COMP 512
Rice University
Spring 2015

Fortran H and PL.8

Papers about Great Optimizing Compilers

R.S. Scarborough and H.G. Kolsky, "Improved Optimization of FORTRAN Object Programs," IBM Journal of Research and Development 24(6), November 1980, pages 660-676

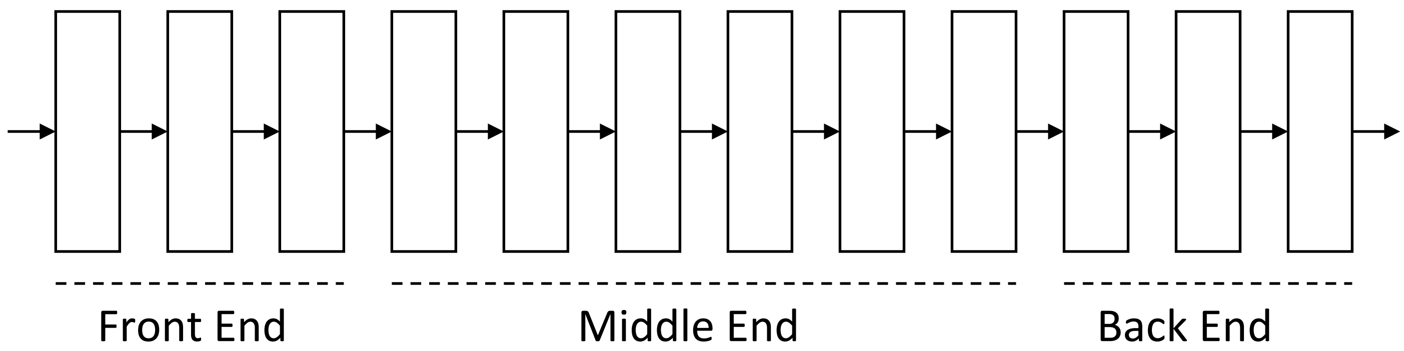
Marc Auslander & Martin Hopkins, "An Overview of the PL.8 Compiler", *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction*, SIGPLAN Notices 17(6), June 1982.

John Cocke & Peter Markstein, "Measurement of Program Improvement Algorithms", *Proceedings of Information Processing 80*, North Holland.

Citation numbers refer to entries in the EaC2e bibliography.

Classic Compilers

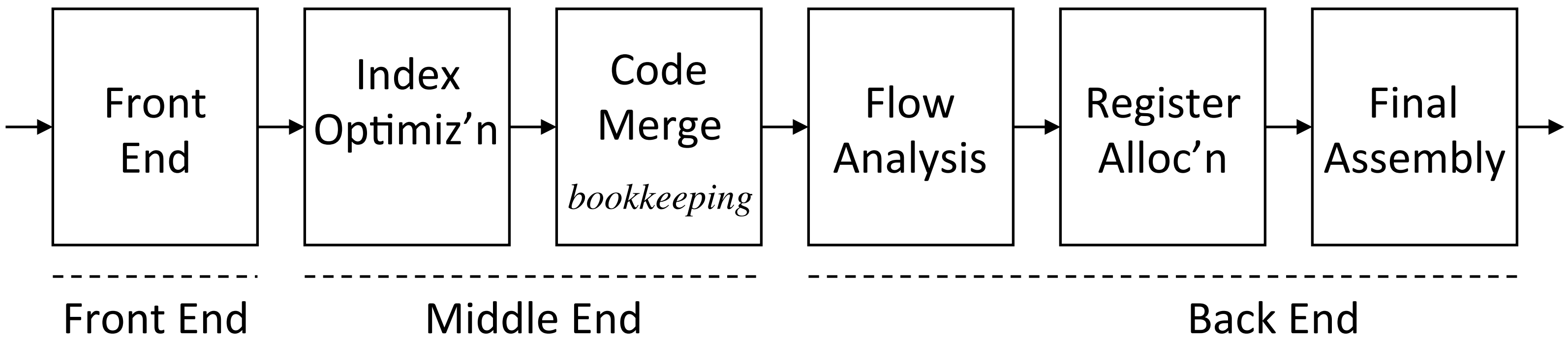
Compiler design has been largely fixed since 1960



- Front End, Middle End, & Back End
- Series of filter-style passes (number of passes varies)
- Fixed order for passes

Classic Compilers

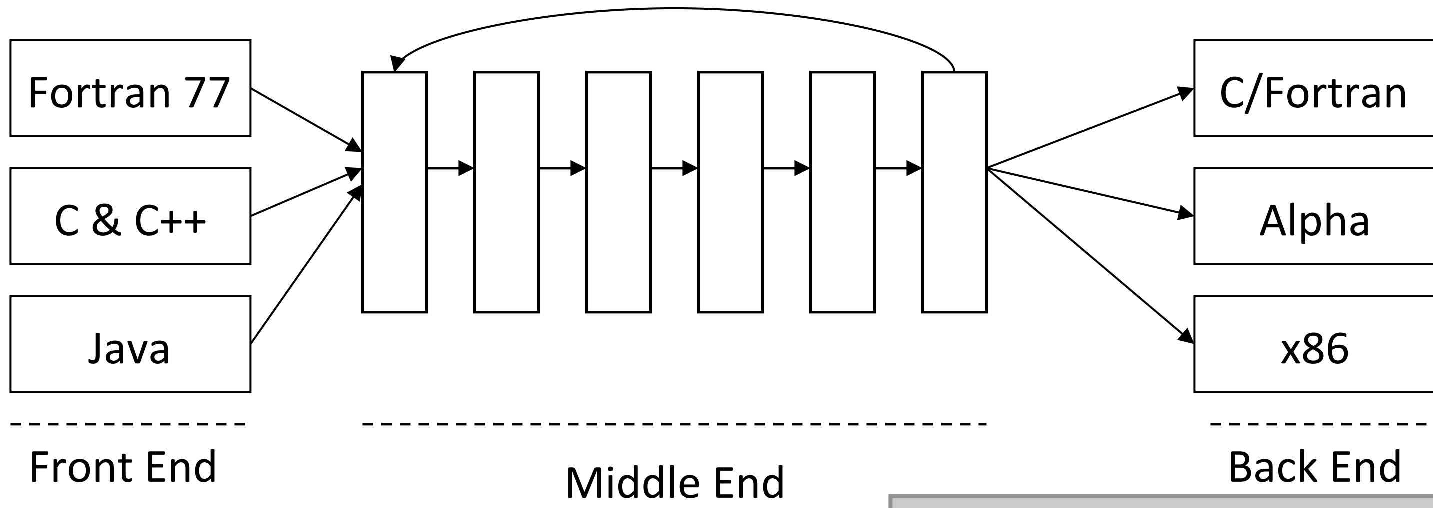
1957: The FORTRAN Automatic Coding System [26, 27]



- Six passes in a fixed order
- Generated good code
  - ♦ Assumed unlimited index registers
  - ♦ Code motion out of loops, with **if**s and **goto**s
  - ♦ Did flow analysis & register allocation

Classic Compilers

1999: The SUIF Compiler System (in the NCI)



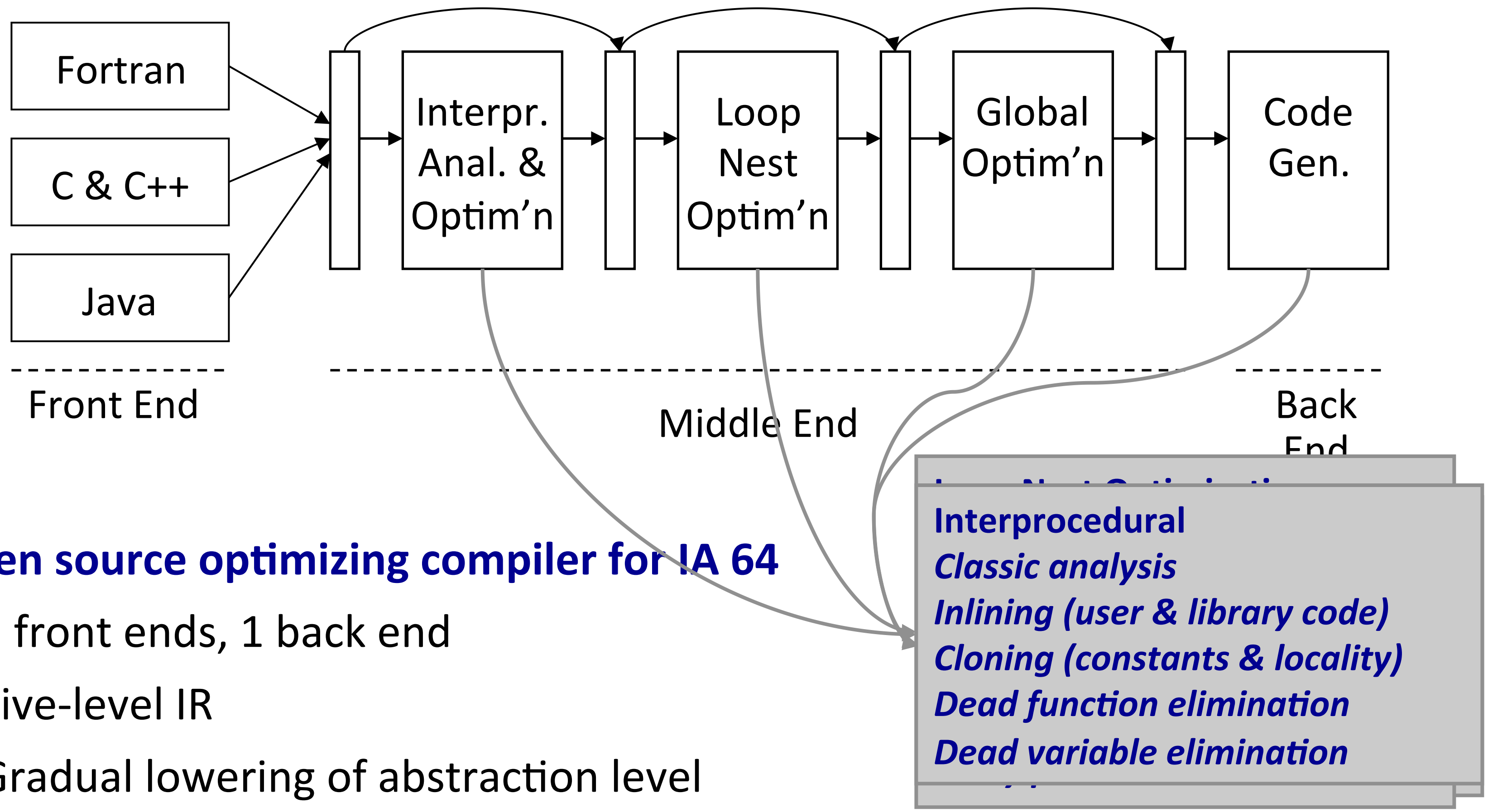
Academic research system (Stanford)

- 3 front ends, 3 back ends
- 18 passes, configurable order
- Two-level IR (High SUIF, Low SUIF)
- Intended as research infrastructure

Data dependence analysis
Scalar & array privitization
Reduction recognition
Pointer analysis
Affine loop transformations
Blocking
Capturing object definitions
Virtual function call elimination
Garbage collection

Classic Compilers

2000: The SGI Pro64 Compiler, now "Open 64"



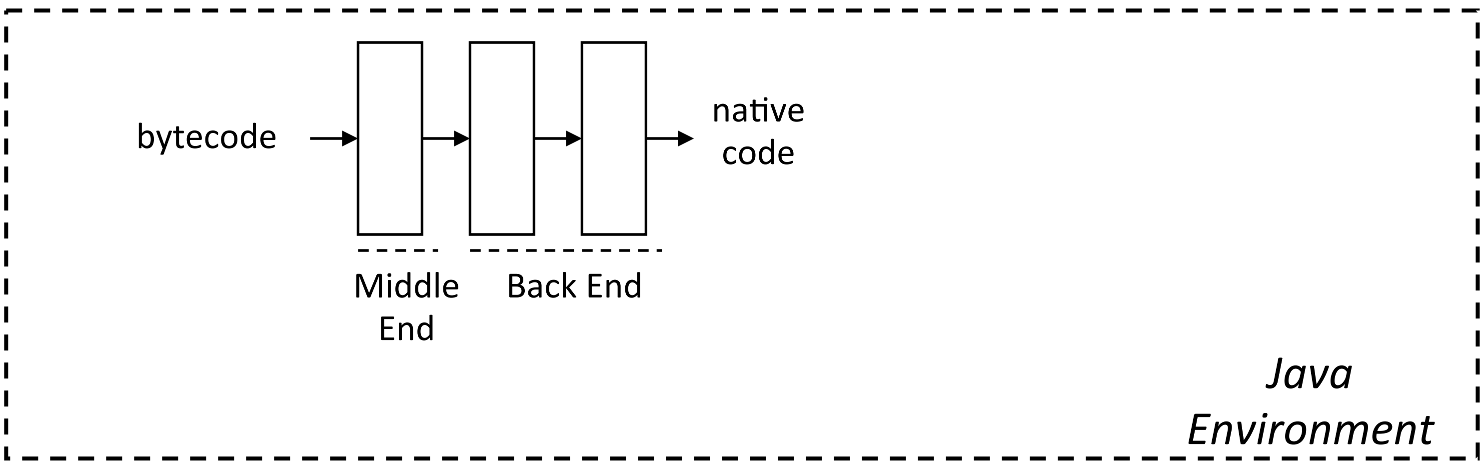
Open source optimizing compiler for IA 64

- 3 front ends, 1 back end
- Five-level IR
- Gradual lowering of abstraction level

*

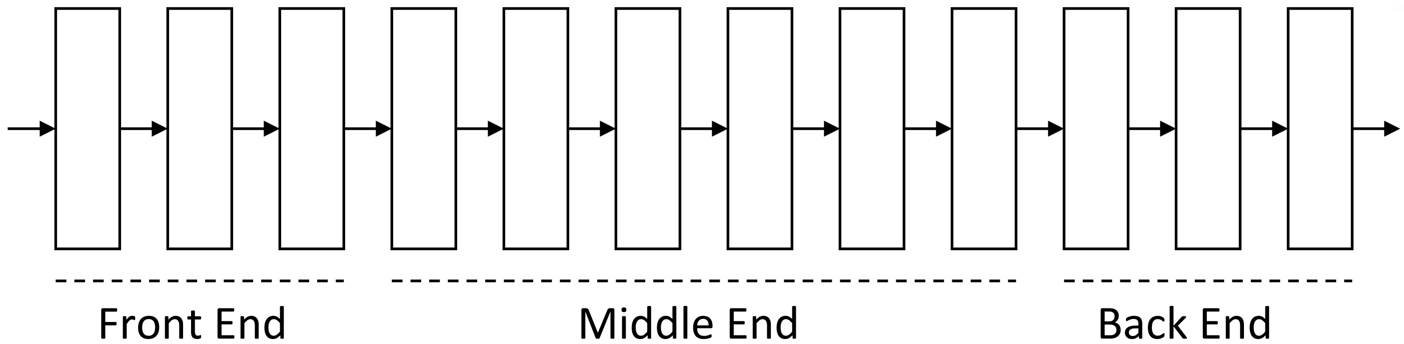
Classic Compilers

Even a modern JIT fits the mold, albeit with fewer passes [27]



- Front end tasks are handled elsewhere
- Few (if any) optimizations
  - ♦ Avoid expensive analysis
  - ♦ Emphasis on generating native code
  - ♦ Compilation must be profitable

Classic Compilers



Most optimizing compilers fit this basic framework

- What's the difference between them?
  - ♦ More boxes, better boxes, different boxes
  - ♦ Picking the right boxes in the right order
- To understand the issues
  - ♦ Must study compilers, for big picture issues
  - ♦ Must study boxes, for detail issues

  } In 512, we try to do both
- We will look at some of the great compilers of yesteryear

Fortran H Enhanced (the “new” compiler)

Improved Optimization of Fortran Object Programs [307]

R.G. Scarborough & H.G. Kolsky

Started with a good compiler — Fortran H Extended

- Fortran H — one of 1st commercial compilers to perform systematic analysis (both control flow & data flow)
- Extended for System 370 features
- Subsequently served as model for parts of VS Fortran

— not a great compiler

Authors had commercial concerns

- Compilation speed
- Bit-by-bit equality of results
- Numerical methods must remain fixed

Fortran H had 3 paths: -O0, -O1, and -O2.

The paper describes improvements in the -O2 optimization path

Fortran H Extended (the "old" compiler)

Some of its quality comes from choosing the right code shape

Translation to quads performs careful local optimization

- Replace integer multiply by 2^k with a shift
- Expand exponentiation by known integer constant
- Performs minor algebraic simplification on the fly
  - ♦ Handling multiple negations, local constant folding

Classic example of "code shape"

- Bill Wulf popularized the term [356] (*probably coined it*)
- Refers to the choice of specific code sequences
- "Shape" often encodes heuristics to handle complex issues

Fortran H Extended (old)

Some of the improvement in Fortran H comes from choosing the right code shape for the target & the compiler's optimizations

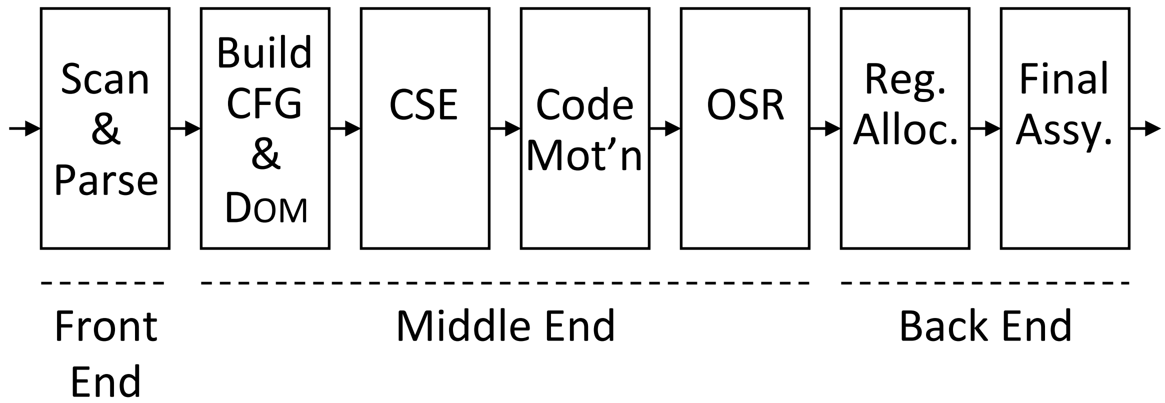
- Shape simplifies the analysis & optimization
- Shape encodes heuristics to handle complex issues

The rest came from systematic application of a few optimizations

- Common subexpression elimination
- Code motion
- Strength reduction
- Register allocation
- Branch optimization

Not many optimizations, by modern standards …
(*e.g.*, **SUIF, OPEN 64, GCC, LLVM**)

Classic Compilers (old)



-O2 Path

Summary

- This compiler fits the classic model
- Focused on a single loop at a time for optimization
- Worked innermost loop to outermost loop
- Compiler was just 27,415 lines of Fortran + 16,721 lines of asm

The parser was written in Fortran 66!

Fortran H Enhanced (new)

This work began as a study of customer applications

- Found many loops that could be better
- Project aimed to produce hand-coded quality
- Project had clear, well-defined standards & goals
- Project had clear, well-defined stopping point

Fortran H Extended was already an effective compiler

Instruction Type	Fortran G1 count	Fortran G1 pct	H Extended count	H Extended pct	H Enhanced count	H Enhanced pct
Integer	70.216	83.5	7.120	38.3	1.372	11.4
Float	10.994	13.1	9.976	53.7	9.207	76.4
Control	1.456	1.7	1.435	7.7	1.435	11.9
Others	1.459	1.7	0.044	0.2	0.044	0.4
Totals	84.126	100.0	18.575	100.0	12.058	100.0

Aggregate operations for a plasma physics code, in millions

Little decrease in useful ops

Huge decrease in overhead ops

Another 35%

78% reduction

*

Fortran H Enhanced (new)

How did they achieve this 35% improvement?

The work focused on four areas

- Reassociation of subscript expressions
- Rejuvenating strength reduction
- Improving register allocation
- Engineering issues

Note: this is ***not*** a long list !

Reassociation of Subscript Expressions

Don't generate the standard address polynomial

For those of you educated from EaC, a history lesson is needed

- Prior to this paper (& much later in the texts) the conventional wisdom was to generate the following code, following Horner's rule:

For a 2-d array `A` declared as $\texttt{A(low}_1\texttt{:high}_1\texttt{,low}_2\texttt{:high}_2\texttt{)}$

The reference $\texttt{A(i}_1\texttt{,i}_2\texttt{)}$ generates the polynomial

$$A_0 + ((i_2 - low_2) \times \underline{(high_1 - low_1 + 1)} + i_1 - low_1) \times w$$

Length of dim 1
Precompute it

- This form of the polynomial minimizes total ops
  - ♦ Good for operation count, bad for common subexpression elimination, strength reduction, instruction scheduling, …

Column-major order because the paper is about FORTRAN

Reassociation of Subscript Expressions

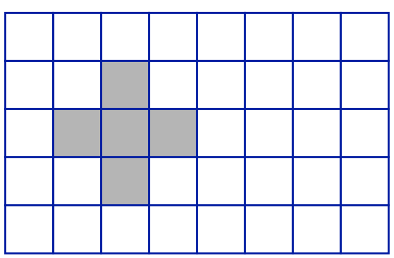
For a 2-d array `A` declared as $A(low_1{:}high_1, low_2{:}high_2)$

The reference $A(i_1, i_2)$ generates the polynomial

$$A_0 + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times w$$

- This form of the polynomial minimizes total ops
  - ♦ Good for operation count, bad for common subexpression elimination, strength reduction, instruction scheduling, …
  - ♦ With `A(i+1,j)`and `A(i+1,j+1)` the difference is bound into the expression before the common piece can be exposed
- Now, imagine a typical "stencil" computation

```
a(i,j) = (a(i-1,j) + a(i,j) + a(i+1,j) + a(i,j-1) + a(i,j+1))/5
```



Surrounding loops (on i, then j) move the stencil over the entire array, adjusting the value of the central element …

Typical stencils include 5, 7, 11 points

Still column-major order

Reassociation of Subscript Expressions

For a 2-d array `A` declared as $A(low_1:high_1, low_2:high_2)$

The reference $A(i_1, i_2)$ generates the polynomial

$$A_0 + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times w$$

- This form of the polynomial minimizes total ops
  - ♦ Good for operation count, bad for common subexpression elimination, strength reduction, instruction scheduling, …
  - ♦ With `A(i+1,j)`and `A(i+1,j+1)` the difference is bound into the expression before the common piece can be exposed
- Now, imagine a typical "stencil" computation

```
a(i,j) = (a(i-1,j) + a(i,j) + a(i+1,j) + a(i,j-1) + a(i,j+1))/5
```

And the subexpressions found (or hidden) inside it …

Still column-major order

Reassociation of Subscript Expressions

Don't generate the standard address polynomial

… Forget the classic address polynomial …

- Break polynomial into six parts
  - ♦ Separate the parts that fall naturally into outer loops
  - ♦ Compute everything possible at compile time
- Makes the tree for address expressions broad, not deep
- Group together operands that vary at the same loop level

The point

- Pick the *right* shape for the code
- Let other optimizations do the work
- Sources of improvement
  - ♦ Fewer operations execute
  - ♦ Decreases sensitivity to number of dimensions

(*expose the opportunity*)

Tradeoff driven by **CSE** versus strength reduction

FORTRAN H chooses the shape based on local analysis of the subscript, trading off possible **CSE** & **LICM** against **OSR**.

Read pp 665ff in the paper carefully.

Reassociation of Subscript Expressions

Distribution creates different expressions

`w + y * (x + z)` ⇒ `w + y * x + y * z`

More operations, but they may move to different places

Consider `A(i,j)`, where `A` is declared `A(0:n,0:m)`

- Standard polynomial: `@A + (i * m + j) * w`
- Alternative: `@A + i * m * w + j * w`

Does this help?

- In a typical loop nest, the `i` part and `j` part vary in different loops
- Standard polynomial pins `j` in the loop where i varies

Can produce **significant** reductions in operation count

General problem, however, is **quite** complex

Operator Strength Reduction (OSR)

Their OSR was not particularly effective at the start of the study

- Many cases had been disabled in maintenance
  - ♦ Almost all the subtraction cases turned off
- Fixed the bugs and re-enables the corresponding cases
- Caught "almost all" the eligible cases

Extensions

Increases the cost, but has less practical impact than asymptotic analysis would suggest.

- Iterate the transformations
  - ♦ Avoid ordering problems (i+j)*4
  - ♦ Catch secondary effects
- Capitalize on user-coded reductions (*shape*)
- Eliminate duplicate induction variables
  - ♦ Explicit xform to shift address calculations to common induction variables

In this context, OSR refers to the general optimization, not the specific Cooper-Simpson-Vick algorithm [107].

Register Allocation

Original Allocator

- Divide register set into local & global pools
  - ♦ "Global" means a loop nest
- Different mechanisms for each pool
  - ♦ Local based on Best's algorithm
  - ♦ Global based on frequency counts

Best's algorithm is also known as "bottom-up local" in EaC2e and as Belady's **MIN** algorithm for optimal offline page replacement.

Problems

- Bad interactions between local & global allocation
- Unused registers dedicated to the procedure linkage
- Unused registers dedicated to the global pool
- Extra (unneeded) initializations

Remember the 360
- ♦ Two-address machine
- ♦ Destructive operations

Register Allocation

New Allocator

- Remap to avoid local/global duplication
- Scavenge unused registers for local use
- Remove dead initializations
- Section-oriented branch optimizations

All symptoms arise from not having an actual global register allocator

Plus …

- Change in local spill heuristic from frequency to distance
- Can allocate all **FOUR** floating-point registers
- Bias register choice by selection in inner loops
- Better spill cost estimates
- Better branch-on-index selection

Engineering Issues

Increased the name space used in analysis

- Was 127 slots (80 for variables & constants, 47 for compiler)
- Increased to 991 slots
- Constants no longer need slots
- "Very large" routines need < 700 slots (*remember inlining study*?)

Common subexpression elimination (CSE)

- Removed limit on backward search for CSEs
- Taught CSE to avoid some substitutions that cause spills

Again, sounds as if it would cause asymptotic problems, but it did not, in practice.

Extended constant handling to negative values

Results

Hand-coding no longer improved the inner loops.

They stopped working on the optimizer.

⟹ Produced a significant change in ratio of flops to instructions

Fortran H Extended is the classic Fortran optimizing compiler

Instruction Type	Fortran G1 count	Fortran G1 pct	H Extended count	H Extended pct	H Enhanced count	H Enhanced pct
Integer	70.216	83.5	7.120	38.3	1.372	11.4
Float	10.994	13.1	9.976	53.7	9.207	76.4
Control	1.456	1.7	1.435	7.7	1.435	11.9
Others	1.459	1.7	0.044	0.2	0.044	0.4
Totals	84.126	100.0	18.575	100.0	12.058	100.0

Aggregate operations for a plasma physics code, in millions

The PL.8 Compiler (*ten years after Fortran H*)

First RISC Compiler [24, 90, 74, 75]

- Original target was **IBM 801** minicomputer
- Tight coupling of architecture & compiler
- Later targets included **S/370**, **MC680x0**, & others
- Basis for **XL** compiler series for **RS/6000** & **POWER** machines

Hardware/software co-design

Research compiler

- Compilation speed was not critical
- Emphasis on code quality, methodology, & theory
- Several breakthrough ideas
- Underlying philosophy governs **RISC** compilers today

The Language

A PL/I Subset (*80/20 rule*)

- Threw out **ON** conditions (*exception handling*)
- Permanently enabled subscript range checking
- Replaced unrestricted pointers with offsets & areas
- Bit string lengths fixed and restricted
- New declarations for call-by-value
- No internal static variables
- Relaxed implicit conversion rules
- Simplified rules governing arithmetic precision

Eventually built other front ends

- Pascal, Fortran, & C

Compiler Summary

Intermediate Representation

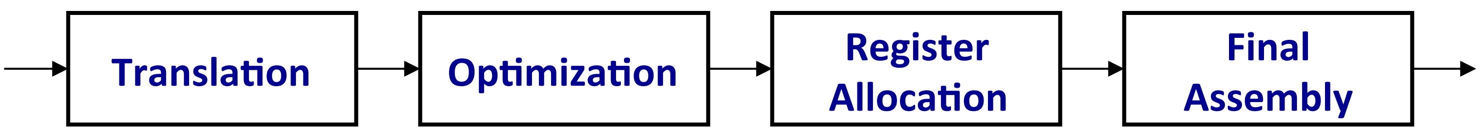
- Linear, low-level, abstract machine code
- Byte addressable storage
- Unlimited set of symbolic or virtual registers
- High-level operations to encapsulate control flow

MAX
MIN
MVCL
CHECK

Optimization

- Use global (*whole procedure*) techniques
- Expose ***every*** detail to uniform optimization

Structure



Principles

Assumptions

- Register allocator does a great job (*separation of concerns*)
- Instruction set has limited number of alternatives
- Little or no special case analysis
- Broad set of optimizations covers the IR

Doctrine

- Data-flow analysis pays off, so do it when needed
- Passes are independent but complementary
- Code is shaped for optimization
- Optimize, elaborate, optimize
- Finite machine is the allocator's problem

As a matter of timing, PL.8 came out at a time when **DF** analysis was well developed & before **SSA** was invented.

As in **FORTRAN H**, **BLISS 11**, and other classic optimizers

Philosophy

PL.8 followed a somewhat rigid set of guidelines that influenced its successors, down to the present day.

- It used an **IR** with two distinct levels of abstraction
  - ♦ Macro-like expansion from abstract to detailed
- It repeated optimizations multiple times for best effect
  - ♦ Trade compile time directly for performance
  - ♦ Acknowledgement that "phase ordering" affects actual performance
- It relied heavily on separation of concerns & a single point of control
  - ♦ **CSE** and **OSR** inserted new code, **DCE** removed (now) unused code
  - ♦ **GCRA** worried about register pressure and copies, other passes did not
- It repeated analysis rather than trying to update it after change to the **IR**
  - ♦ Incremental updates are tricky and easy to break; re-analysis simply reuses code
  - ♦ Repetition allows passes to be reordered and repeated.

This slide is redundant, but as my friend **SKW** says: tell them what you are going to tell them, tell it to them, and tell them what you told them.

Translation Phase

Simple front end

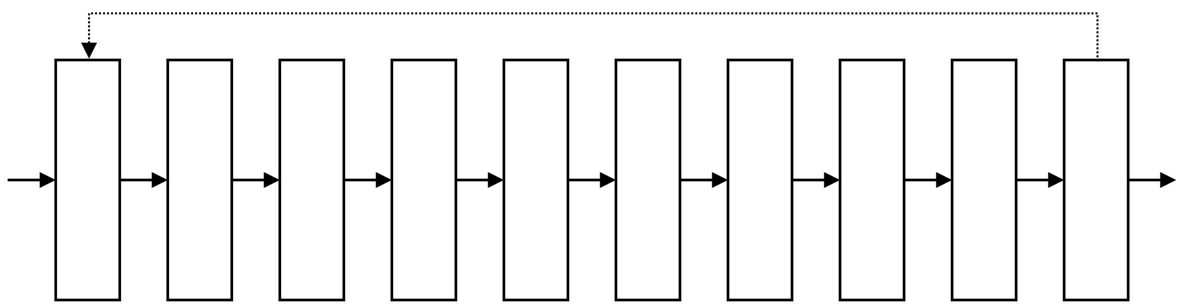
- **LALR(1)** parser
- Bottom-up generation of **IR**
- No significant analysis during translation
- Some machine-specific detail creeps in (*branch ranges*)
- Shape the code for optimization (*syntactic & local*)

Front end does not

- Build a control-flow graph
- Analyze the content for special cases
- Pre-assign registers (*other than the **ARP***)

Optimizer

Structure



(conceptual)

- Many passes
  - ♦ Independent & interdependent
  - ♦ Single point of control
  - ♦ Repeats some passes multiple times
- IR is definitive representation
  - ♦ Re-derive rather than update
- Insert & eliminate rather than replace
  - ♦ Rely on dead code elimination

Dead code elimination
Global CSE
Code motion
Constant folding
Strength reduction
Value numbering
Dead store elimination
Code straightening
Trap elimination
Algebraic reassociation

*

PL.8 had the first graph-coloring register allocator [74, 75].

Register Allocator

Graph coloring allocator (*see Chapter 13, EaC2e*)

- Constructs precise interference graph
- Use interference graph to coalesce copies (*unlike Chow*)
- Machine-specific constraints modeled in graph
- Use smallest degree last coloring
- Allocator handles all spill decisions

Effectiveness - compiling the compiler

- For S/370 (16 **GPR**s): little more than 50% of values spill
- For 801 (32 **GPR**s): over 95% do not spill
- Coloring works better with larger register sets (*spill heuristic*)

Scheduling & Final Assembly

Schedule Twice

- Pre-allocation scheduling to avoid constraints
- Post-allocation scheduling to place spill code

Final assembly

- Convert allocated, scheduled IR to object code
- Two passes with some local fix-up (*peephole*)
- Generate debugging information, tags for link-time checking
- Added tailored procedure prologs and epilogs

Schedule-Allocate-Schedule contradicts their description of Allocation as the 3rd major phase and Scheduling as the 4th. I would attribute the discrepancy to the fact that they seemed to reconfigure the compiler often. For example, the two papers seem to describe somewhat different setups for the compiler.

Miscellany

Range checking

- One goal was to decrease overhead of checking
- Lots of intellectual effort invested in this problem (*V. Markstein*)
- Area + offset could be checked, pointer could not
- Cocke & P. Markstein report 5% to 10% overhead
  - ♦ V. Markstein reports (eventually) getting that down to 2% [257]

Reliability

- **PL.8** was built with **PL.8**
- Daily use improved actual & perceived reliability

Results (From Cocke & Markstein)

	Optimization Level				
Transformation	-1	0	1	2	3
Dead code elimination		x	x	x	x
Value numbering		x	x	x	x
Local constant propagation		x	x		
Global commoning, code motion			x	x	x
Strength reduction			x	x	x
Macro expansion	x	x	x	x	x
Dead code elimination				x	x
Value numbering				x	x
Local constant propagation				x	x
Register allocation ($k = r - 4$)				x	
Register allocation (k = $r + 4$)					x

PL.8 compiler option flags

Results

(From Cocke & Markstein)

Program		Optimization Level -1	0	1	2	3
USEDEF	*Compile time*	19.7	19.7	31.7	34.2	51.2
(360 lines)	*Code Space*	12,138	5,386	6,390	6,098	5,942
	Run time	0.720	0.230	0.134	0.129	0.124
Puzzle	*Compile time*	6.2	5.7	9.3	10.2	14.7
(154 lines)	*Code Space*	2,790	1,682	1,778	1,782	1,698
	Run time	1.330	0.730	0.670	0.670	0.620
IPOO	*Compile time*	9.8	10.3	15.5	17.3	20.5
(295 lines)	*Code Space*	4,908	3,404	3,232	3,216	3,156
	Run time	5.880	4.250	3.610	3.590	3.510
Heapsort	*Compile time*	2.2	1.9	2.3	2.5	2.5
(84 lines)	*Code Space*	1,024	432	384	368	368
	Run time	5.600	2.260	2.120	2.020	2.020
Heapsort	*Compile time*		0.83		0.96	
(in PL/I)	*Code Space*		740		700	
	Run time		4.310		4.000	
Heapsort	*Compile time*		0.26		0.33	0.38
(in Fortran)	*Code Space*		674		490	442
	Run time		4.830		2.880	2.880

Spill code

System 370, times in seconds

Notes on Results Slides

- Level 0 pays for itself (smaller code)
- Global code motion & cse lengthen live ranges (level 0 to 1)
- Biggest payoff is level -1 to 0, then 0 to 1; global optimization compensates for longer lifetimes
- Level 3 only helps with spill code (made obsolete by Briggs)
- Spilling increases code space, but increased optimization makes up for it (zero wait state memory)
- USEDEF references complex data structures in nested loops
- Tests exclude reassociation; Cocke & Markstein report that reassociation removes up to 50% of the code in USEDEF's inner loops; helps with spilling & speed
- No linear function test replacement
- Constant propagation underperformed expectations; initial values not represented in the IR
- Heapsort doesn't show off Fortran H, because it doesn't use the loop index variable as a subscript index!

Key Points

Strong philosophical influence on later compilers

- Single point of control
- Repeat optimizations
- Two-level IR
- Separation of concerns (strong back end)
- Reanalyze rather than update incrementally

Scope of optimization

- Notice large improvement from -1 to 0 (*local optimization*)
- Design emphasizes global analysis and optimization
- Results show a payoff (*smaller than local, but, …*)
- Contrasts with Fortran H's emphasis on loop nests

Hardware/software co-design

- 801 ISA designed as target for this compiler

Key Points

Graph coloring register allocation

- Precise interference graph
- Uniform approach to spilling (*no local RA*)
- Powerful method for coalescing copies (*vs. Chow*)

Reassociation

- Recognized the potential & worked on the problem
- In the end, method did not work as promised *(Hopkins, private communication)*

Triumph of global analysis & optimization

- Decade of new algorithms
- This compiler showed that, in practice, it all worked
- Did well against mature S/370 compilers (*not just on 801*)