The SGI Pro64 Compiler Infrastructure

- A Tutorial

Guang R. Gao (U of Delaware)  J. Dehnert (SGI)
J. N. Amaral (U of Alberta)  R. Towle (SGI)
Acknowledgement

The SGI Compiler Development Teams
  – The MIPSpro/Pro64 Development Team

University of Delaware
  – CAPSL Compiler Team

These individuals contributed directly to this tutorial

A. Douillet (Udel)     F. Chow (Equator)
S. Chan (Intel)        W. Ho (Routefree)
Z. Hu (Udel)           K. Lesniak (SGI)
S. Liu (HP)            R. Lo (Routefree)
S. Mantripragada (SGI)  C. Murthy (SGI)
M. Murphy (SGI)        G. Pirocanac (SGI)
D. Stephenson (SGI)    D. Whitney (SGI)
H. Yang (Udel)
What is Pro64?

- A suite of optimizing compiler tools for Linux/Intel IA-64 systems
- C, C++ and Fortran90/95 compilers
- Conforming to the IA-64 Linux ABI and API standards
- Open to all researchers/developers in the community
- Compatible with HP Native User Environment
Who Might Want to Use Pro64?

- *Researchers*: test new compiler analysis and optimization algorithms
- *Developers*: retarget to another architecture/system
- *Educators*: a compiler teaching platform
Outline

• Background and Motivation
• Part I: An overview of the SGI Pro64 compiler infrastructure
• Part II: The Pro64 code generator design
• Part III: Using Pro64 in compiler research & development
• SGI Pro64 support
• Summary
PART I:
Overview of the Pro64 Compiler
Outline

- Logical compilation model and component flow
- WHIRL Intermediate Representation
- Inter-Procedural Analysis (IPA)
- Loop Nest Optimizer (LNO) and Parallelization
- Global optimization (WOPT)
- Feedback
- Design for debugability and testability
Logical Compilation Model

- **Driver** (sgicc/sgif90/sgiCC)
  - **Front end + IPA** (gfec/gfecc/mfef90)
  - **Back end** (be, as)
  - **Linker** (ld)

- **Source** (.c/.C/.f)
- **WHIRL** (.B/.I)
- **Object** (.o)
- **Output** (a.out/.so)

Data Path

Fork and Exec
Components of Pro64

Front end

Interprocedural Analysis and Optimization

Loop Nest Optimization and Parallelization

Global Optimization

Code Generation
Data Flow Relationship Between Modules

Lower to High W. -IPA Local IPA Main IPA -O₃ LNO

.gfec .B

gfecc .I

.f90 lower I/O (only for f90)

.WHIRL .w2c.c

.C .w2c.h

.WHIRL .w2f.f

.fortran

Lower all -O₀

-phases: w=off

-Lower Mid W -O₂/O₃

Main opt

CG

Take either path

-cn Very high WHIRL

-cn High WHIRL

-cn Mid WHIRL

-cn Low WHIRL
Front Ends

- C front end based on gcc
- C++ front end based on g++
- Fortran90/95 front end from MIPSpro
Intermediate Representation

IR is called WHIRL

- Tree structured, with references to symbol table
- Maps used for local or sparse annotation
- Common interface between components
- Multiple languages, multiple targets
- Same IR, 5 levels of representation
- Continuous lowering during compilation
- Optimization strategy tied to level
IPA Main Stage

Analysis
  – alias analysis
  – array section
  – code layout

Optimization (fully integrated)
  – inlining
  – cloning
  – dead function and variable elimination
  – constant propagation
IPA Design Features

• User transparent
  – No makefile changes
  – Handles DSOs, unanalyzed objects

• Provide info (e.g. alias analysis, procedure properties) smoothly to:
  – loop nest optimizer
  – main optimizer
  – code generator
Loop Nest Optimizer/Parallelizer

- All languages (including OpenMP)
- Loop level dependence analysis
- Uniprocessor loop level transformations
- Automatic parallelization
Loop Level Transformations

• Based on unified cost model
• Heuristics integrated with software pipelining
• Loop vector dependency info passed to CG

  – Loop Fission
  – Loop Fusion
  – Loop Unroll and Jam
  – Loop Interchange
  – Loop Peeling
  – Loop Tiling
  – Vector Data Prefetching
Parallelization

- Automatic
  - Array privatization
  - Doacross parallelization
  - Array section analysis
- Directive based
  - OpenMP
  - Integrated with automatic methods
Global Optimization Phase

- SSA is unifying technology
- Use only SSA as program representation
- All traditional global optimizations implemented
- Every optimization preserves SSA form
- Can reapply each optimization as needed
Pro64 Extensions to SSA

- Representing aliases and indirect memory operations (Chow et al, CC 96)
- Integrated partial redundancy elimination (Chow et al, PLDI 97; Kennedy et al, CC 98, TOPLAS 99)
- Support for speculative code motion
- Register promotion via load and store placement (Lo et al, PLDI 98)
Feedback

Used throughout the compiler

- Instrumentation can be added at any stage
- Explicit instrumentation data incorporated where inserted
- Instrumentation data maintained and checked for consistency through program transformations.
Design for Debugability (DFD) and Testability (DFT)

- DFD and DFT built-in from start
- Can build with extra validity checks
- Simple option specification used to:
  - Substitute components known to be good
  - Enable/disable full components or specific optimizations
  - Invoke alternative heuristics
  - Trace individual phases
Where to Obtain Pro64 Compiler and its Support

- **SGI Source download**
  
  [http://oss.sgi.com/projects/Pro64/](http://oss.sgi.com/projects/Pro64/)

- **University of Delaware Pro64 Support Group**
  
  [http://www.capsl.udel.edu/~pro64](http://www.capsl.udel.edu/~pro64)
  
  pro64@capsl.udel.edu