

COMP 512



This is COMP 512 — “Advanced Compiler Construction”

- **Subject Matter**
 - > **Compiler-based code improvement techniques**
 - Sometimes called “optimization”
 - > **Analysis required to support them**
 - > **No vector or multiprocessor parallelism**
 - See COMP 515, taught by Ken Kennedy
- **Required Work**
 - > **Mid-term (25%), Final (25%), & Projects (50%)**
 - > **Details of project will depend on class size**

Notice: Any student with a disability requiring accommodations in this class is encouraged to contact me after class or during office hours. Students should also contact Rice’s Coordinator for Disabled Student Services



What about the book?

- **We will use many different sources**
 - > **Chapters 8, 9, & 10 of “Engineering a Compiler” + appendices**
 - > **“Compiler-based Code Improvement Techniques”**
(Cooper, McKinley, & Torczon)
 - > **The original papers**

Expect to read a lot for this class
- **Slides from lecture will be available on the web site**
 - > **<http://www.cs.rice.edu/~keith/512>**
 - > **I will try to post them before class**

Your part is to read the material *before* coming to class

COMP 512



My goals

- Convey a fundamental understanding of the current state-of-the-art in code optimization and code generation
- Develop a mental framework for approaching these techniques
- Differentiate between the past & the present
- Motivate current research areas (*and expose dead problems*)

Explicit non-goals

- Cover every transformation in the “catalog”
- Teach every data-flow analysis algorithm
- Cover issues related to multiprocessor parallelism



COMP 512



Rough syllabus

- Introduction to optimization
 - > Motivation & history
 - > An example compiler (Fortran H)
 - > Redundancy elimination as an example (Chapter 8, EaC)
- Data-flow analysis (Chapter 9, EaC)
 - > Iterative algorithm
 - > SSA construction
- Classical scalar optimization (Chapter 10 & CMT)
 - > Taxonomy for transformations
 - > Populate the taxonomy (papers)
- Combining optimizations (papers)
- Analyzing and improving whole programs (papers)

COMP 512



For next class read

R.G. Scarborough and H.G. Kolsky, “Improved Optimization of FORTRAN Object Programs”, *IBM Journal of Research and Development*, November, 1980, pages 660-676.



How does optimization change the program?

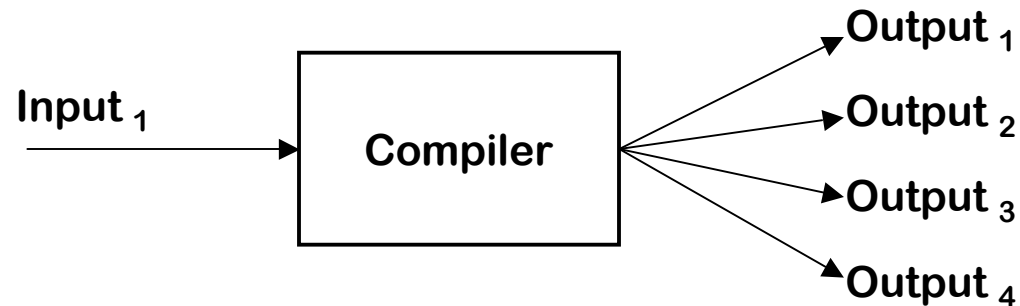


Optimizer tries to

1. Eliminate overhead from language abstractions
2. Map source program onto hardware efficiently
 - > Hide hardware weaknesses, utilize hardware strengths
3. Equal the efficiency of a good assembly programmer



What does optimization do?

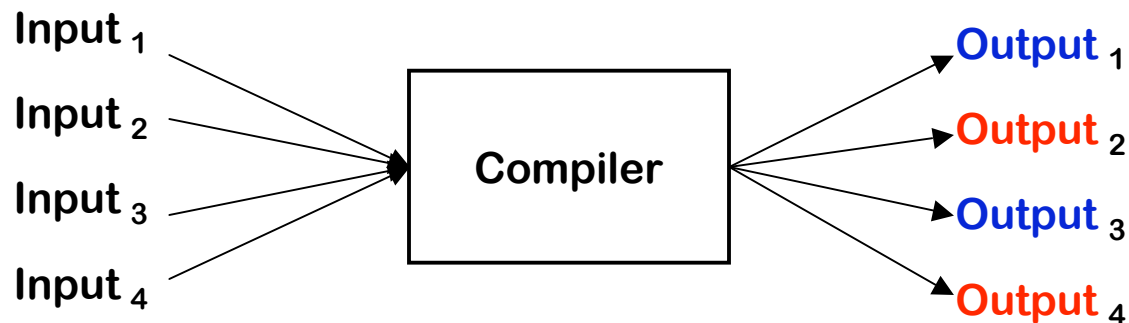


- The compiler can produce many outputs for a given input
 - > The user might want the fastest code
 - > The user might want the smallest code
 - > The user might want the code that pages least
 - > The user might want the code that ...
- Optimization tries to reshape the code to better fit the user's goal

COMP 512



- Some inputs have always produced good code
 - > First Fortran compiler focused on loops
 - > PCC did well on assembly-like programs



- The compiler should provide robust optimization
 - > Small changes in the input should not produce wild changes in the output
 - > Create (& fulfill) an expectation of excellent code quality
 - > Broaden the set of inputs that produce good code
- Routinely attain large fraction of peak performance (not 5%)

COMP 512



Good optimizing compilers are crafted, not assembled

- **Consistent philosophy**
- **Careful selection of transformations**
- **Thorough application of those transformations**
- **Careful use of algorithms and data structures**
- **Attention to the output code**

Compilers are engineered objects

- **Try to minimize running time of compiled code**
- **Try to minimize compile time**
- **Try to limit use of compile-time space**
- **With all these constraints, results are sometimes unexpected**



A quick look at real compilers

Consider inline substitution

- Replace procedure call with body of called procedure
 - > Rename to handle naming issues
 - > Widely used in optimizing OOPs
- How well do compilers handle inlined code?

Characteristics of inline substitution

- **Safety:** almost always safe
- **Profitability:** expect improvement from avoiding the overhead of a procedure call and from specialization of the code
- **Opportunity:** inline leaf procedures, procedures called once, others where specialization seems likely

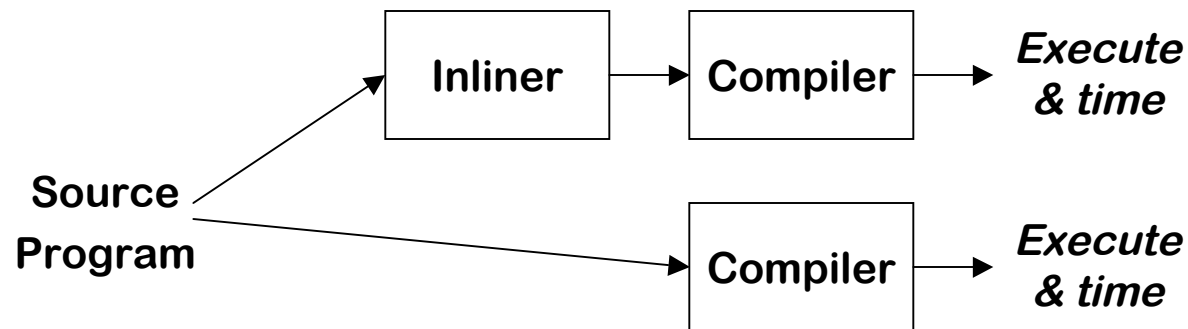


A quick look at real compilers

The study

Five good compilers!

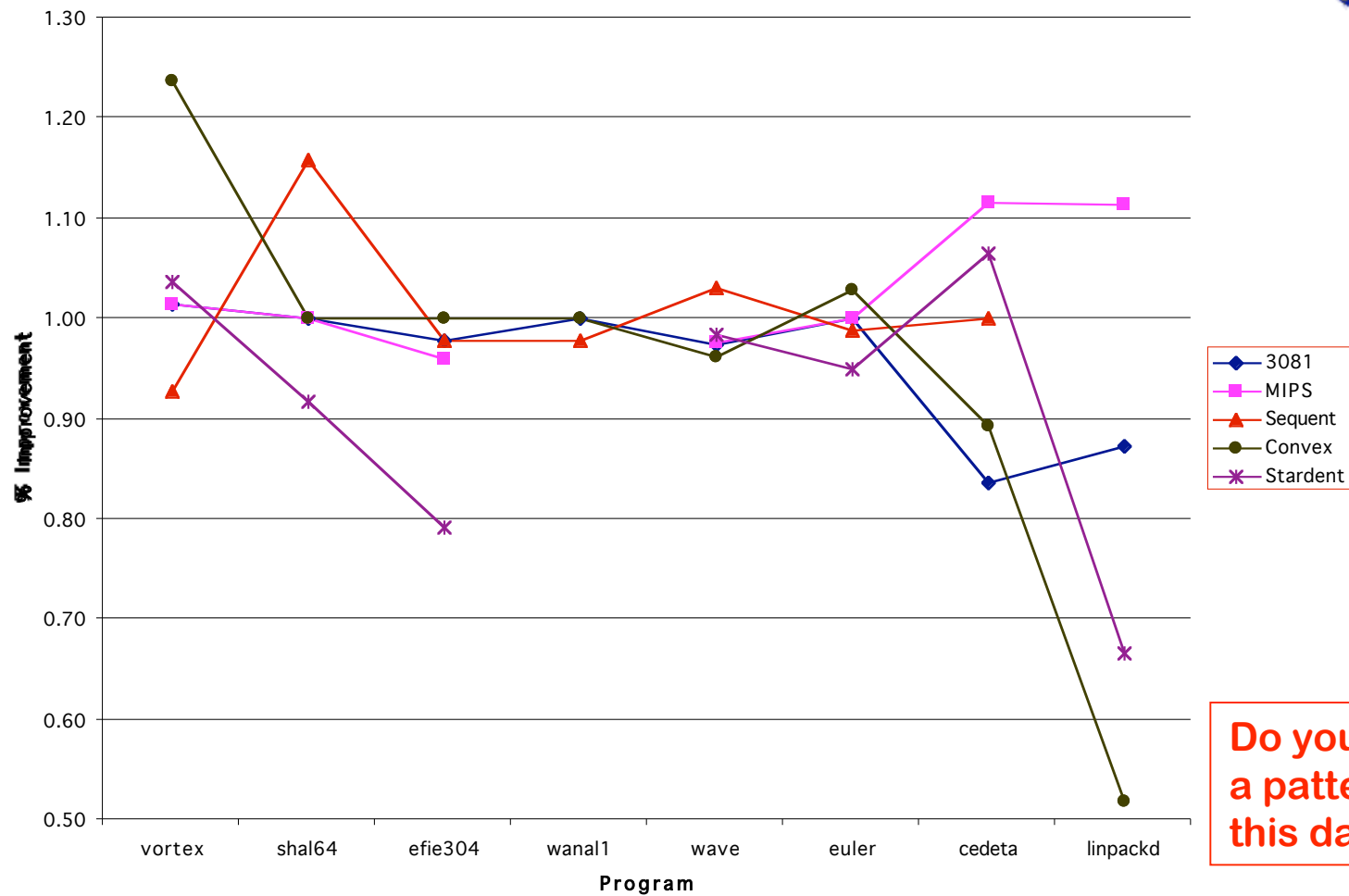
- Eight programs, five compilers, five processors
- Eliminated over 99% of dynamic calls in 5 of programs
- Measured speed of original versus transformed code



Experimental Setup

- We expected uniform speed up, at least from call overhead
- What really happened?

A quick look at real compilers



Do you see a pattern in this data?



A quick look at real compilers

And this happened with good compilers!

What happened?

- **Input code violated assumptions made by compiler writers**
 - > **Longer procedures**
 - > **More names**
 - > **Different code shapes**
- **Exacerbated problems that are unimportant on “normal” code**
 - > **Imprecise analysis**
 - > **Algorithms that scale poorly**
 - > **Tradeoffs between global and local speed**
 - > **Limitations in the implementations**

**The compiler writers were surprised
them)**

(most of



A quick look at real compilers

One standout story

- MIPS M120/5, 16 MB of memory
- Running standalone, *wanal1* took > 95 hours to compile
 - > Original code, not the transformed code
 - > 1,252 lines of Fortran (not a large program)
 - > COMP 512 met twice during the compilation
- Running standalone with 48 MB of memory, it took < 9 minutes
- The compiler swapped for over 95 hours !?!

- For several years, *wanal1* was a popular benchmark
 - > Compiler writers included it to show their compile times!

COMP 512



Intent of this class

- Theory & practice of scalar optimization
 - > The underpinning for all modern compilers
 - > Influences the practice of computer architecture
- Learn not only “what” but also “how” and “why”
- Provide a framework for thinking about compilation
- Class will emphasize transformations
- Analysis should be driven by needs of transformations

Remember
register
windows?

Role of the lab

- Critically important to provide hands-on experience
- Little time pressure

Disclaimer



Disclaimer:

The following slides contain a rough history of code optimization from 1955 to 2000. They are intended to convey to you my own impressions of what was happening in the field. They are quite subjective. They are quite incomplete. (Hundreds of papers were published during each five year period. I cannot, and did not, try to be comprehensive.) They are based on perusing conference proceedings for the various periods.

Events are listed when (in my perception) the subject came to the fore. In some cases, this is different than when the idea first appeared. For example, software pipelining was clearly invented by Glaser & Rau in 1981. That notwithstanding, the technique became widely known and understood in the latter half of the 1980's, which is why I cited the two PLDI 88 papers.

Again, this history is neither definitive or objective.

- Keith



A Sense of History

1955-1959	<i>Commercial compilers generated good code</i>
Fortran	Separation of concerns (Backus, 1956)
Cobol	Control-flow graph, register allocation (Haibt, 1957)
1960–1964	<i>Academics try to catch up with industrial trade secrets</i>
Algol 60	Early algorithms for “code generation” (1960, 1961)
	Relating theory to practice (Lavrov, 1962)
	Alpha project at Novosibirsk (Ershov, 1963 & 1965)
1965-1969	<i>Technology begins to spread</i>
PL/I	Fortran H (Medlock & Lowry, 1967)
Algol 68	Value numbering (Balke, 1967 ?)
Simula 67	Literature begins to emerge (Allen, 1969)

A Sense of History



1970-1974	<i>The literature explodes and optimization grows up</i>
SETL	Cocke & Schwartz, Allen-Cocke Catalog, 1971
Smalltalk	Theory of analysis (Kildall, 1971, Allen & Cocke, 1972)
Lisp	Interprocedural analysis (Spillman, 1972)
APL	Strength reduction, dead code elimination, Live (SETL) Expression tree algorithms (Sethi, Aho & Ullman)
1975-1979	<i>Global optimization comes of age</i>
Pascal	Full literature of data-flow analysis
CLU	Strength reduction (Cocke & Kennedy, 1977)
Alphard	Partial redundancy elimination (Morel & Renvoise, 1979)
Com. Lisp	Inline substitution studies (Scheiffler, 1977, Ball, 1979) Tail recursion elimination (Steele, 1978) Data dependence analysis (Bannerjee, 1979)

A Sense of History



1980-1984	<i>Programming environments and new architectures</i>
Smalltalk80	Incremental analysis (Reps, 1982; Ryder, Zadeck, 1983)
ADA	Incremental compilation (Schwartz <i>et al.</i> , 1984)
Scheme	Interprocedural analysis (Myers, 1981; Cooper, 1984)
	RISC compilers (PL.8, 1980; MIPS, 1983)
	Graph coloring allocation (Chaitin, 1981; Chow, 1983)
	Vectorization (Wolfe, 1982; R. Allen, 1983)
1985–1989	<i>Resurgence of interest in classical optimization</i>
C++	Constant propagation (Wegman & Zadeck, Torczon, 1985)
ML	Code motion of control structures (Cytron <i>et al.</i> , 1986)
Modula-3	Value numbering (Alpern <i>et al.</i> , Rosen <i>et al.</i> , 1988)
	Software pipelining (Lam, Aiken & Nicolau, 1988)
	Pointer analysis (Ruggeri, 1988)
	SSA-form (Cytron <i>et al.</i> , 1989)

A Sense of History



- 1990-1994** *Architects (and memory speed) drive the process*
Fortran 90 Hierarchical allocation (Koblenz & Callahan, 1991)
Scalar replacement (Carr 1991)
Cache blocking (Wolf, 1991)
Prefetch placement (Mowry, 1992)
Commercial interprocedural compilers (Convex, 1992)
- 1995–1999** *The internet age & SSA comes of age*
Java JIT compilers (Everyone, from 1996 to present)
Perl (?) Code compression (Franz, 1995; Frasier et al., 1997; ...)
SSA-based formulations of old methods (lots of papers)
Compile to VM (Java, 1995; Bernstein, 1998; ...)
Memory layout optimizations (Smith, 19??; others ...)
Widespread use of analysis (pointers, omega test, ...)

It's still too early for an epitaph !