application codes are available, from the SPEC benchmarks to small kernel codes.

Projects can be designed to examine the effect of locality improving optimizations on the various architectures, software pipelining versus hardware register renaming, scalar optimizations, parallelizing enhancing optimizations, prefetching, etc. The goal of the project is to design an experiment that will enable you to make some interesting remarks about some aspect of program transformations, in terms of their interactions with each other, and/or their varying effects in different architectures. You should discuss your planned experiment with the instructor prior to jumping too far into your project. You might want to work a little on the project to decide what direction you want to go first. The project will culminate in a written report of your experiment and results, and a short 15 minute presentation of your project to the class during the final examination hour.

3. Oral Presentations:

Each student will choose a *topic* from the list below which they will *teach* the class. The *teaching* will consist of 2 full class lectures covering the topic. The oral presentations should be sure to include the following:

1. a clear description of the problem being addressed
2. a breakdown of the specific issues to be considered in approaching the problem
3. an overview of the different approaches to the problem, focusing on the most well known approaches and recent new approaches, presented in historical order
4. a critique/assessment of the approaches considering their limitations, how well they succeed in practice,...
5. what issues remain to be considered, and the focus of current and future research in this area

I have already listed the major papers in each topic, so there is no need to go searching through the literature for more papers. You should have plenty of material to fill 2 lectures.

Topics should be chosen within the first 2 weeks of the course, from the list below. The topics will be assigned on a first-come, first-serve basis. You should notify me of your choice by the end of the second full week of classes.

- code generation for scalar machines
- register allocation
- cache management
- scheduling
- alias analysis
- interprocedural analysis

If there are more students, then you can also choose from:

- vector code generation
- parallel code generation
- interprocedural analysis

It is strongly suggested that you use overhead slides for your oral presentation. You are required to hand in a copy of your slides within a week after your presentation, to the instructor, and each student in the class.

4. Written Summary/Discussion Report:

Each student will choose 1 topic from the syllabus (with the exception of control and data flow analysis and the one topic that you have presented to the class). For this topic, you should write a report that accomplishes the same as the oral presentations. That is, the report should resemble the BACKGROUND, LITERATURE REVIEW, and CRITIQUE of the area as would be included in the Related Work section of a journal or conference paper. It should contain the proper citations and a bibliography. However, there is no need to read any more papers than those listed on the syllabus for the topic. You should hand in your report within 2 weeks after the oral presentations for that topic have been completed. The report should be typed single space, 10 point, no longer than 10 pages. You will be graded on not only your coverage of the area, but also your critical assessment of work in the area.