Open Research Compiler (ORC): Beyond Version 1.0

Presenters:
Roy Ju (MRL, Intel Labs)
Sun Chan (MRL, Intel Labs)
Fred Chow (Key Research Inc)
Xiaobing Feng (ICT, CAS)
William Chen (ICRC, Intel Labs)

Presented at The Eleventh International Conference on Parallel Architectures and Compilation Techniques (PACT-2002)

Charlottesville, Virginia, USA
September 22, 2002

Agenda

• Overview of ORC
• Overview of Code Generation
• SSA Representation & Usage in WOPT
• Inter-procedural Analysis and Optimization (IPA)
• Tools and Demo
• Status and Activities
Overview of ORC

ORC

- Objective: provides a leading open source IPF (IA-64) compiler infrastructure to the compiler and architecture research community
- Requirements:
  - Robustness
  - Timely availability
  - Flexibility
  - Performance

* IPF for Itanium Processor Family in this presentation
What’s in ORC?

- C/C++ and Fortran compilers targeting IPF
- Based on the Pro64 (Open64) open source compiler from SGI
  - Retargeted from the MIPSpro product compiler
  - open64.sourceforge.net
- Major components:
  - Front-ends: C/C++ FE and F90 FE
  - Interprocedural analysis and optimizations (IPA)
  - Loop-nest optimizations (LNO)
  - Scalar global optimizations (WOPT)
  - Code generation (CG)
- On Linux

Flow of Open64

- C/C++: Very High WHIRL
- F90: Very Low WHIRL
- Front-ends: Un-nest calls
- Lower COMMA’s, RCOMMA’s
- Lower ARRAY’s
- Lower Complex Numbers
- Lower high level control flow
- Lower IO
- Lower bit-fields
- Lower intrinsic to calls
- All data mapped to segments
- Expose code sequences for constants and addresses
- Expose static link for nested procedures
- Code Generation
The ORC Project

- Initiated by Intel Microprocessor Research Labs (MRL)
- Joint efforts among
  - Programming Systems Lab, MRL
  - Institute of Computing Technology, Chinese Academy of Sciences
  - Intel China Research Center, MRL
- Core engineering team: 15 - 20 people
- Received support from the Open64 community and various users

The ORC Project (cont.)

- Development efforts started in Q4 2000
- ORC 1.0 released in Jan ‘02
- ORC 1.1 released in July ‘02
- Accomplishments:
  - Largely redesigned CG
  - Enhanced IPA and WOPT
  - Various enhancements to boost performance
  - Tools and other functionality
Overview of CG

What’s new in CG?

- CG has been largely redesigned from Open64
- Research infrastructure features:
  - Region-based compilation
  - Rich profiling support
  - Parameterized machine descriptions
- IPF optimizations:
  - If-conversion and predicate analysis
  - Control and data speculation with recovery code generation
  - Global instruction scheduling with resource management
- Other enhancements
Region-based Compilation

- Motivations:
  - To form a scope for optimizations
  - To control compilation time and space
- Region:
  - A directed graph
  - Connected subset of CFG
  - Acyclic
  - Single-entry-multiple-exit
    - More general than hyperblocks, treigion, etc
- Regions under hierarchical relations
  - Regions could be nested within regions
Region-based Compilation (cont.)

- Region structure can be constructed and deleted at different optimization phases
- Optimization-guiding attributes at each region
- Region formation algorithm decoupled from the region structure
  - Algorithm posted on ORC web site
  - Consider size, shape, topology, exit prob., code duplication, etc.
- Being used to support multi-threading research

Profiling Support

- Edge profiling at WHIRL in Open64 remained and extended
- New profiling support added at CG to allow various instrumentation points
- Types of profiling:
  - Edge profiling
  - Value profiling
    - Based on Calder, Feller, Eustace, “Value Profiling”, Micro-30
  - Memory Profiling
  - Can be further extended
- Important tool for limit study or to collect program statistics
Profiling Support (cont.)

- User model:
  - Instrumentation and feedback annotation at same point of compilation phase
  - Consistent optimization levels to ensure the same inputs at both instrumentation and annotation
  - Later phases maintain valid feedback information through propagation and verification
- Feedback format
  - Flexible to extend
  - Same format for every phase
- Feedback at different phases go to different feedback files – simple scheme to deal with various profiles

If-conversion

- Converts control flow (branches eliminated) to predicated instructions
- A new design to iteratively detect patterns for if-conversion candidates within regions
  - Consider critical path length, resource usage, br mis-pred. rate & penalty, # of inst., etc.
- Utilizes parallel compare instructions to reduce control dependence height
- Invoked after region formation and before loop optimization
- Displaces the hyperblock formation in Open64
**Predicate Analysis**

- Analyze relations among predicates and control flow
- Relations stored in Predicate Relation Database (PRDB)
- Query interface to PRDB: disjoint, subset/superset, complementary, sum, difference, probability, …
- PRDB can be deleted and recomputed as wish without affecting correctness
- No coupling between the if-conversion and predicate analysis
- Currently used during the construction of dependence DAG for scheduling
- Can be used for predicate-aware data flow analysis

**Global Instruction Scheduling**

- Performs on the scope of SEME regions
- A new design based on D. Berstein, M. Rodeh, “Global Instruction Scheduling for Superscalar Machines,” PLDI 91
- Builds a DAG for the given scope
- Cycle scheduling with priority function based on frequency-weighted path lengths
- Global and local scheduling share the same implementation with different scopes
- Modularizes the legality and profitability testing
Global Instruction Scheduling (cont.)

- Includes and drives many optimizations:
  - Safe speculation across basic blocks
  - Control and data speculation
  - Integrated with full resource management
    - Wide execution units, inst. template, dispersal rules
    - Interaction with micro-scheduler
  - Code motion with compensation code
  - Partial ready code motion
  - Motion with disjoint predicates

Control and Data Speculation

- Features missing in Open64 and added to ORC
- Speculative dependence edges added on DAG
- Selection of speculation candidates driven by scheduling priority function
- For a speculated load, insert \( \text{chk} \) and add DAG edges to ensure recoverability
- Includes cascaded speculation
- Future work to introduce speculation in other phases
Recovery Code Generation

- Recovery code generation decoupled from scheduling phase
  - Reduce the complexity of the scheduler
- To generate recovery code
  - Starting from the speculative load, follow flow and output dependences to re-identify speculated instructions
  - Duplicate the speculated instructions to a recovery block under the non-speculative mode
- Once a recovery block is generated, avoid changes on the speculative chain
- Allow GRA to properly color registers in recovery blocks

Parameterized Machine Model

- Motivations:
  - To centralize the architectural and micro-architectural details in a well-interfaced module
  - To facilitate the study of hardware/compiler co-design by changing machine parameters
  - To ease the porting of ORC to future generations of IPF
- Read in the (micro-)architecture parameters from KAPI (Knobsfile API) published by Intel
- Automatically generate the machine description tables in Open64
- Being ported to Itanium 2
**Micro-Scheduler**

- Manages resource constraints
  - E.g. templates, dispersal rules, FU’s, machine width, …
- Models instruction dispersal rules
- Interacts with the high-level instruction scheduler
  - Yet to be integrated with SWP
- Reorders instructions within a cycle
- Uses a finite state automata (FSA) to model the resource constraints
  - Each state represents occupied FU’s
  - State transition triggered by incoming scheduling candidate
- Can be ported to other tools as a standalone phase

**Other CG Enhancements in ORC 1.1**

- A large number of enhancements and each contributes a small gain
- Balance between RSE and register spills
  - Improved perlmbk by > 25%
- Multi-way branch synthesis
- Taming I-cache padding and code layout
- More efficient code sequence for mul, div, rem, etc.
- Restore callee-save registers in a path sensitive manner
- FU-sensitive latency for scheduling
  - E.g. 2 cycles for add (I)--> ld vs. 1 cycle for add (M)--> ld
- Scheduling across nested regions
- Scheduling for function entry and exit blocks
Other CG Enhancements in ORC 1.1 (cont.)

- Scheduling into branch-ending cycles
- Padding of nop’s to avoid pipeline flushes
- Avoid expensive loop unrolling factors
- Overhaul scheduling implementation
- Analysis of load safety to reduce the # of speculative lds
- Branch hints
- Bundle chk’s with adjacent instructions into the same cycles
- More uses of loads with gp-relative addresses
- Bug fixes and many others ….

SSA Representation and Usage in WOPT
SSA Representation and Usage in WOPT

Fred Chow
Key Research Inc.
fchow@keyresearch.com

Outline
1. Fundamental Properties of SSA
2. Global Value Numbering
3. Representing Aliasing in SSA
4. Representing indirect memory accesses in SSA
5. Restrictions on WOPT’s SSA
6. New Optimizations Enabled by this Representation
7. Generalization of SSA to Any Memory Accesses
8. Sign Extension Elimination based on SSA
What is SSA

Static Single Assignment form – only one definition allowed per variable over entire program

Main motivation – program representation with *built-in* use-def dependency information

Use-def – a unidirectional edge from each use to its definition

Use-def Dependencies in Straight-line Code

- Each use must be defined by 1 and only 1 def
- Straight-line code trivially single-assignment
- Uses-to-defs: many-to-1 mapping
- Each def dominates all its uses
Use-def Dependencies in Non-straight-line Code

Many uses to many defs
- Overhead in representation
- Hard to manage

*Can recover the good properties in straight-line code by using SSA form*

---

Factoring Operator $\phi$

Factoring – when multiple edges cross a *join* point, create a common node $\phi$ that all edges must pass through

- Number of edges reduced from 9 to 6
- A $\phi$ is regarded as def (its parameters are uses)
- Many uses to 1 def
- *Each def dominates all its uses*
  (uses in $\phi$ operands regarded at predecessors)
Rename to represent use-def edges

- No longer necessary to represent the use-def edges explicitly

\[ a_1 = \quad a_2 = \quad a_3 = \]

\[ a_4 = \phi(a_1, a_2, a_3) \]

\[ a_4 \]

\[ a_4 \]

\[ a_4 \]

Representation of Program Code in Global Optimizers

Two categories of program constructs:
1. Statements – have side effects
   1. Can be reordered only without violating dependencies
   2. “stmtrep” nodes in wopt
2. Expression trees – no side effect
   - Contain only uses
   - Can be aggressively optimized
   - “coderep” nodes in wopt

Expression trees hung from statement nodes
Value Numbering

- Technique to recognize when two expressions compute same value
- Traditionally applied on per-basic-block basis
- Value number \( vn \) is unique location in the hash table
- Leaves are given \( vn \)'s based on their unique data values
- \( vn \) of \( op(opnd0, opnd1) \) is \( \text{Hash-func}(op, opnd0, opnd1) \)

SSA enables value number to be applied globally

Global Value Numbering (GVN)

- In SSA form, all occurrences of same variable have the same value
- Each SSA variable can be given unique \( vn \)
- Need only single node to represent each def and all its uses
  - Defstmt field in node points to its defining statement
- Unique node to represent all occurrences of the same expression tree
  - E.g. \( a_1+b_1 \) and \( a_1+b_2 \) are different nodes
  - while \( a_1+3 \) and \( a_1+3 \) are same node
  - Trivial to test if two expressions are equivalent
  - Storage can be minimized
- Expression trees are now in form of DAGs made of coderep nodes
Example

Program statement:
\[ a[i] = i \]

Representing Aliasing

Hidden defs and uses of scalars due to:

- Procedure calls
- Accesses through pointers
- Partial overlaps in storage
- Raising of exceptions
- Procedure entries and exits (for non-locals)
Modelling use-defs under Aliasing

Introduce new operators for:
- MayDefs – \( \chi \) (chi)
- MayUses – \( \mu \) (not a definition)

Tag these nodes to existing program nodes
\( \chi \) factors defs at MayDefs
Single assignment property preserved

Example
a and b overlaid on top of d in memory

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

Program

a =
b =
d

SSA form

\[ \begin{align*}
    a_i & = \chi(d_i) \\
    b_i & = \chi(d_i) \\
    \mu(a_i) & = \mu(b_i) \\
    d_3 & \\
    a_i & = \mu(d_3) \\
    b_i & = \mu(d_3)
\end{align*} \]
SSA for indirectly accessed data

To be consistent, all writable storage locations should be represented in SSA form.

For occurrences of **(p+1),

Naïve approach:
1. Put p into SSA form
2. Put *(p+1) into SSA form among identical i’s
3. Put *[*(p+1)], into SSA form among identical j’s

Problems:
1. A round of SSA construction for each level of indirection
2. No clue about relationship among related indirect variables, e.g. a[i] and a[i+1]

Introducing Virtual Variables

Associate each indirect variable with an imaginary scalar variable with identical alias characteristics.

Virtual variables tagged to indirect variables via χ’s and μ’s.

One pass SSA construction for both scalar and virtual variables.

Assignment of virtual variables:
1. Related indirect accesses should share same virtual variables, e.g. *p, *(p+1)
2. Flexible:

<table>
<thead>
<tr>
<th>More virtual variables</th>
<th>Greater compilation overhead</th>
<th>Less missed optimization opportunities</th>
</tr>
</thead>
</table>

Sep 22, 2002
Virtual Variables Example

va[i] is virtual variable for accesses to array a

<table>
<thead>
<tr>
<th>Program</th>
<th>SSA form</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[i] = 3</td>
<td>a[i_1] = 3</td>
</tr>
<tr>
<td>i = i + 1</td>
<td>v^a[i]_2 = χ(v^a[i]_1)</td>
</tr>
<tr>
<td>a[i] = 4</td>
<td>a[i_2] = 4</td>
</tr>
<tr>
<td>i = i - 1</td>
<td>v^a[i]_3 = χ(v^a[i]_2)</td>
</tr>
<tr>
<td>return a[i]</td>
<td>return a[i_3]</td>
</tr>
</tbody>
</table>

Possible to determine a[i_1] and a[i_3] are same by following use-def edges of v^a[i]

GVN for Indirect Variables

Virtual variables only serve annotation purpose
Additional condition for two indirect variables with same vn to be same coderep node:
- They must be tagged with same virtual variable version
Result: indirect variables are now in SSA form (single node for its def and all its uses)
- Possible only under GVN
Honor properties of indirect variables as both expressions and variables
Work consistently for multiple levels of indirection
Example of HSSA (GVN form of SSA)

SSA form

\[ a[i_1] = a[i_1] + 1 \]

\[ \nu[i_1] = x(\nu[i_1]) \]

return \( a[i_1] \)

HSSA form

\[ \text{defstmt} \]

\[ \text{deref opnd0} \]

\[ \text{mu opnd0} \]

\[ \text{lhs} \]

\[ \text{rhs} \]

\[ \text{chi resopnd0} \]

\[ \text{istore lhs rhs} \]

\[ \text{return rhs} \]

Restrictions on WOPT's SSA

- \( \Phi \) operands must be based on same variable
  - No constants
  - No expressions
- No overlapped live ranges among different versions of the same variable

Motivation
  - Preserves utility of built-in use-defs
  - Prevent increase in register pressure
  - Trivial to translate out of SSA form
    - (just drop the \( \Phi \)'s and SSA subscripts)
  - Caught many optimization mistakes (e.g. SSA form not preserved)
Elimination of Dead Indirect Stores

void foo(void) {
    int i, a[40];
    for (i=0; i<40; i++)
        a[i] = i;
    return;
}

va[] has no use
Entire loop deleted

Elimination of Dead Indirect Stores

Straight application of SSA dead store elimination algorithm will not identify many dead indirect stores (va[] does not represent a single location)

Need to enhance algorithm by performing analysis along va[]'s use-def chain

a[i1] = 3;
va[1] = \chi(va[1])

a[i1+1] = 4;
va[2] = \chi(va[2])

return a[i1];
Copy Propagation through Indirect Variables

Based on defstmt pointer of indirect variable nodes

Replace indirect variable by r.h.s. of defining statement

Can propagate more than the closest def by following \( \nu[a] \)'s use-def chain:

1. Address expression must be identical
2. Verify non-overlap of intervening indirect stores

\[
\begin{align*}
  a[i_1] &= 3; \\
  \nu[a][i_1] &= \chi(\nu[a][i_1]) \\
  a[i_1+1] &= 4; \\
  \nu[a][i_1+1] &= \chi(\nu[a][i_1+1]) \\
  \mu(\nu[a][i_1+1]) &= \mu(\nu[a][i_1]) \\
  \text{return } a[i_1] &+ a[i_1+1];
\end{align*}
\]

Redundancy Elimination for Indirect Memory Operations

Under SSAPRE framework, indirect memory operations are treated uniformly as other expressions.

These optimizations automatically cover indirect memory operations:

- Full redundancies (common sub-expressions)
- Partial redundancies
- Loop invariant code motion

Arbitrary tree size
Arbitrary levels of indirects (indirects within indirects)
Generalization of SSA Form

- Any constructs that access memory can be represented in SSA form
  - At high levels of representation:
    1. Array aggregates
    2. Composite data structures
      - *Structs*
      - *Classes (objects)*
      - *C++ templates*
  - At low levels of representation:
    - *Bit-fields*

Can apply SSA-based optimization algorithms to them

Optimizations of structs and fields

- Large struct copies often lowered to loops making their optimization difficult
- Apply SSA optimization **before** struct lowering:
  - Dead store elimination of struct copies
  - Copy propagation for structs
- Take into account aliasing with field accesses
- Apply SSA optimization again after lowering to fields
Optimizations for struct aggregates

typedef struct ss {
    int f1;
    int f2;
    int f3;
} S;
S a;

Copy propagation and dead store elimination before struct lowering:

```c
{ S b;
  b = a;
  return b;
}
```

Optimizations for fields

Copy propagation and dead store elimination after lowering structs to fields:

```c
{ S b;
  b = a.;
  b.f2 = 99;
  return b;
}
```
Optimizations of bit-fields

- Bit-fields can be optimized more aggressively as individual fields
- SSA optimizations applied before fields are lowered to extract/deposit:
  - Less associated aliasing due to smaller footprints
  - Same representation as scalars
- After lowering to extract/deposit:
  - Promote word-wise accesses to register to minimize memory accesses
  - Redundancy elimination among masking operations

Sign and Zero Extension Optimalizations

Motivation:
1. Sign/zero extension operations needed when integer size smaller than operation size
2. Also show up when user performs:
   - Casting
   - Truncation

Especially important for Itanium:
- Only unsigned loads provided
- Mostly 64-bit operations in ISA (majority of operations in programs are 32-bit)
Sign/Zero Extension Operations

Definitions:
sext $n$ – sign bit is at bit $n-1$; all bits at position $n$ and higher set to sign bit
zext $n$ – unsigned integer of size $n$; all bits at position $n$ and higher set to zero

Example:

```
k = sext 16
short i, j, k;
k = i + j;
```

(zext if unsigned)

SSA-based Dead Code Elimination

Summary of Algorithm:
1. Assume all local variables are dead and all statements not required
2. Mark following excepted statements required:
   a. Return statements
   b. Statements with side effects (calls, indirect stores)
   c. I/O statements
3. Variables connected to required statements via computation edges are live
4. Propagate liveness backwards iteratively through:
   a. use-def edges – when a variable is live, its def statement is made required
   b. computation edges in required statements
   c. control dependences
5. Delete statements not marked required
Sign Extension Elimination Algorithm
An extension to SSA-based dead code elimination algorithm
- (perform dead code elimination simultaneously)
Use a liveness bit mask for each variable (instead of a single flag)
Use a liveness bit mask for each expression tree node
Two phases:
1. Propagate liveness of individual bits backward through use-defs, computation edges and control dependences
2. Delete operations

[Full implementation in be/opt/opt_bdce.cxx]

Propagation of bit liveness
- Top-down propagation in expression trees (from operation result to its operands)
- Based on semantics of operation, only the bits of the operand that affect the result made LIVE
- At leaves, follow use-def edges to the def statements of SSA variables
- Propagation stops when no new liveness found
Deletion of useless operations

Pass over entire program:

- Assignment statements: delete if bit mask of SSA variable has no live bit
- Other statements: delete if required flag not set
- Zero/sign extension operations: delete in either of following 2 cases:
  - Dead bits – Affected bits are dead
  - Redundant extension – Affected bits already have said values

Operations where Dead Bits Arise

- Bit-wise AND with constant: bits AND’ed with 0 are dead
- Bit-wise OR with constant: bits OR’ed with 1 are dead
- EXTRACT_BITS and COMPOSE_BITS
- “sext n (opnd)” and “zext n (opnd)”: bits of opnd higher than n are dead
- Right shifts: right bits of operand shifted out are dead
- Left shifts: left bits operand shifted out are dead
- Others
Redundant Extension Operations

Given “sext n (opnd)” or “zext n (opnd)”

Cases where the sign/zero extension can be determined redundant:
1. \textit{opnd} is small integer type with size \(\leq n\) (known values for higher bits)
2. \textit{opnd} is integer constants
3. \textit{opnd} is load of memory location of size \(\leq n\)
4. \textit{opnd} is another sign/zero extension operation with length \(\leq n\)
5. \textit{opnd} is SSA variable: following use-def to its definition and analyse its r.h.s. recursively

Summary

- Aliases in real programs can be modelled completely and concisely in SSA form
- Both direct and indirect memory accesses can be represented uniformly in SSA form using global value numbering
- SSA-based optimizations on scalar variables can be extended to indirect variables
- Benefit percolated back to scalar variables by not giving up in presence of indirect accesses
- \textit{Any} construct representing data storage can be represented in SSA form and benefits from SSA-based optimizations
Overview of IPA
InterProcedural Optimizer

Suffix of IR files between different components

Gnu C/C++

Loop Nest Opt

Scalar Global Opt

IPF Back-End

GNU IPF AS/LD
Logical Compilation Model

- .B files
- IPL
- .o files (fake)
- .G, .I files
  - analysis
  - IPA_LINK
  - optimization
- .o files (real)
  - be

InterProcedural Optimizer Processes

- Summary info gathering
  - IPL
- InterProcedural Analysis
  - IPA_LINK
- InterProcedural Optimization
Command Line View

orcc –O2 –ipa file1.c file2.c –c

orcc –O2 –ipa file1.o file2.o –o a.out

Command Line View

orcc –O2 –ipa file1.c file2.c –c
  ipl -PHASE:p:i -fB,file1.B -fo,file1.o file1.c
  ipl -PHASE:p:i -fB,file2.B -fo,file2.o file2.c

orcc –O2 –ipa file1.o file2.o –o a.out
  ipa_link –ipa –L/usr/lib /lib/crt*.o file1.o file2.o /lib/crtn.o
Command Line View

```
orcc -O2 -ipa file1.o file2.o -o a.out
ipa_link -ipa -L/usr/lib /lib/crt*.o file1.o file2.o /lib/crtn.o
orcc -c symtab.I -o symtab.o -TENV:emit_global_data=symtab.G
orcc -c -O2 -TENV:read_global_data=symtab.G 1.I -o 1.o
... ...
```

final linking with symtab.o 1.o 2.o... -o a.out

Key Observations

- Compilation model does not require users to change existing makefiles
- Output files from `ipl` (e.g. `file1.o`) are ELF files with WHIRL contents
- `ipa_link` is the linker in reality
  - Same symbol resolution and DSO dependency rule
- `symtab.G` file is the merged symbol table from all user files
- Partitioning of user code into `1.I, 2.I, ..., n.I` enables parallel make
IPL Processing

• Summary building phase
  ➤ Works on *High Whirl*
  ➤ PU is processed one at a time
  ➤ Invoked by *preopt* through *be_driver*
  ➤ Utilizes scaled down version of global optimizer to produce SSA form for flow sensitive summary info

IPL - Typical Summary Info

• Call site specific formals and actuals
• mod/ref counts of variables
• Fortran common shape
• Slice of program in SSA form (actuals)
• Array section and shape
• Call site frequency counts
• Address taken analysis
IPA_LINK Processing

• General design philosophy
  - Most optimizations are divided into two phases
    • Analysis and annotate
    • Actual transformation
  - Example: Inlining
    • Each callee is analyzed at call site
    • If decided to inline, that call-site is annotated in call graph
    • Actual inlining is done after all other analysis is done

IPA_LINK Processing

• Linker (*gnu-ld*) in reality as the driver
  - Ensure same symbol resolution rules
  - Ensure same DSO dependence rules
• Possible input file types:
  - *High Whirl* files disguise as .o files,
  - Real .o files and archives
  - .so dynamic shared objects
IPA - Analysis

- Build combined global symbol and type table
- Build call graph
- Dead function elimination
- Global symbol attribute analysis
- Array padding/splitting analysis
- Inline cost analysis and decision heuristics
- Jump function data flow solver
- Array sectioning data flow solver
- ...

IPA - Optimizations

- Perform transformation based on
  ↗️ Info collected during analysis
    - Data promotion
    - Constant propagation
    - Indirect call to direct call
    - Assigned once globals
    - ...
  ↗️ Decisions made during analysis
    - Inlining
    - Common padding and splitting
    - …
IPA – Optimization Topics

Inlining

- Each call site in call graph is considered for inline candidate
- Inline heuristic based on
  - Static call depth
  - Max and min absolute size limit
  - Hotness as a function of frequency and estimated cycle count
  - Code expansion ratio as a function of estimated caller and callee size

Data Promotion

- Symbols are of the following classes
  - Auto
  - Static
  - Common (linker allocated)
  - Extern (unallocated extern data)
  - Dglobal (initialized global data)
  - UGlobal (uninitialized global data)
- Data promotion enables more optimization opportunities
IPA – Optimization Topics

Data Promotion examples

Symbol classes can be altered using IPA

- Uglobal used in one PU and address NOT taken can be made auto
- Auto with no address taken and 0 mod/ref count is dead
- Dglobal is NOT address taken if
  - Address is never passed as an argument and
  - Address is never assigned to a global (directly or indirectly)
- Dglobal is initialized constant if
  - Mod count is 1
  - Export scope is internal

IPA – Optimization Topics

Whole Program Analysis

- Traditional WPA requires having entire program during IPA
- Without WPA
  - Global not defined in current compilation scope cannot be allocated in gp-rel area
    - Cannot ascertain true allocation of such objects
  - Fortran common cannot be splitted or padded
  - Dead function cannot be eliminated
  - Dead variable cannot be eliminated
IPA – Optimization Topics
Whole Program Analysis (WPA)

• Real programs in NT and Unix consist of
  √ User executable
  √ Dependent DSO (dynamic shared objects a.k.a. dll)

• Three obstacles to WPA
  √ Separate compilation – solved by cross file compilation system
  √ Dependency on archive libraries
  √ Dependency on DSO (such as libc.so)

IPA – Optimization Topics
WPA

• InterProcedural Optimizer must be cognizant of
  √ ABI rules
  √ Relocatable object files and archives
  √ DSO (dynamic shared objects)

• Symbol table of IPA should consists of
  √ User symbols from source code
  √ Symbols from relocatable object files
    • They will eventually become part of user code
  √ Symbols from DSOs
IPA – Optimization Topics

WPA

• WPA improves precision of analysis, but not a requirement for IPA
  ➢ Each optimization has specific export scope requirements for legality check
• Sharpen export scope with
  ➢ extensive symbol table (src, .o, .so)
  ➢ relocation information
  ➢ Data promotion to reduce export scope of symbols

WPA – Sharpening Symbol Scopes

• Dead function can be eliminated
  ➢ Promote preemptible functions to internal
• Dead variable can be eliminated
  ➢ Promote global symbols to static or auto
• Address taken analysis
  ➢ Relocation info tells whether address has been taken in a relocatable or dynamic shared object
• …
IPA – Optimization Topics

PIC

• DSO/DLL are runtime relocatable objects
  ➤ Cannot use “fix” address to access DSO objects
  ➤ Call to function defined in a DSO
    • Indirect or
    • PC relative
  ➤ Access to data object defined in a DSO
    • Indirect
    • PC relative (requires text segment copy on write)
  ➤ Text segment is shared among different processes
    • Copy on write is not desirable (no address in text segment)

• GP-rel addressing (not PIC related)
  ➤ Objects are placed in “small data area”: .sdata
  ➤ Access value through a register (gp)
  ➤ Number of objects accessible with gp-rel is restricted due to ISA
• Position Independent Code
  ➤ Indirection usually through Program Linkage Table
• Position Independent Data
  ➤ Indirection usually through Global Offset Table
• Most RISC vendors place PLT/GOT in .sdata
  ➤ IA64, Mips, Alpha, …
IPA – Optimization Topics

PIC

• **PLT/GOT** access through gp-rel addressing:
  - Entries quickly overflow **GOT** in real apps
    - Once overflowed, entire app must be recompiled
  - Function call to objects defined in DSO
    - Indirect through PLT entry – *one extra load*
    - Save/restore gp at call site (gp value is different across different DSO)
  - Data access to objects defined in DSO
    - Indirect through GOT entry – *one extra load*

PIC – Calls

**Direct Calls**

```
br.call foo
mov reg = gp
br.call rp = foo
mov gp = reg
```

**Indirect Calls**

```
mov b = reg
br.call b
mov reg0 = gp
ld8 reg1 = [reg2], 8
ld8 gp = [reg2]
mov b6 = reg1
br.call rp = b6
mov gp = reg0
```
PIC – Load Data Value

Direct load, non-pic

movl  reg = addr_var
ld8  reg1 = [reg]

gp-rel load, pic, var in small data

addl  reg = @gprel(var), gp
ld8  reg3 = [reg]

load through linkage table

addl  reg = @ltoff(var), gp
ld8  reg2 = [reg]
ld8  reg3 = [reg2]

IPA – Optimization Topics

PIC-opt

• PIC optimizations involves
  ➤ Minimize PLT/GOT entries
  ➤ Identify which object does not need to be accessed through PLT/GOT
  ➤ Identify which call sites do not need save/restore gp
IPA – Optimization Topics
PIC - wpa

• Without WPA
  ➢ All globals must be access through \textit{PLT/GOT}
    • Cannot ascertain export scope of a global
  ➢ All calls to non-static function must save/restore gp
    • Cannot ascertain preemptibility of callee
  ➢ Average loss of 5\% to 18\% performance
  ➢ Commercial database reported 10\% performance

• Use data promotion and address taken analysis technique to enable these optimizations

IPA – Optimization Topics
PIC - Data Promotion

• Symbols also falls into following \textit{export scope}:
  ➢ Internal
    • Visible only within DSO or executable
  ➢ Hidden
    • Hidden within a DSO or executable, address can be exported via pointers
  ➢ Protected
    • Non-preemptible by another object (usually in another DSO or executable)
  ➢ Preemptible
    • Can be replaced (at runtime) by another object
IPA - Optimization Topics
PIC - Data Promotion, examples

• **Internal** symbols can reside in *gp-rel* area
  - Save one extra load/store per access
  - Save one entry in GOT table
• Calling **hidden** functions does not need to
  *save/restore gp* before and after the call
  - Save one load/store or move per call site
• **Hidden** symbols does not need to have an entry in
  the *PLT/GOT* table
  - e.g. IA64 has 2**19 entry limits

• **Combining** storage class and export scope
  analysis, more aggressive symbol attribute and
  promotion can be achieved
  - Dglobal’s export scope is *internal* (from *preemptible*)
    • Defined in executable with *main*, with no addr taken
    • Not used or defined in dependent DSOs or .o’s
  - Static’s export scope is *internal* if not address taken
  - Uglobal’s is Dglobal if not used in dependent DSOs
    but defined in a .o
Debugging IPA

- IPA runs before LNO, WOPT and CG
- IPA may trigger bugs down stream due to
  - Change in IR
  - Change in symbol table attributes
- Without IPA, one can use binary search to pinpoint the source file, procedure, basic block, ...
- With IPA, excluding one procedure has global effect
  - Inlining decisions
  - Symbol scope rules
  - ...

Debugging IPA

- Debugging IPA is hard work in ORC
  - Exclude local information has global effect that disturbs entire optimization process
    - Not easily amenable to a fixed point solution
  - Is there compiler outside that solved this problem?
- Debug process usually involves
  - Pinpoint which phase causes problem
  - Pinpoint where in user source code manifests problem
  - Map problem to IR or symbol table issue
  - Root cause back to compiler code
Debugging IPA

orcc -O3 -IPA file1.o file2.o -o test

test fails at runtime

• Try –O3 (don’t do IPA)
  ➨ If test passes, problem is NOT in IPA
• Try –O0 -IPA
  ➨ If test passes, problem likely in later phases
    • Could still be due to IPA marking wrong symbol table attribute
  ➨ If test fails, problem almost certainly in IPA

Debugging IPA

-O0 -IPA passes

• Pinpoint which later phase cause problem:
  “orcc -O3 -IPA file1.o file2.o -o test –keep”
• In directory test.ipakeep, all intermediate files are saved
  ➨ 1.I, 2.I, …, n.I (IR files)
  ➨ symtab.G (merged symbol table file)
  ➨ linkopt.cmd, makefile.ipaxxx (helper files to recompile and generate object and executable files)
Debugging IPA

- **O0 -IPA passes**

- Pinpoint which .I file cause problem
  - Compile each x./ with lower optimization
  - -O0 on all .I files is the fix point
  - Process similar to debugging -O3 problems
  - Compile line is in makefile.ipaxxx

- This process can be automated
  - We have not done the work
  - Any volunteers?

---

Debugging IPA

- **O0 -IPA fails**

- Problem is most likely in IPA
- Pinpoint which phase in IPA
  - IPL
  - IPA_LINK
    - Linker
    - Ipa analysis
    - Ipa optimization
  - Could turn off optimization one at a time
    - Options in config_ipa.{cxx, h}
    - Pass options into ipl with –Wj
    - Pass options into ipa with –Wi
Because of `dlopen`, gdb requires breakpoint after all `dlopen` done before symbols from other .so visible to gdb

`ipl` (a.k.a. `be`) must be built debug

`ipl.so` must be built debug \(\text{(make BUILD_OPTIMIZE=DEBUG)}\)
Other Related IPA Analysis

• Alias analysis
  ➢ Uses Steensgaard’s points_to analysis
  ➢ A separate run after IPA
  ➢ Partitioned “alias class” is used as part of alias query by later phases
  ➢ Simple naïve implementation
    • Do not chase down heap objects
    • F90 allocatable objects are fully differentiated

Other Related IPA Analysis
Function Layout

• Cooperation between IPA, code generator and linker
  ➢ IPA decides layout order of specific functions
  ➢ Named functions output to order script file
  ➢ Functions are assigned to separate and unique text sections
  ➢ Linker reads in order-script file and put the text sections in order specified
Future Enhancements
Taker, Any?

- Alias analysis does not try to analyze heap objects
- Alias analysis is used for alias query only
  - Could use alias class result to refine intraprocedural SSA construction
    - Each alias class assigns one virtual variable
- Context sensitive mod/ref
- Class hierarchy analysis and de-virtualization
- Context sensitive alias analysis in linear (or close to) time

Tools and Demo
Developing Tools of ORC

- Tools: An Important Component of ORC
  - Information Representing Tools:
  - Debugging and Testing Tools:
- Showing Compilation Information with Graph
- Hot Path Tool

Information Representing Tools

- DaVinci: Graph Drawing Tool
- Showing Different Information
  - CFG
    - Show the effect of Opt.
  - Region Tree
  - Partition Graph of Predict Analysis
Hot path tool – hpe.pl

- **Motivation:**
  - Finding compiler performance defects through analyzing assembly code is a tedious work.
  - Analyzing assembly code on hot paths is more efficient and more effective.

- **Use:**
  - Find compiler performance highlights/defacts.
  - Compare optimization strategy of different compilers or different versions of a same compiler.

Hot path tool – hpe.pl (cont.)

- **Example:**
  - Two loops:
  - Whole procedure (Loop1)={a,c,d,f,g}
  - Loop2={b,e}
  - Hot paths
    - In loop1:
      - path = a, d freq=1
      - path = a, f, g freq=1
      - path = a, c, g freq=8
    - In loop2:
      - path = b, e freq=99
Status of ORC

ORC 1.0

- Released in Jan '02
- Major redesign of CG
- Supported optimization levels up to –O3
- Focused on general purpose applications
  - E.g. CPU2Kint, Olden, Jpeg, Mesa, …
- Good stability
- Performance:
  - ~ 5% - 10% better than GCC (2.96) at O2 and O3
  - ~ 10% better than Open64
ORC 1.1

- Released in July ’02
- Enabled IPA+inlining
- Enabled Itanium build environment
  - In addition to the cross-build environment on IA-32
- Various enhancements and bug fixes in CG, IPA, and WOPT
- Performance:
  - > 10% better than ORC 1.0 at O3+profiling
  - IPA+inlining provides additional gain

Performance Disclaimer

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/procs/perf/limits.htm or call (U.S.) 1-800-628-8686 or 1-916-356-3104.
Support

• ORC home page http://ipf-orc.sourceforge.net/
  ➤ Source code, binaries, instructions, documents, …
• Licensing: Open64 under GPL and ORC delta under BSD
• ORC mail alias:
  ➤ ipf-orc-support@lists.sourceforge.net
• Open64 mail alias:
  ➤ open64-devel@lists.sourceforge.net
• Report problems, raise questions, request info, and post contributions to the mail aliases
• The Open64 user community is organized by Prof. Gao at Univ. of Delaware and Prof. Amaral at Univ. of Alberta

Future Plan

• ORC 2.0
  ➤ To release around Jan ’03
  ➤ Focus on performance
  ➤ Major version to include all key functionality and performance results
• ORC to proliferate
  ➤ For various research: IPF, multithreading, domain-specific processors, …
• ORC will be maintained
  ➤ To drive and collect enhancements and bug fixes
• Open64/ORC user community to grow
ORC/Open64 Proliferation
(Selected Activities)

University of Delaware

• By Prof. G. Gao
• Low power/energy research
  ➤ Compiler optimizations, such as loop transformation and restructuring, SWP, register allocation, etc.
• Open64-based Kylin compiler infrastructure (kcc)
  ➤ Xscale code generator
  ➤ Kcc vs. gcc preliminary encouraging results
  ➤ Beta release this year
University of Minnesota

- By Profs P. Yew and W. Hsu
- Use ORC as an instrumentation and profiling tool
  ➤ to study alias, dependence, thread-level parallelism for speculative multithreaded architectures.
- Feed the profiling information back into ORC
  ➤ to replace and/or guide compiler analyses and optimizations.
- Use ORC to generate code to exploit speculative thread-level parallelism.

University of Alberta

- By Prof. J. N. Amaral
- ORC/Open64 for class projects
  ➤ Machine SSA, pointer-based prefetching, …
- Research projects:
  ➤ (w/ A. Douillet) on multi-alloc placement
  ➤ Later phase SSA representation
  ➤ Profile-based partial inlining
Georgia Institute of Technology

• By Prof. Krishna Palem
• Compile-time memory optimizations:
  ➢ Data remapping
  ➢ Load dependence graphs
  ➢ Cache sensitive scheduling
  ➢ Static Markovian-based data prefetching
• Design space exploration

CAS and Others in China

• Chinese Academy of Sciences
  ➢ Using ORC’s profiling framework and IPA to implement a parallel program performance analyzer (ParaVT)
  ➢ Domain-specific processors
• Tsinghua Univ.
  ➢ OpenMP
    • Explore thread-level parallelism
    • Make ORC compliant to OpenMP F90 API 1.0 (Intel's OpenMP library)
    • First release with OpenMP support in mid-2002
  ➢ Software pipelining (SWP)
    • Research on advanced SWP algorithms for multi-level loop nests and loops with branches inside
Intel

• Speculative Multi-Threading (SpMT) at ICRC
  ➢ Exploit thread-level parallelism by partitioning single-threaded apps into potentially independent threads
• Region-based optimizations intended to support multi-threading study
• Intel Barcelona Research Center led by Antonio Gonzalez also uses ORC for their SpMT study
• JIT leverages the ORC micro-scheduler

Many More ...

• Tensilica (extensible embedded processor)
• ST Microelectronics (embedded processors, etc.)
• Cognigine Corp.
  ➢ Variable ISA, PACT 2002
• Universiteit Gent, Belgium
  ➢ Reuse distance-based cache hint selection, Euro-Par 02
• Univ. of Maryland (Prof. Barua)
  ➢ Optimal scheduling
• Rice University
  ➢ Restructuring optimizer for co-array Fortran
• … (other universities and companies)
Contributions and Acknowledgements

• Institute of Computing Technology, Chinese Academy of Sciences
• Programming Systems Lab, Intel Labs
• Intel China Research Center, Intel Labs
• Pro64 developers
• Many ORC/Open64 users