OEP of SYSTEM PROGRAMMING

Prepared by

Name	Enrollment no.
Dipen Prajapati	170050107090
Meet Prajapati	170050107091

Aim: Case Studies of Compilers and Future Trends

Ran Shaham

Outline

We first discuss three commercial compilers. This is intended for the understanding of how the various techniques and implementation considerardions presented throughout the course, are implemented in real-world compilers.
For each compiler we start by giving a short description of the machine architecture or architectures for which it is targeted. Then we give its history, discuss the compiler's strcuture and use two example programs to illustrate the effects of the compiler, focusing in the performed and missed opitimizations on the two programs. We conclude this part by summarizing the performance of the discussed compilers on the example programs. This can serve as a partial comparision criteria for the compilers.
The presented compilers are :

- The Sun compilers for SPARC
- The IBM XL compilers for the POWER and POWERPC architectures
- The Intel Reference compilers for the Intel 386 architecture family

The second part of the class is devoted to discussing the clear main (and other) trends in compilers design and implementation.
At last, we list theoretical techniques and understand where they come into practice in the compiler phases.

The Example Programs

The first example program is the following C routine :

```
1   int length, width, radius;
2   enum figure {RECTANGLE, CIRCLE};
3   main()
4   {   int area=0, volume=0, height;
5       enum figure kind = RECTANGLE;
6       for (height = 0; height < 10; height++)
7       {       if (kind == RECTANGLE)
8               {       area += length * width;
9                       volume += length * width * height;
10              }
11              else if (kind == CIRCLE)
12              {       area += 3.14 * radius * radius;
13                      volume += 3.14 * radius * radius * height;
14              }
15      }
16      process(area, volume);
```

Note that in the if statement inside the loop, the value of kind is the constant RECTANGLE, so the second branch and the test itself are dead code. The value of length*width is loop-invariant, so in fact area can be calcultaed by a single multiplication (area=10*length*width). Volume is an induction variable.
We should expect compiler to perform loop unrolling by some factor.
Note also that the call to process is a tail call.

The second example program is the following Fortran 77 routine :

```
1               integer a(500,500), k, l
2               do 20 k = 1,500
3                       do 20 l = 1,500
4                               a(k,l) = k + l
5       20      continue
6               call s1(a,500)
7               end
8
9               subroutine s1(a,n)
10              integer a(500,500),a
11              do 100 i = 1,n
12                      do 100 j = i+1,n
13                              do 100 k = 1,n
14                                      l = a(k,i)
15                                      m = a(k,j)
16                                      a(k,j) = l + m
17      100     continue
18              en
```

The main simply initializes the elmenets of the array a() and calls s1(). The second and third array references in s1()are to same element, so computation of its address should be subject to common-subexpression elimination. The innermost loop should load two values, add them, store the result, update both address, test for temination and branch. The test termination condition should be replaced by a linear-function based on one of the array element addresses, thus basic induction variables can be eliminated.
Note also that s1()can be integrated into main(). Alternatively, interprocedural constant propagation can be used to detect n has constant value of 500 in s1().

The Sun Compilers for SPARC

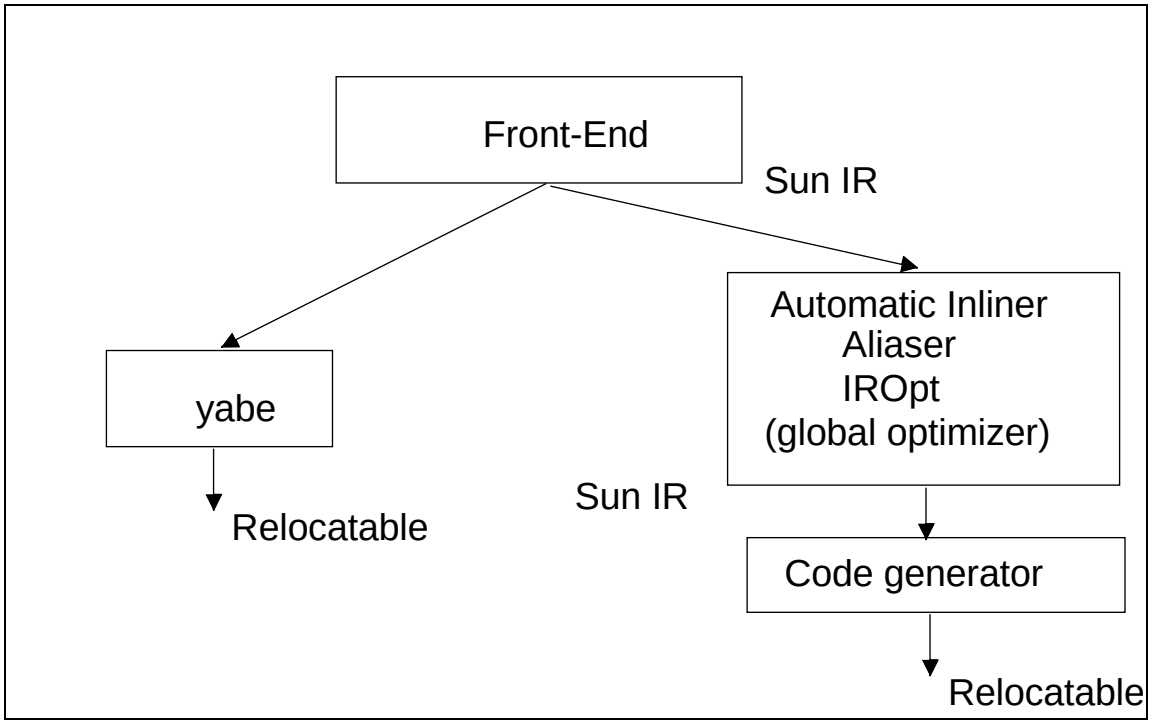
The SPARC Architecture

- SPARC 8
- 32 bit RISC superscalar system with pipeline
- integer and floating point units
- 8 general purpose registers (r_0 □0)
- load, store, arithmetic, shift, branch, call and system control
- simple addressing modes (register+register, register+displacement)
- three address instruction (i.e., in general the instruction is of the form y = x □ z, where x,y,z are three address and □ is an operator) - here, the first source and the result are amost always a register.
- several 24 register windows (spilling by OS) - architecture and OS support for calls.
- SPARC 9
- 64 bit architecture (upward compatible)

The Sun Compilers for SPARC

- Front-ends for C, C++, Fortran 77, Pascal
- Originated from Berkely 4.2 BSD Unix
- Developed at Sun since 1982
- Original backend for Motorola 68010
- Migrated to M6800 and then for SPARC
- Global optimization developed at 1984
- Interprocedural optimization began at 1984
- Mixed compiler Mode - 2 levels of optimization : before and after code generation

The target of the front ends is an intermediate language called SUN IR that represent a program as a linked list of triplets (executable operations) and several tables (declerative information). The compiler is able to produce relocatable code without optimization through YABE (“yet another back end”), or an optimized code through IROPT, the global optimizer, and then using the code generator with a postpass optimizer to produce the relocatable code. The below figure illustrate the structure of the compiler :



Optimization Levels

The compiler supports four levels of optimization.:

- O1 Limited optimizations only in the optimizing components of the code generator
- O2 Optimize expressions not involving global, aliased local, and volatile variables, automatic inlining, software pipelining, loop unrolling and instruction scheduling are skipped
- O3 optimize expressions that involve global variables, but makes worst case assumptions on pointer aliases.
- O4 makes worst case assumptions on pointer aliases only when necessary. it depends on the front-ends to provide aliases. automatic inliner and aliaser are used in this level of optimization.

iropt

This is the global optimizer, performing the optimizations on the Sun-IR (medium level IR). iropt transform Sun-IR to Sun-IR. some characteristics of it :

- Processes each procedure seperately
- Use basic blocks
- Control flow analysis using dominators
- Parallalizer uses structural analysis
- Other optimizations using iterative algorithms

Optimizations in iropt

- Scalar replacement of aggregates (replace a structure by several scalar variables) and expansion of Fortran arithmetic on complex numbers.
- dependence-based analysis and transformation in order to support parallelization and data-cache optimization (O3, O4 only)
- linearization of array addresses
- algebraic simplification and reassociation of address expressions
- loop invariant code motion
- strength reduction and induction variable removal
- global common-subexpression elimination
- dead-code elimination

Dependence Based Analysis

- Constant propagation
- dead-code elimination
- structural control flow analysis
- loop discovery (index variables, lower and upperbounds)
- segregation of loops that have calls and early exits
- dependence analysis using GCD
- prove that array subscripts are independent by verifying that for at least one subscript position the GCD is 1
- loop distribution (split loop to two loops with same iteration space, such that the first loop contains several statements of the original loop and the second contains the others).
- loop interchange (reverse the order of two adjacent loops in a loop nest)
- loop fusion (combine two loops with the same iteration space to a single loop)
- scalar replacement of array elements (this makes them available to register allocation)
- recognition of reductions
- data-cache tiling (choose tile sizes, such that each fragment of work can be completable without any cache mises other than compulsory ones)
- profitability analysis for parrallel code generation

Code Generator

First translates Sun-IR to asm+, which is an assembly + control flow and data dependence information. Then performs the following phases :

- instruction selection
- inline of assembly language templates
- local optimizations (dead-code elimination, branch chaining, ...)
- macro expansion (replace macro by explicit code, to possibly “help” the optimizations in the following phases)
- data-flow analysis of live variables
- early instruction selection - assuming all instructions use registers
- register allocation by graph coloring
- stack frame layout
- macro expansion (MIN, MAX, MOV)

- late instruction scheduling
- inline of assembly language constructs
- macro expansion
- emission of relocatable code

Optimizations on main

We now look at the resulting code by the Sun SPARC compiler (virtually look, since the really looking at it will not lead to a better understanding...). let us point out the optimizations performed (and the ones that the compiler have missed) :

- Removal of dead code in else (except o)
- detected that the value of kind is the constant RECTANGLE, and as a result the the else code is dead
- o was "prepared" as a constant, but was not removed after detecting that all uses of this constant are in dead code
- Removal of loop invariant "length * width"
- Strength reduction of "height" - cacluated using additions, although it could be computed using one multiplication
- Loop unrolling by factor of four - although all loop iterations could be unrolled
- Local variables in regisers
- All computations in registers
- Identify tail call

As already mentiones the missed optimizations were

- Removal of o computations
- One multiplication operation for computing "height"
- Completely unroll the loop

Optimizations on the Fortran example

Optimizations performed :
- Inlining of S1 (n=500)
- Common subexpression elimination of "a[k,j]"
- Loop unrolling
- Local variables in registers
- Software pipelining

Missed Optimizations :
- Eliminating S1
- Eliminating addition in loop via linear function test replacement

The IBM XL Compilers

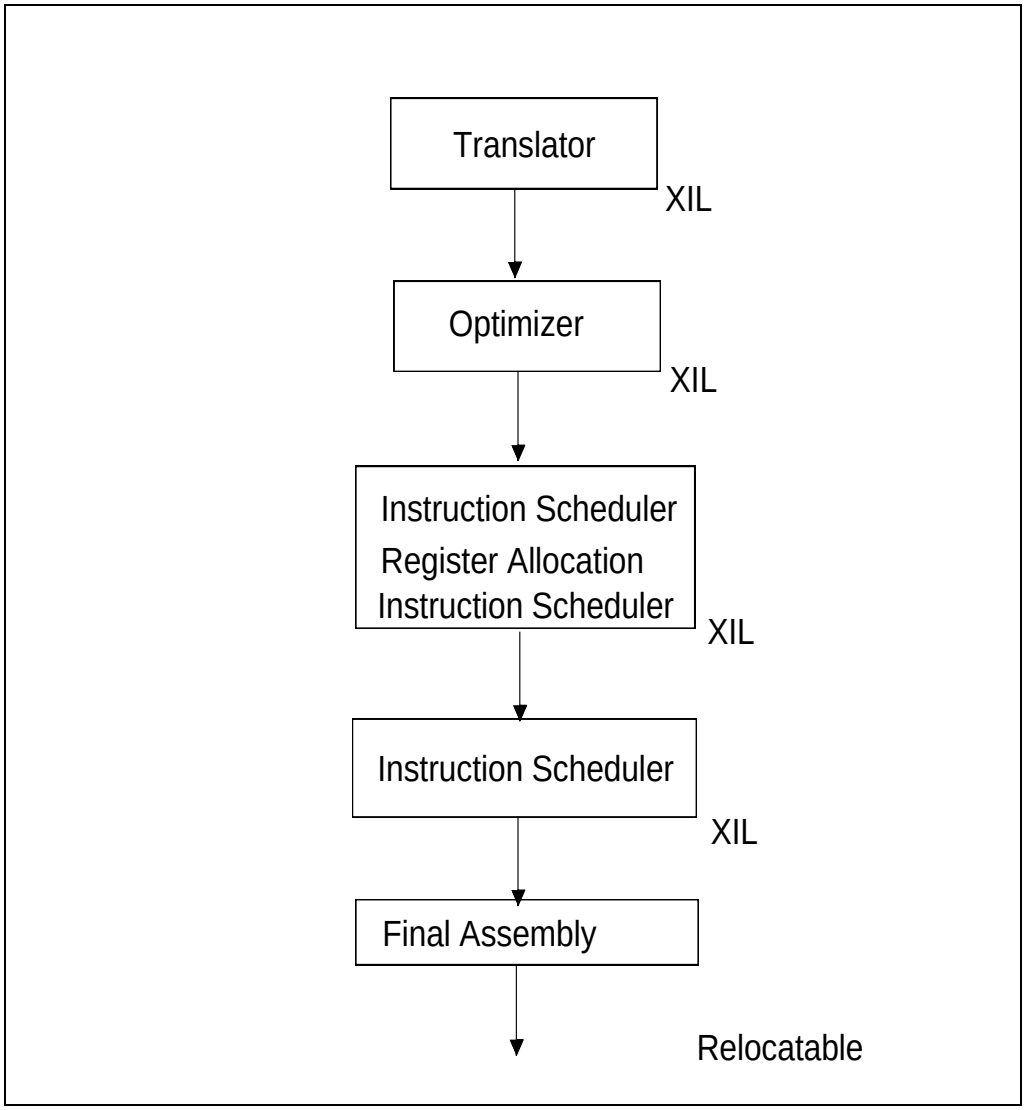
The POWE/PowerPC Architecture

- POWER
- 32 bit RISC superscalar system with
- branch, integer and floating point units
- optional multiprocessors, using shared regiseters (one branch unit)
- 32 (shared) general purpose integer registers (gr_0 0)
- load, store, arithmetic, shift, branch, call and system control
- simple addressing modes (register+register, register+displacement)
- three address instruction (i.e., in general the instruction is of the form y = x z, where x,y,z are three address and is an operator)
- PowerPC
- Both 32 and 64 bit architecture

The IBM XL Compilers

- Front-ends for PL. 8, C, C++, Fortran 77, Pascal
- Originated in 1983
- Written in PL.8
- First released for PC/RT
- Generated code for Power, Intel386, SPARC and PowerPC
- No interprocedural optimization
- (Almost) all optimizations on low level IR (XIR)

The structure of the XL compiler, follow the the low-level model of optimization. It consists of a front-end call a translator, a global optimizer, an instruction scheduler, a register allocator, an instruction selector, and a phase called final assembly that produces the relocatable image and assembly-language listings. The translator converts the source language to an intermediate form, call XIL. All other phases, actually, the compiler backend, are named TOBEY.



TOBEY

- Processes each procedure seperately
- Use basic blocks
- Control flow analysis in using DFS and intervals
- YIL a higher level representation, constucrted from XIL,used for storage related optimization
- YIL includes loop constructs
- YIL code is represented SSA form
- after that YIL is translated back to XIL
- Data flow analysis by interval analysis
- Iterative algorithm for non reducible graph

Optimizations in TOBEY

- transforms Switches to a sequence of compares and conditional branches or branches through a branch table according to the density of the labels
- Mapping local variables to register+offset

- inline routines from current compilation module
- Aggressive value numbering
- global common subexpression elimination
- loop-invariant code motion
- downward store motion - since store instructions can delay the code execution, perform it as late as possible.
- dead-store elimination
- reassoication, strength reduction
- global constant propagation
- architecture specific optimizations (MAX)
- value numbering
- global common subexpression elimination
- dead code elimination
- elimination of dead induction variables

Optimizations on main

We now look at the resulting code by the IBM XL compiler. Again, without really looking at the resulted code, and again, let us point out the optimizations performed (and the ones that the compiler have missed) :

- Removal of dead code in else
- detected that the value of kind is the constant RECTANGLE, and as a result the the else code is dead
- Removal of loop invariant “length * width”
- Strength reduction of “height” - cacluated using additions, although it could be computed using one multiplication
- Loop unrolling by factor of two - although all loop iterations could be unrolled
- Local variables in regisers
- All computations in registers

missed optimizations :

- One multiplication operation for computing “height”
- Completely unroll the loop
- Identify tail call

Optimizations on the Fortran example

Optimizations performed :
- Inlining of S1 (n=500)
- Common subexpression elimination of “a[k,j]”
- Eliminating addition in loop via linear function test replacement
- Local variables in registers

Missed Optimizations :
- Loop unrolling

The Intel Compilers

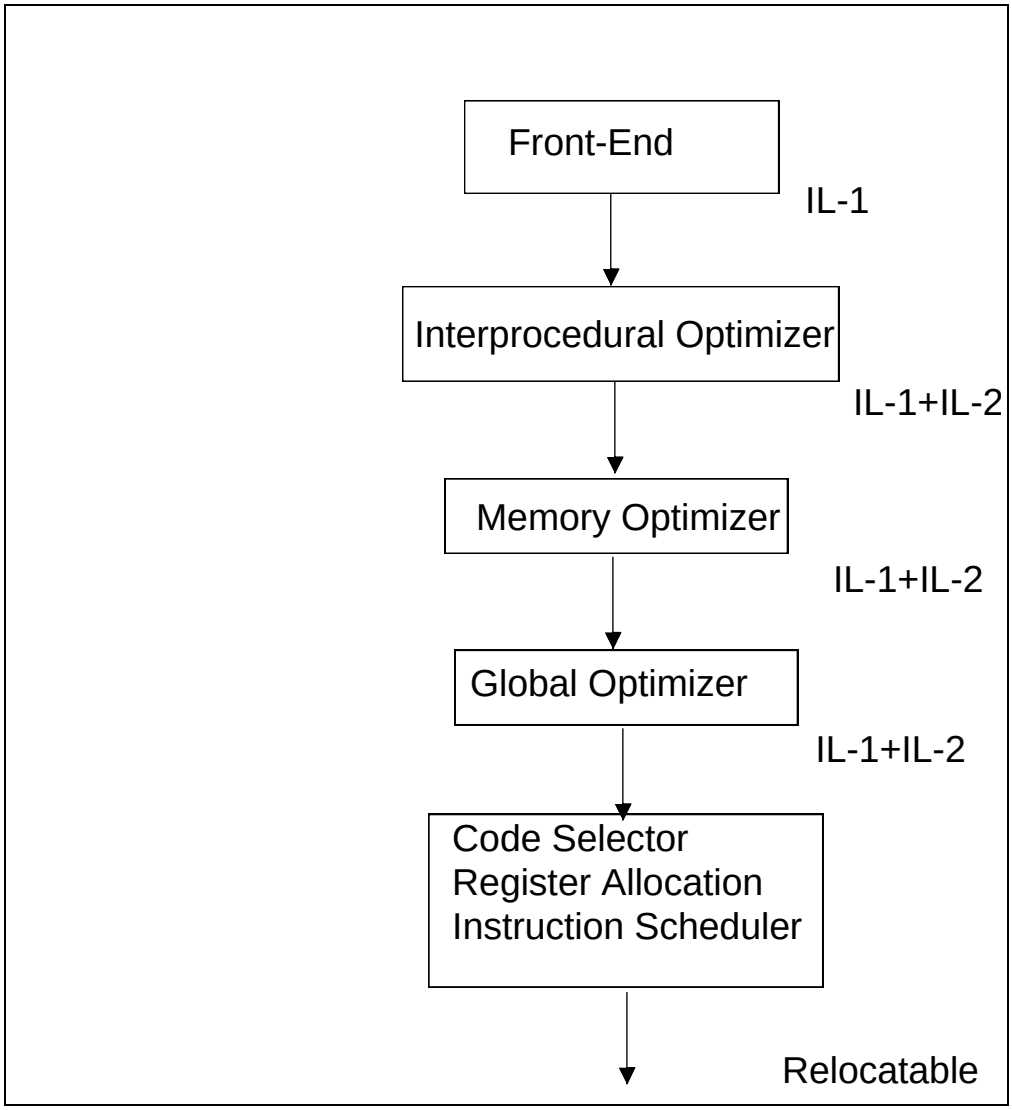
Intel 386 Architecture

- 32 bit CISC system
- 8 32-bit integer registers
- Support 16 and 8 bit registers
- Dedicated Registers (e.g. stack frame)
- Many address modes
- Two address instructions
- 80 bits floating point registers

The Intel Compilers

- Intel provides what it calls a reference compiler. A compiler “demonstrating “ how to write a compiler for the Intel 386 architecture family.
- Front-ends for C, C++, Fortran 77, Fortran 90
- Front-End from Multiflow and Edison Design Group (EDG)
- Generates 386 code
- Interprocedural optimization were added (1991)
- Mixed optimization mode
- Many optimizations based on partial redundency elimination
- Uses relatively new techniques (as above mentioned, interprocedural optimization and partial redundency elimination)

The structure of the compiler is illustrated in the below figure. The front-ends produce a medium-level IR called IL-1 (includes features like array indexes). The interprocedural optimizer operates accross file boundaries (by saving the IL-1 form of each routine to files). The output of the interprocedural optimizer is a lowered version of IL-1, called IL-2 allong with IL-1. The memory optimizer is concerned with improving use of memory and caches. The other phases are global optimizer, described later and a code generator.



Interprocedural Optimizer

- Cross Module, accross file boundaries, by saving intermediate representations
- Interprocedural constant propagation

Memory Optimizer

- Improves memory and caches, mainly by loop transformations.
- Uses SSA form to perform sparse conditional constant propagation, and then perform data-dependence tests for loops with known boundaries

Global Optimizer

- Constant propagation
- Dead code elimination
- local common subexpression elimination

- copy propagation
- parial redundency elimination
- copy propagation
- Dead code elimination

Optimizations on main

We now look at the resulting code by the IBM XL compiler. Again, without really looking at the resulted code, and again, let us point out the optimizations performed (and the ones that the compiler have missed) :

- Removal of dead code in else
- detected that the value of kind is the constant RECTANGLE, and as a result the the else code is dead
- Removal of loop invariant “length * width”
- Strength reduction of “height” - cacluated using additions, although it could be computed using one multiplication
- Local variables in regisers
- All computations in registers

missed optimizations :

- One multiplication operation for computing “height”
- Loop unrolling
- Identify tail call

Optimizations on the Fortran example

Optimizations performed :
- Common subexpression elimination of “a[k,j]”
- Loop unrolling by factor of two
- Local variables in registers

Missed Optimizations :
- Inlining of S1 (n=500)- despite of the inline routine optimization in TOBEY
- Eliminating addition in loop via linear function test replacement

Future Trends in Compiler Design and Implementation

- SSA is being used more and more”
- generalizes basic block optimization to extended basic blocks
- leads to performance improvements
- Partial redundency elimiation is being used more, and as we’ve seen it is used in the Intel reference compiler.

- Partial redundency and SSA are being combined
- Parallelizationds and vectorization are being integrated into production compilers
- Data-dependence testing, data-cache optimization and software pipelining will advance significantly
- The most active research area in scalar compilation will be optimization.

Other Trends

- More and more work will be shifted from hardware to compilers
- More advanced hardware will be available
- Higher order programming languages will be used
- Memory management will be simpler
- Modularity facilities
- Assembly programming will hardly be used
- Dynamic (run-time) compilation will become more significant

Theoretical Techniques in Compilers

The following table lists theoritical techniques and their use in the compiler phases:

Technique	Compiler Phase
Data Structures	All
Automata Algorithms	Front-End, Instruction Selection
Graph Algorithm	Control-Flow Data-Flow Register Allocation
Linear Programming	Instruction Selection (Complex Machines)
Diophantic Equations	Parallelization
Random Algorithm	NONE