Micro-35 Tutorial

Open Research Compiler (ORC) 2.0 and Tuning Performance on Itanium

Presenters:

Roy Ju, Sun Chan, Tin-Fook Ngai
(MRL, Intel Labs)
Chengyong Wu, Yunzhao Lu, Junchao Zhang
(ICT, CAS)

Presented at the 35th International Symposium on Microarchitecture

(Micro-35)

Istanbul, Turkey
November 19, 2002

intel®

Micro-35 Tutorial
Open Research Compiler (ORC) 2.0 and Tuning Performance on Itanium

Presenters:
Roy Ju, Sun Chan, Tin-Fook Ngai
(MRL, Intel Labs)
Chengyong Wu, Yunzhao Lu, Junchao Zhang
(ICT, CAS)

Presented at the 35th International Symposium on Microarchitecture (Micro-35)

Istanbul, Turkey
November 19, 2002

Agenda

- Overview of ORC
- IR – WHIRL and Optimizations
- Overview of CG
- Performance Features in ORC 2.0
- Performance Analysis Tools and Experience
- Demo
- ORC for Speculative Multithreading
- Research Activities and Plan

Overview of ORC

ORC

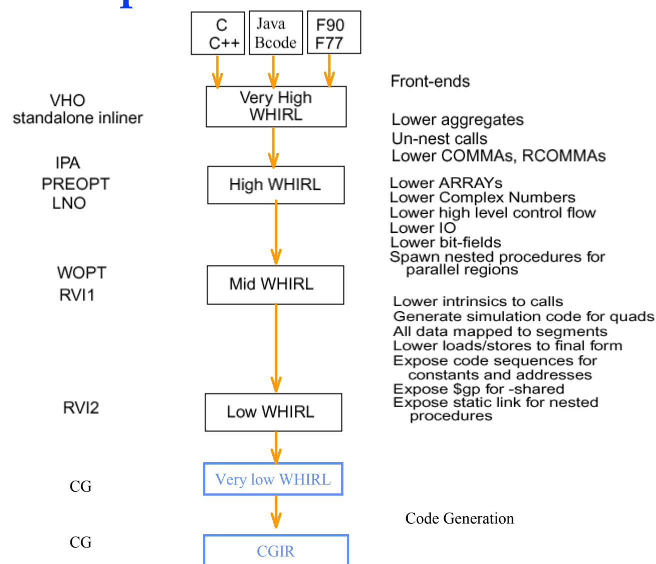
- Objective: provides a leading open source IPF (IA-64) compiler infrastructure to the compiler and architecture research community
- Requirements:
  - Robustness
  - Timely availability
  - Flexibility
  - Performance

* 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



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

intel

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
- ORC 2.0 planned early '03

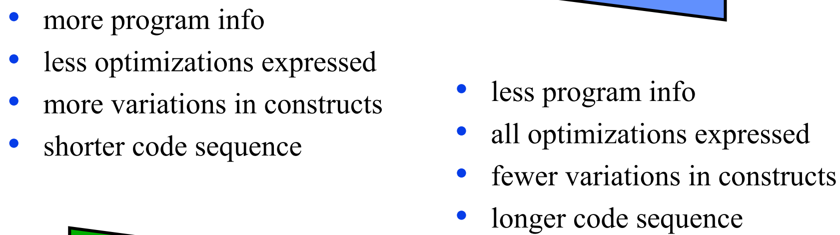
intel

IR – WHIRL and Optimizations

Role of IR in Compiler

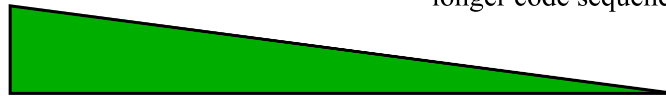
- Bridge semantic gap from source language to machine instructions
- Same IR
  - multiple languages
  - multiple targets
- Interfaces between compiler components
  - Well defined input and output
- Medium for IR-based optimizations

Levels of IR Representation



- more program info
- less optimizations expressed
- more variations in constructs
- shorter code sequence

- less program info
- all optimizations expressed
- fewer variations in constructs
- longer code sequence



Impact of IR on Optimizations

- **Ease of implementation**
  - Info is right at hand
  - Canonical form reduces implementation complexity
- **Ease of debugging**
  - Ability to verify consistency
  - Well defined input/output
- **Compile time efficiency**
  - No need for extra info to help analysis
- **Quality of optimizations performed**
  - Well formed IR prevents need to pattern match
  - Canonical form improves chance to optimize
- **Type of optimizations possible**
  - No lost of information when optimization done at the right level

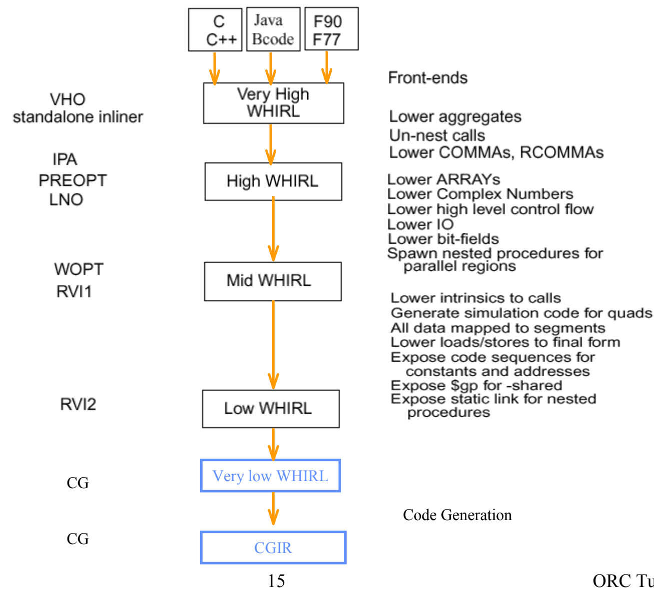
WHIRL

- An Intermediate Representation of a given user program
- Multiple levels of abstraction
  - Each level semantically the same
  - Less information at lower level
  - Allow optimization to be perform at most appropriate representation
- Hierarchical
  - lower level subset of the higher form
- Symbol table
  - User defined symbols
  - User defined types

WHIRL

- Continuous lowering of IR through each component
  - Call ***lowerer*** to translate to lower level
- Supporting components to
  - minimize variations in IR form
  - Avoid duplication of optimization/analysis functionalities
    - ***Simplifier***
    - ***Preopt***
- High WHIRL and up can be raised back to C/Fortran

Flow of IR



Tools for WHIRL

- *ir_b2a*
  - Dumps IR in pre/post order form
- *whirl2c*, *whirl2f*
  - Dumps IR in high level C/Fortran form
  - May not be re-compilable if input is low-WHIRL
- Numerous verifier and self-checkers throughout compiler
  - WHIRL consistency
  - Symbol table and WHIRL consistency

Tools for WHIRL

- To get IR file after each component:

.N (after LNO or Preopt)	-PHASE:c=off:w=off
.O (after Wopt)	-PHASE:c=off
.B (after Frontend)	-keep
.I (after IPA or Inlining)	-keep

Normalization – PreOpt

- Runs before *IPA*, *LNO*, *WOPT*
- Prepares and clean up code
  - Expose more opportunities
  - Minimize variations in IR forms
  - Collect supporting info for later phases
    - Alias info
    - Loop info: trip count, body, induction expression, …
    - SSA slices for IPA summary info

Simplification (Simplifier)

- Callable at anytime during compilation when WHIRL is the IR (i.e. before code generator)
- Works on expressions
  - Constant folding
  - Reassociation
  - Simple strength reduction
  - Canonicalization
- Compiler writer is guaranteed a well formed expression for any expression he generates

intel®

Optimization Topics
Switch Case Optimization

- Simple switch cases
  - Use cascaded if-then-else sequences
- Large switch cases with skewed frequent cases
  - Hoist most frequent cases outside of switch case
- IR – Very High WHIRL

  SWITCH, CASEGOTO
- Implementation advantages
  - No need for control flow analysis
  - No need to update control flow structure

intel®

Optimization Topics
Array Dependency Analysis

- Dependency of subscripted variables inside loops
- Model as a system of integer programming
  - Canonical form for indices to start at 1
  - Loops must be well form do loops (vs while, repeat)
- IR – ***ARRAYEXP*** and ***DO_LOOP***
- High WHIRL
- Implementation advantages
  - No need for do_loop recognition in that component
  - Pointer access and array access produces same result

Optimization Topics
Data Flow Problems

- Operate over program's control structure and flow of data
  - Simple model of program transfer points
  - Uniform treatment of memory accesses
- IR – Mid WHIRL
  - Explicit control transfer
  - Full address expression form

  X [I – 1] [J – 1] ➔ &X + (J – 1) * sizeof(X) – (I – 1)
- Implementation advantages
  - Simpler algorithms for various control dependency data structure
  - No need to update loop structures after transformation

Optimization Topics
Memory Disambiguation

- Dependency of given pair of memory accesses
- Memory access variations
  - Direct
  - Explicitly indirect
    - Pointer dereference
  - Implicitly indirect
    - Structure field access
    - ABI requirement (access through global offset table)
- Indirect access manifests as:

  &base + offset

Optimization Topics
Memory Disambiguation

VH WHIRL	Mid WHIRL	Low WHIRL	Machine Inst
ldid v1	*ldid v1*	*lda &v1*	*adds r1=sp,16*
		iload	*ld8 r2=[r1]*

- Address taken detrimental to precision
  - Indirect access interfere with address taken analysis
  - Reference parameter access after inlining

    e.g. *&p -> f* best if seen as *p.f*
- Performed when IR is High WHIRL:

  LDID object

  where object is scalar, struct, field in struct, …
- Implementation advantages
  - More precise address taken analysis
  - Overlaps in memory is exactly represented in same IR node

Optimization Topics
Structure Field Reorder

- Reorders layout of fields inside an aggregate object
- Effectively manipulates ***base_address + offset***
- Implementation considerations
  - Base address and/or field offset in disguised form
    - User code written as such
    - Results of optimization (e.g. CSE of offset arithmetic)
  - ***sizeof*** and ***offset*** operators in user code
  - New order must be reflected consistently throughout user program
    - Layout assumptions outside of current compilation unit
    - Object preemption rules

Optimization Topics
Structure Field Reorder

- IR – Memory chunk type (M) in memory access
  e.g. ***MISTORE*** – indirect store of memory chunk
- Field ID provided in IR when access is expressed as individual field operation
- High WHIRL
- Symbol table contains ***FLD_TAB***
  - Byte offset, bit offset
  - Other info such as equivalence to another object

Optimization Topics
Structure Field Reorder

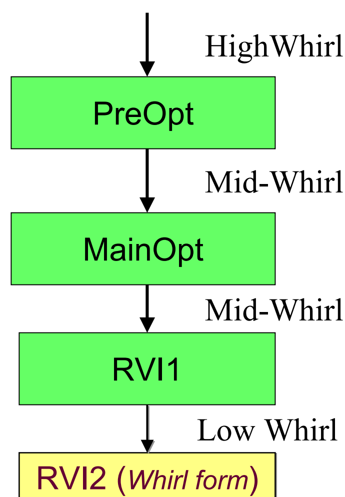
- Implementation advantages
  - No need to hunter for base address
  - No need to distinguish between offset and constant in IR
  - No restriction on type of optimizations done ahead
    - Those optimizations must guarantee consistency though
  - Not solved issues
    - Need ***sizeof*** and ***offset*** operators

Major Flow of Global Optimizer

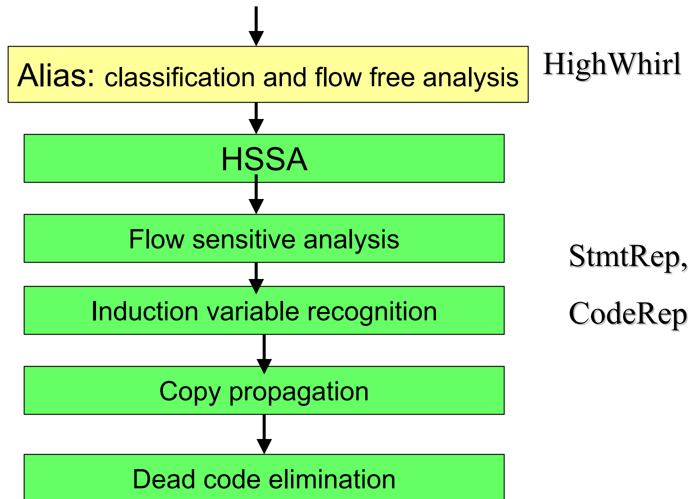
IR inside Global Optimizer (Wopt)

- WHIRL is **NOT** an SSA language
- WHIRL is translated into HSSA form inside Wopt
- HSSA is a SSA form, extended to include
  - Array and indirect memory references
  - Alias (mod/ref) info as part of the IR
  - Representation in
    - expression trees (***coderep***)
    - Statements (***stmtrep***)

Flow of Global Optimizer



Major Components of Preopt

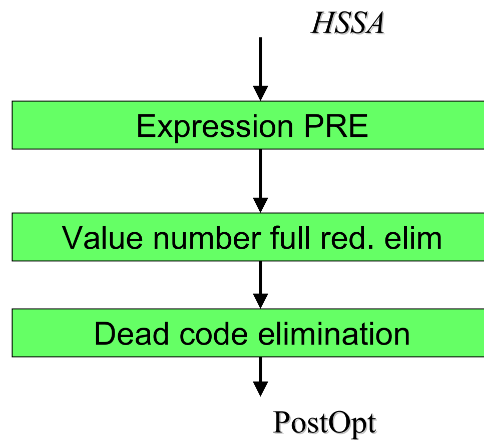


HighWhirl

StmtRep,
CodeRep

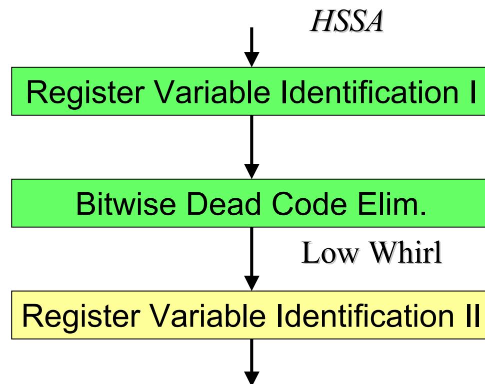
Major components of MainOpt

HSSA



PostOpt

Major components of PostOpt



Detailed IR Lowering

- Various specific lowering of expression occur during MainOpt
  - To take better advantage of optimization at hand
  - Extract/Deposit in replace of LDBITS
  - Signed-ness of load exposed
  - ***&p->i*** folded to ***p.I***
    - Redo address taken analysis

Value Number Full Redundancy Elim.

- Some redundancy cannot be eliminated by PRE (and vice versa)
- Fast mechanism to deal with important cases where PRE misses
  - Induction coalescing for some important loops
- Experimental phase to evaluate and explore new ideas (turned on by default)

Partial Redundancy Elimination

- Subsumes most major classical optimizations
  - Common subexpression
  - Loop invariant code motion
  - Strength reduction
  - Code hoisting
  - Redundancy elimination (partial and full)
  - Register Promotion (Register variable identification)
  - Partial dead store elimination

SSA based PRE (SSAPRE)

- Result also in SSA form
- Advantage over bit vector approach
  - Performs on local and global level in one shot
  - Demand driven enable
    - Prioritized worklist
    - Re-optimize previously optimized item for secondary effect
  - Sparse evaluation, demand driven plus result in SSA
    - Optimization can stop at any work item
      - Suitable for use in time sensitive environment (JIT)
    - Enable automatic debug/triage tool for the compiler
  - Optimized code debugging becomes annotation problem
    - Code motion done in one shot
  - Almost linear time algorithm

intel

Expression PRE (EPRE)

- Works on expression level
- Indirect memory operations are treated as other expressions
  - Covers indirect memory operations
  - Arbitrary tree size
  - Arbitrary levels of indirects
- Easy to expand to other optimizations
  - Array bounds check elimination
  - Speculative code motion
  - …

intel

Register Variable Identification (Register Promotion)

- Preparation phase for register allocation
- Problem formulated as PRE problem with lifetime optimal placement solution
- Advantage over non-PRE approach
  - Placement not random, but provably lifetime optimal
  - Easily extensible to resource optimal also
- Advantage over non-SSA approach
  - Demand driven – controllable compile time
  - Enable automatic debug/triage tool

Register Variable Identification (RVI)

- ***3 separate phases of RVI - I***
  - Shares same code as EPRE for 2 phases
    - Software reuse
  - Local register promotion
    - Simple scan over all locals
  - LPRE
    - Works on loads of variables
  - SPRE
    - Works on stores of variables

Register Variable Identification II

- Works on Low-Whirl
  - Binary level details exposed
    - bss segment, gp relative, …
  - IR closer to machine form
    - Load of variable value becomes
      - Calculate address
      - Load content through address
- Solves data flow equation over CFG
- Remnants of pre-SSAPRE implementation
  - Takers, any?

intel

Overview of CG

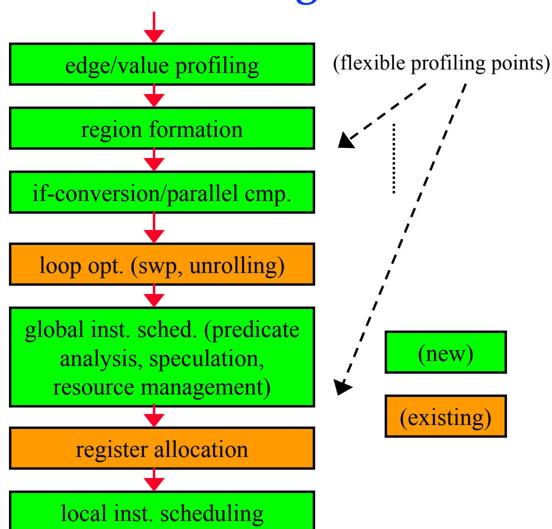
intel

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

intel

Major Phase Ordering in CG



intel

Region-based Compilation

- Motivations:
  - To form profitable scopes 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, treegion, 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 in CG to allow various instrumentation points
- Types of profiling:
  - Edge profiling, value profiling, memory profiling, …
- Important for limit study or collecting program statistics
- User model:
  - Instrumentation and feedback annotation at the same compilation phase
  - Later phases maintain valid feedback information through propagation and verification

intel

If-conversion

- Converts control flow (branches eliminated) to predicated instructions
- A simple 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

intel

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
- Cycle scheduling with priority function based on frequency-weighted path lengths
- Full resource management by interacting with a micro-scheduler
- Modularizes the legality and profitability checking
- Includes
  - Safe speculation across basic blocks
  - Control and data speculation
  - Code motion with compensation code
  - Partial ready code motion
  - Motion of predicated instructions
  - …

Control and Data Speculation

- Features missing in Open64 but added in ORC
- Speculative dependence edges added on DAG
- Selection of speculation candidates driven by scheduling priority function
- For a speculated load, insert *chk* and add DAG edges to ensure recoverability
- Recovery code generation decoupled from scheduling phase
- Starting from the speculative load, follow flow and output dependences to re-identify speculated instructions
- 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

intel®

Performance Features in ORC 2.0

intel®

Performance Features

- Enable optimizations on object fields
  - Improved *eon* by ~40%
- Balance between RSE and register spills
  - Improved *perlbmk* by > 30 %
- Memory optimizations
  - Stride prefetching
  - Reordering of hot/cold struct fields
  - Conversions to memset/memcpy
- Tuning the cost model for function inlining
- IPA-enabled optimizations
  - De-virtualization, more uses of gp-rel, procedure reordering,
- Aliasing enhancements: type-based disambiguation

Performance Features (cont.)

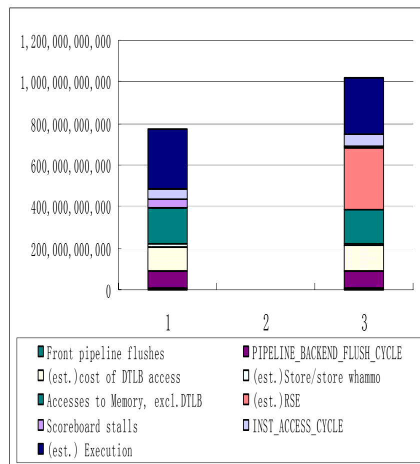
- Enhancements of instruction scheduling (>3% overall)
  - A new, more modular implementation
  - Tuning of scheduling heuristics to enable more speculation
  - P-ready code motion
  - Across nested regions; branch “delay slots”; entry and exit blocks
- Efficient code expansion
  - Mul/div/rem
  - Avoid expensive loop unrolling factors
  - Boolean expressions
- Multiway branch synthesis
- Branch hints

Performance Features (cont.)

- Restore callee-save registers in a path sensitive manner
- Analysis of load safety to reduce the # of speculative lds
- Bundle chk's with adjacent instructions into the same cycles
- Micro-architectural features
  - Padding of nop's to avoid pipeline flushes
  - Taming I-cache padding and code layout
  - FU-sensitive latency for scheduling
    - E.g. 2 cycles for add (I)-> ld vs. 1 cycle for add (M)-> ld
- …
- A large number of enhancements and each contributes a small gain (often < 0.5%)

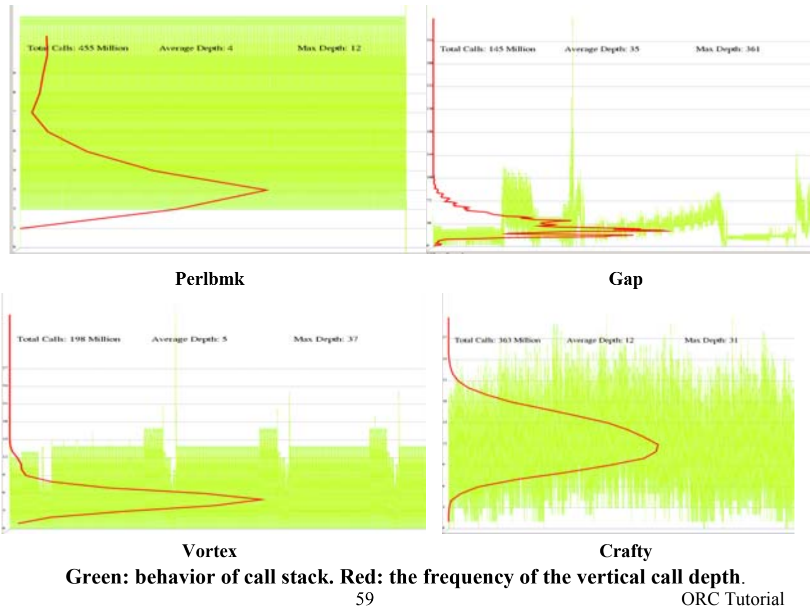
RSE (Register Stack Engine) Problem in *Perlbmk*

- RSE Problem in *perlbmk*
  - *regmatch*: self recursion and using > 96 registers
  - Excessive stalls due to RSE
  - Average call depth: 4
- Current solution
  - Reduce register usage with spills
  - Live ranges for stacked registers ordered
  - Current usage of stacked registers in *regmatch*: 27



Performance after and before tuning

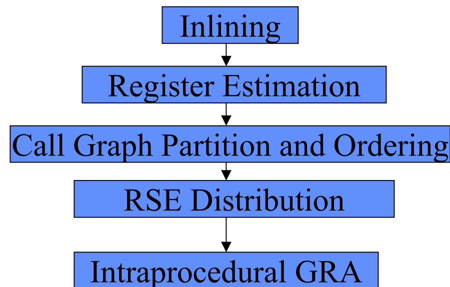
Call Stack Graph



Green: behavior of call stack. Red: the frequency of the vertical call depth.

Generalized RSE Solution
(In progress by Yang Liu @ ICT)

- RSE cost related to those paths in call graph with high invocation frequency and high register pressure
- *Formulation*: given a fixed number of registers, distribute them to different functions to minimize the total spill cost and RSE overflow cost
- *Framework*:



Optimizations for C++ Programs
(by Kai Yu Chen @ ICRC)

- Some characteristics of C++ programs
  - Small member functions, structure copy, virtual function calls, abstract data class, template functions, …
- Challenges for *eon*
  - Insufficient inlining
    - Inappropriate symbol attribute and inline heuristic
  - Overhead introduced by inlining
    - Excessive memory copies of structure fields
  - Traditional optimizations techniques less effective to structures and fields
    - Copy propagation, DCE, folding of field accesses, IVR, …
    - Delay the lowering of IR for large structures
    - Flatten well organized small structures early

Optimizations for Structures and Fields - Examples

- Remove unnecessary structure copies through copy propagation and dead code elimination

```
S2 = S1;
S3 = S2;        ⇨    return S1;
return S3;
```

- Remove unnecessary field accesses

```
S1.data = i;
S2 = S1;        ⇨    return i;
j = S2.data;
return j;
```

Measurement for Abstract Penalty

Stepanov's benchmark for measuring abstract penalty

compiler \ data	Abstraction Penalty	Total Run Time (s)
Gcc (-O3, 2.96)	1.37	11.29
Orcc (Note #1)	48.10	63.67
Orcc (Note #2)	28.62	45.97
Orcc (Note #3)	6.82	7.90
Orcc (Note #4)	4.24	4.76
Orcc (Note #5)	0.77	0.88

Note: 1. ORC 1.0
2. Enable copy propagation and DSE for struct's
3. Enable inlining of template functions
4. Disable prefetch and enable CPROP for FP PREG inside loops
5. Enable indirect memory access folding and flatten struct for IVR

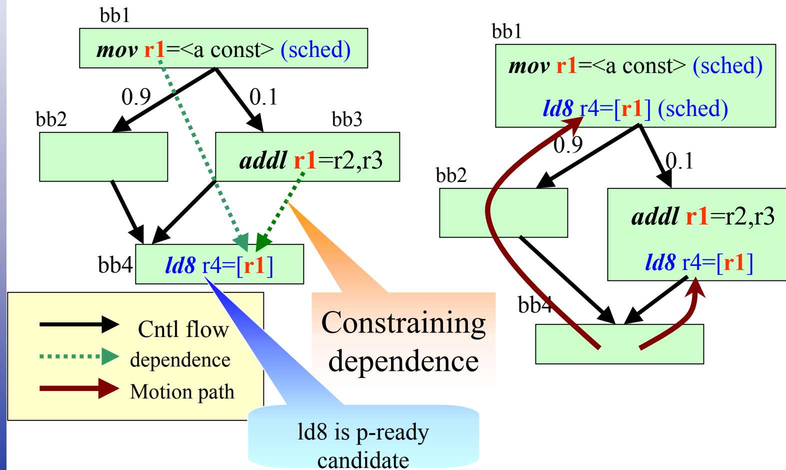
intel

Partial Ready Code Motion
(by Shuxin Yang @ ICT)

- *Problem*: scheduling dependences ready on one path but not on another
  - Lost scheduling opportunities
- To schedule aggressively on hot paths by leaving compensation copies on cold, yet ready paths
- Extended from existing upward code motion
  - Identify a cutting set to find the nodes collectively post-dominating the dependences not ready
  - Place compensation copies on these nodes

intel

Partial Ready Code Motion - Example



Enable and Tune Function Inlining

(by Peng Zhao @ U. of Alberta and by Ge Gan & Liping Xue @ ICT)

- Inlining: enables more optimization opportunities
- Guided by profiling feedback
  - A problem: functions never invoked slipping through the cost checking and frequently getting inlined
- Taking into account: invocation frequency, sizes of caller and callee, estimated cycles in callee,
- Lowered hotness threshold to enable more inlining
- Currently ~10% improvement from IPA/inlining and expecting more

Memory Optimizations - Stride Prefetching
(by JiaJun Wu & Xiaobing Feng @ ICT)

- ~ 40% of all cycles in SPEC CInt2000 due to data stalls (cache and DTLB)
- ~80% of the cycles in *mcf* due to data stalls
- Prefetching can reduce data stalls
  - But only stride-based prefetching has been effective and only to fp apps
- Fixed strides observed in some integer apps due to the patterns in allocation and usage of data objects
- Stride prefetching guided by profiling feedback
  - Based on the work by Y. Wu, PLDI ’02
  - Expecting large gains in *mcf* and some gains in *gap* and *parser*

Memory Opt. – Reordering of Structure Fields
(by Li Chen @ICT & William Chen @ ICRC)

- Reorder hot and cold fields in objects to improve the spatial locality in cache
- Many considerations on legality and profitability
- *Example*: hottest loop in *mcf*

```
While (arc) {
    .. node->time
    .. node->potential
    ...
    .. node->mark
}
```

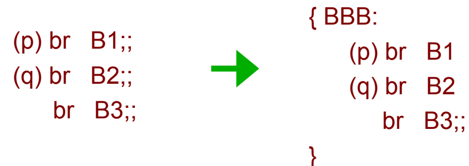
```
struct node {
  long number;
  char *ident;
  struct node *pred, …
  …
  cost_t potential;
  flow_t flow;
  size_t mark;
  long time;
}
```

→

```
struct node {
  size_t mark;
  long time;
  cost_t potential;
  long number;
  char *ident;
  struct node *pred, …
  …
  flow_t flow;
}
```

Multiway Branch Synthesis (by Lixia Liu @ ICRC)

- Utilize the multiway branch feature on IPF
- Issue more than one branch per cycle
  - E.g. MBB, BBB



- A separate synthesis phase after scheduling
- Branch target has to be bundle aligned

Performance Analysis Tools and Experience

Performance Analysis Tools on IPF

- Tools: an important part of ORC
  - Automatic testing tools
  - Cycle counting tools: static and dynamic
  - pfmon : to access the IA-64 PMU on Linux
  - VTune: perfromance analyzer from Intel
  - Visualization tool based on daVinci
  - Hot path enumeration tool

Automatic Testing Tools

- Support various testing (correctness, performance, checkin, regression, etc.)
- Automatic start (crontab settings) and update the newest source from a version control system (e.g. CVS)
- Generate reports with comprehensive information.
- Customizable on many aspects:
  - Optimization levels: O2/O3, +profiling, +IPA, peak mode.
  - Benchmarks and compilers used.
  - Compilation modes (cross/native) and running platforms.

Cycle Counting Tools

- Count cycles caused by stop bits and latencies
  - Cycles due to dynamic events, e.g. cache misses, not counted.
- Count cycles of pre-selected hot functions.
- Generate reports of comparisons with history data.
- Static cycle counting
  - Based on annotations in assembly code, e.g. frequency weighted cycles of each basic block.
  - Need pre-generated feedback information.
- Dynamic cycle counting
  - Need additional implementation to
    - Insert the definitions of counters into Symtab.
    - Find unused registers for instrumentation instructions.
  - Frequency counted at run time.

Sample Assembly Code

```
// Block: 45 Pred: 44 51 Succ: 46 51
// Freq  22459.0 (feedback)  Prob 0.43924  0.56076
// <lentry>
// BB:45 cycle count: 7
.Lt_0_146:  // 0x6b0
 { .mmi
    .loc    1   578 0
          adds r35=8,r32 ;;              // [0:578]
          ld8 r35=[r35]                  // [1:578] id:232
          nop.i 0 ;;                     // [:0]
 } { .mmi
          adds r35=48,r35 ;;             // [3:578]
          ld8 r35=[r35]                  // [4:578] id:233
          nop.i 0 ;;                     // [:0]
 } { .mib
          cmp.eq p0,p11=0,r35            // [6:578]
          nop.i 0                        // [:0]
    (p11) br.cond.dpnt.few .Lt_0_149 ;;      // [6:578]
 }

// Block: 51 Pred: 50 45 Succ: 52 45
// Freq  22406.0 (feedback)  Prob 0.81893  0.18107
//<loop> Part of loop body line 578, head labeled .Lt_0_146
// BB:51 cycle count: 4
.Lt_0_147:  // 0x6e0
 { .mmi
    .loc    1   641 0
          adds r32=16,r32 ;;             // [0:641]
          ld8 r32=[r32]                  // [1:641] id:237
          nop.i 0 ;;                     // [:0]
 } { .mib
          cmp.ne p12,p0=0,r32            // [3:641]
          nop.i 0                        // [:0]
    (p12) br.cond.dpnt.few .Lt_0_146 ;;      // [3:641]
 }
```

Sample Report Generated by the Tool

Comparison with CVS			
THIS_TIME: 2-Jul-2002 vs. LAST_TIME: 1-Jul-2002			
PU_name	THIS TIME	LAST TIME	DELTA
bzip2:generateMTFValues	4.554E+09	4.554E+09	0.000E+00
bzip2:sendMTFValues	8.250E+08	8.250E+08	0.000E+00
crafty:Evaluate	4.564E+09	4.699E+09	1.355E+08
crafty:FirstOne	1.190E+09	1.190E+09	0.000E+00
crafty:Attacked	8.506E+08	8.884E+08	3.785E+07
gap:CollectGarb	2.302E+08	2.302E+08	0.000E+00
gzip:longest_match	7.222E+09	7.139E+09	-8.334E+07
gzip:deflate	2.697E+09	2.795E+09	9.885E+07
gzip:inflate_codes	3.601E+09	3.601E+09	0.000E+00
. . .			
Result:			
TOTAL:17 PASSED:17 FAILED:0			
Degraded:1 Improved:5 Unchanged:11			

Performance Monitoring and pfmon

- Performance monitoring features on Itanium
  - A suite of performance monitoring registers
  - More than 150 events
  - Advanced features, such as “sampling”
- Methods of performance analysis:
  - Work load characterization (Event rate monitoring and Cycle breakdown)
  - Profiling (PC sampling, EAR sampling, BTB)
- pfmon: monitors runtime behavior of unmodified binaries

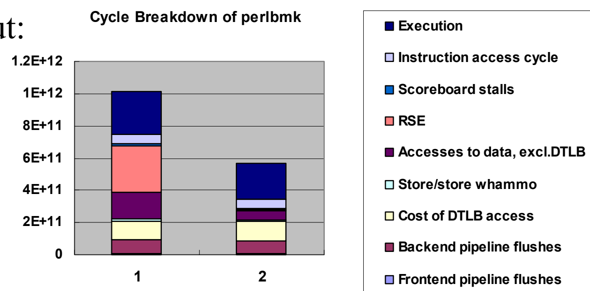
pfmon – Event Counting

- Count the occurrences of important events
- Example: failed data speculations
  - *pfmon –u –e ALAT_INST_CHKA_LDC.ALL – foo …*
  - *pfmon –u –e ALAT_INST_FAILED_CHKA_LDC.ALL orcc_foo …*
- Output:

orcc_	ALAT_INST_CHKA_LDC.ALL	ALAT_INST_FAILED_CHKA_LDC.ALL	failed/all
bzip2	1929041132	325549130	16.88%
crafty	494898879	56087	0.01%
eon	4100283083	328561109	8.01%
gap	5218362450	283104235	5.43%
gcc	334475307	23143799	6.92%
gzip	316462073	314647	0.10%
mcf	278918345	3632297	1.30%
parser	7361718884	755482308	10.26%
perlbmk	2368043225	3676097	0.16%
twolf	10867872791	637333338	5.86%
vortex	4541508419	32142975	0.71%
vpr	1385861255	4956940	0.36%

pfmon – Cycle Breakdown

- Attributes a reason for every cycle spent.
- Need multiple runs to make a complete breakdown
  - *pfmon -k -u –e CPU_CYCLES, EXECUTION_CYCLE, PIPELINE_ALL_FLUSH_CYCLE, DTLB_MISSES --outfile=$res_file*
- Output:



pfmon – Event based Sampling

- Relating performance problems back to source code.
- Example: D-cache misses causing latencies > 16 cycles
  - *pfmon --dear-smpl-rate=1000 -e DATA_EAR_CACHE_LAT16 --smpl-file=sample -- ls /usr*
- Output:

```
Entry 0 PID:9239 CPU:0 STAMP:0xe468a2709334 IIP:0x2000000000013750
          PMD OVFL: DATA_EAR_CACHE_LAT16(4)
          PMD2  : 0x2000000000070dd9
          PMD3  : 0x0000000000000011 , Latency 17
          PMD17 : 0x2000000000024f30 (slot 0) valid=Y

Entry 1 PID:9239 CPU:0 STAMP:0xe468a270a75d IIP:0x2000000000013580
          PMD OVFL: DATA_EAR_CACHE_LAT16(4)
          PMD2  : 0x200000000005ace0
          PMD3  : 0x0000000000000014 , Latency 20
          PMD17 : 0x2000000000013760 (slot 0) valid=Y
```

Intel VTune Performance Analyzer

- Collect performance data on application and system
  - Hotspots, critical functions
  - Processor events, e.g. cache misses
- Various collectors with display in graph or table
  - Counter monitor
  - Sampling
  - Call graph
- Use collected info to identify performance bottlenecks
- Experience in finding the deficiencies in compiler
  - Cannot modify apps' source code
  - Performance bottlenecks may be legitimate
  - Still need much work to map to an root cause

VTune – Sample Display

VTune(TM) Performance Analyzer 6.0 - [Sampling Hotspots - [Sampling Results [143.183.134.81] - Thu May 02 15:15:27 2002]]

File Edit View Activity Configure Window Help

Copy of gap-orc-peak-refinput-PFMON-ref input

Process Thread Module Hotspot Source

Group by: Function

Function	CPU Cycles %(9)	CPU Cycles Samples(9)	Clockticks %(9)
deflate	23.42	814	23.03
longest_match	18.87	656	18.90
inflate_codes	16.74	582	16.99
CD	13.29	462	13.70
NP	13.03	453	12.89
BD	4.06	141	4.22
updcrc	3.77	131	3.58
inflate_stored	3.45	120	3.41
HP	1.27	44	1.56
huft_build	0.60	21	0.43
copy_block	0.40	14	0.35
GP	0.40	14	0.46
DP	0.26	9	0.26
JP	0.17	6	0.06
inflate_dynamic	0.09	3	0.06
IP	0.06	2	0.03
send_bits	0.06	2	0.06

Selection Summary

Events	Total
CPU Cycles(9)	814.00
Instruction Access Cycles(9)	241.00
Scoreboard Dependency Cycles(9)	908.00
Scoreboard Dependency Dispersal Break ...	720.00
Hardware Page Walker Inserts into the DT...	218.00
L2 Flush Details(9)	229.00
Pipeline Flush(9)	140.00
Unstalled Back-end Cycles(9)	1,019...
Combination of Pipeline Flush Cycles caus...	470.00
Combination of Pipeline Flush Cycles caus...	486.00
Combined Memory Stall Cycles(9)	859.00
DTLB Misses(9)	230.00
Clockticks(9)	797.00
CPU Cycles events(9)	650,5...
CPU Cycles %(9)	23.42
Instruction Access Cycles events(9)	1,846...
Instruction Access Cycles %(9)	19.82
Scoreboard Dependency Cycles events(9)	39,03...

Legend

Activity ID	Activity Result	Total Samples	Duration	Machine Name	Processor
9	Sampling Results [143.183.134.81] - Thu May 02 15:15:27 2002	22908	5.557s	i64lnx2	Intel® Itanium(TM) proces:
9	Sampling Results [143.183.134.81] - Thu May 02 15:15:27 2002	22833	4.378s	i64lnx2	Intel® Itanium(TM) proces:
9	Sampling Results [143.183.134.81] - Thu May 02 15:15:27 2002	28227	6.002s	i64lnx2	Intel® Itanium(TM) proces:
9	Sampling Results [143.183.134.81] - Thu May 02 15:15:27 2002	4739	4.764s	i64lnx2	Intel® Itanium(TM) proces:

36 items, 13 events, 1 item(s) selected. All Process All Thread 1 Module

Output

General

For Help, press F1

Visualization Tool

- Based on daVinci: an X-Window visualization tool from b-novative, a spin-off of University of Bremen.
- ORC can communicate with daVinci and support interactive examination of graph-based data structures.
- Various internal data structures visualized in ORC:
  - Global and regional CFG, region tree
  - Regional and local (basic block) dependence DAG
  - Predicate partition graph
  - CFG visualization tool on assembly code

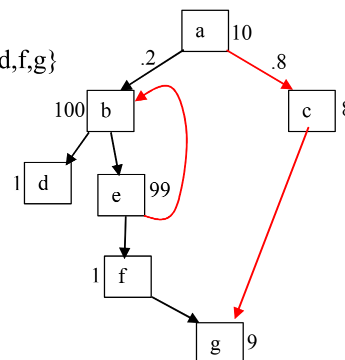
Hot Path Enumeration Tool – hpe.pl

- Motivation:
  - Analyzing assembly code of large PUs is tedious.
  - Focusing on hot paths only is more effective.
- Uses of the tool:
  - Find performance hot spots / defects.
  - Comparison between different compilers.
  - Comparison between different versions of same compiler.

intel.

Hot Path Enumeration – an Example

- 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



intel.

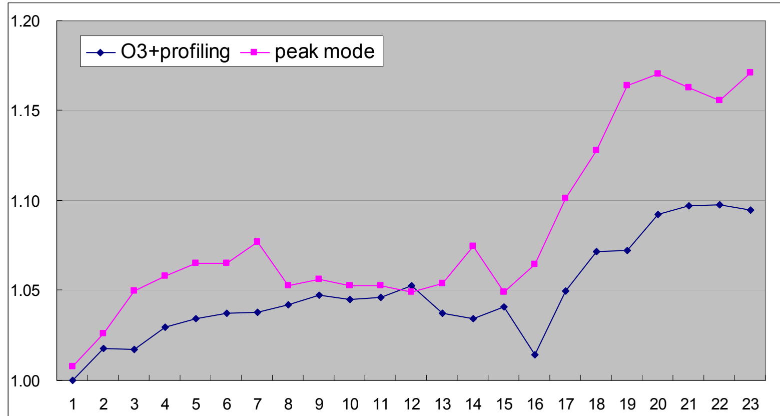
Driving the Performance of ORC

- Avoid performance regression
  - Checkin testing and criteria
  - Fast investigation of performance degradations.
- Identify new opportunities through continuous performance analysis
  - Inspection of assembly code for hot paths in hot functions
  - Comparisons between different compilers
  - Understanding the performance impact of various enhancements.

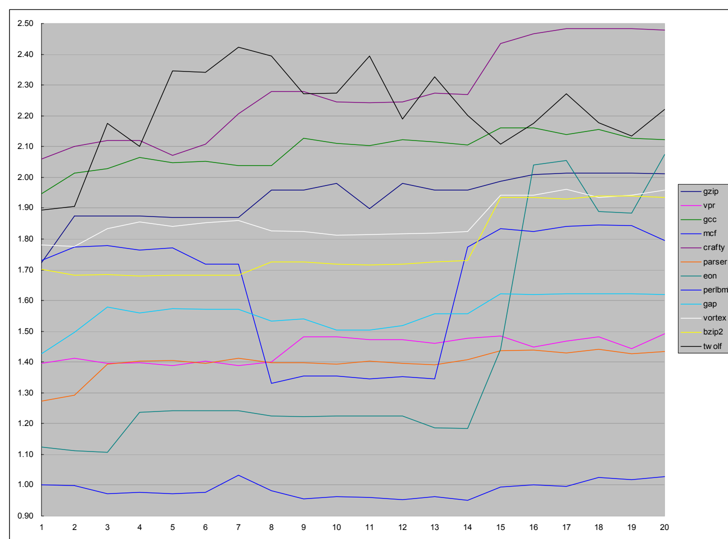
Investigation of Degradations

- Some methods
  - Repeat the testing to filter out noises.
  - Check cycle counts of pre-selected hot functions.
  - Check the corresponding assembly code.
  - Make hypothesis on causes, conduct experiments, and prove or disprove the hypothesis.
  - Use pfmon to find out the effects of dynamic events.
  - …

Performance Trend Analysis



Performance Trend Analysis



Summary: What We Have Learned

- Performance analysis is the base of gaining performance.
- Two steps for understanding and tuning performance
  - Workload characterization
  - Profiling
- Tools are important
  - Reduce the time and tedious manual work needed.
  - Help understand various aspects of performance
- Methodology also important
  - Progress in the right direction
  - Avoid regression

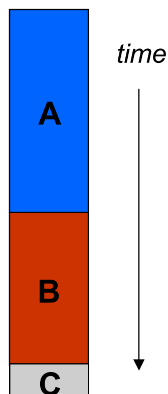
Demo

Use of ORC in Compiler Research for Speculative Multithreading (SpecMT)

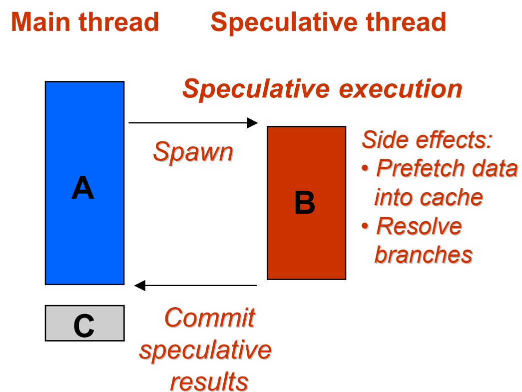
Tin-Fook Ngai @ MRL
Zhaohui Du, Tao Huang*, William Chen @ ICRC
Saisanthosh Balakrishnan @ U. Wisconsin

Concept of Speculative Multithreading

Original program execution:



SpecMT execution:



Hardware Supports for SpecMT

- Special SpecMT instructions
  - Fork, kill, etc.
- Speculative execution
  - Buffering of speculative results
  - No exception on speculative execution
- Dependence checking
  - Check for register/memory RAW dependence violation
- Recovery upon misspeculation
  - Trash all speculative results and re-execute, or
  - Commit correct results and re-execute only misspeculated instructions

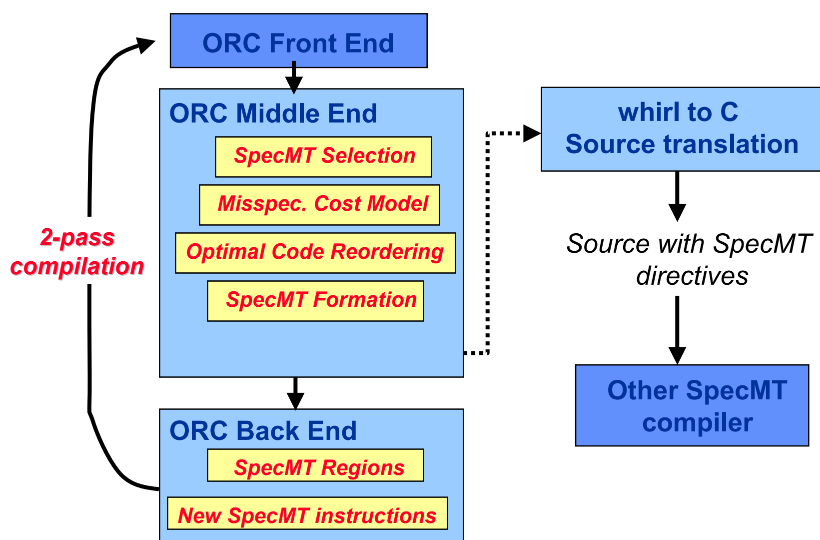
SpecMT Compiler Research Issues

- How to identify and expose every good SpecMT opportunities in a program?
  - Good program coverage by SpecMT is essential
- How to analyze and manage thread-level speculation?
  - Misspeculation cost modeling
  - Need new probabilistic dependence, alias and value analyses
- How to optimize SpecMT code?
  - To reduce misspeculation
- How to best tailor for different forms of hardware supports?

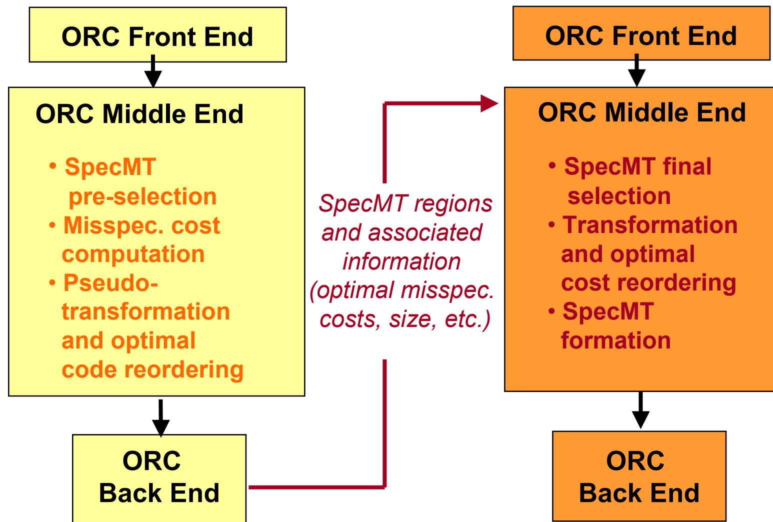
SpecMT Compiler Research at Intel/PSL

- To identify opportunistic speculative threads
  - When and which thread to spawn
  - Initial focus on loops
- To restructure/transform programs
  - More SpecMT opportunities, less misspeculation
- To minimize misspeculation penalty
  - Precompute or predict critical data value before spawning
  - Schedule inter-thread dependent instructions far apart

Our SpecMT Compiler Platform

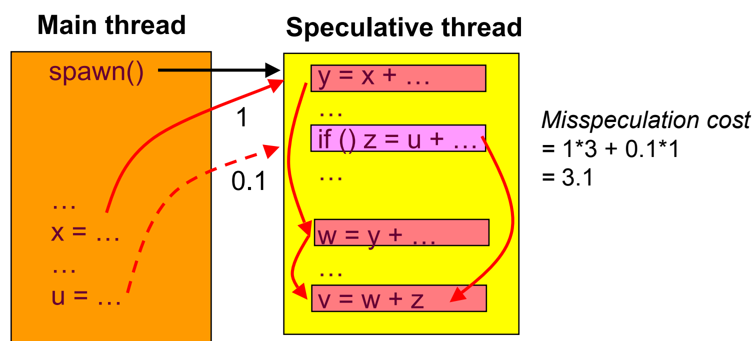


2-Pass SpecMT Compilation



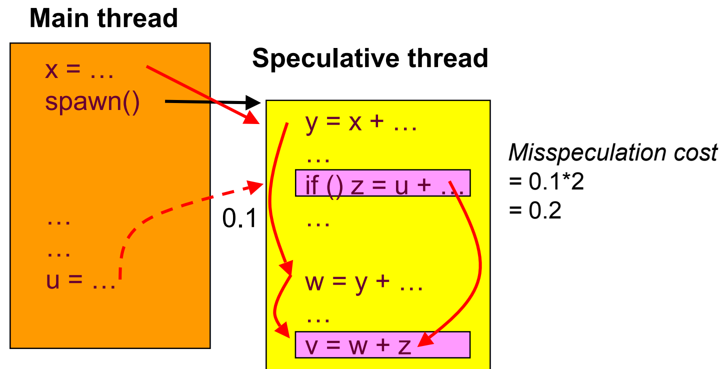
SpecMT Cost Model

- Based on data/control flow dependences and estimated execution probabilities and costs
- Support different SpecMT hardware models



Optimal Code Reordering

- Reorder code before SpecMT spawn point to minimize misspeculation cost
- Reduce critical path to spawn SpecMT threads early



Changes and Enhancements to ORC (1)

ORC Middle End

- A new SpecMT phase in mainopt
  - Right after SSA construction, IVR, copy propagation and first DCE
  - Build internal dependence graph with estimated coderep sizes and profile-feedback edge probabilities
  - Perform code reordering inside the loop body
    - Tackling the non-overlapped live range requirement in ORC SSA
    - Handling of motion of partial conditional statements
  - Insert SpecMT directives as intrinsic calls

Changes and Enhancements to ORC (2)

ORC Middle End

- Unique loop id assignment
  - For loop matching in the 2-pass compilation
  - Propagate preopt loop id to mainopt and reassign loop id after LNO
- IPA summary of function size information
- LNO: Selective loop unrolling
  - Including outer loop unrolling

Changes and Enhancements to ORC (3)

ORC Backend

- Introduce and schedule new SpecMT instructions
  - Have similar semantics to existing chk instructions but executed on B-unit
  - Minor change to the existing machine model
- Translate SpecMT intrinsic calls from Whirl to SpecMT instructions in CGIR
- Form SpecMT regions
  - Both for the SpecMT thread body and for the preparation code before the fork instruction
  - Mark SpecMT regions to be **NO_OPTMIZATION_ACROSS_REGION_BOUNDARIES**

Changes and Enhancements to ORC (4)

ORC Backend

- CFO and EBO
  - Before Region Formation, disable the first CFO stage and limit EBO within single basic block
  - Make CFO and EBO being aware of regions with **NO_OPTMIZATION_ACROSS_REGION_BOUNDARIES** and honor the no-optimization attribute
    - Check blocks for region memberships

Changes and Enhancements to ORC (5)

Whirl2C

- Many fixes to allow source translation of low-level whirl emitted from main-opt
  - Skip machine-dependent whirl lowering after the second DCE

Our Experience with ORC

- A solid, full featured compiler to start with
- Good and handy supports:
  - Rich IR and supports: whirl, SSA, regions in CGIR
  - Profile feedback and IPA
  - Flexible machine model
  - Whirl2c
- Major problems w.r.t. our work
  - Code reordering is complicate with ORC SSA due to its non-overlapped live range requirement (and we solved part of it)
  - Limited outer/while-loop optimizations
  - Current CG phases (including scheduler) do not fully honor region attributes
    - CFO and EBO in CG are not region-based but we made it region-aware (w.r.t. one key attribute)

Research Activities and Plan

Compiler Research on IA-64

Haibo(Jason) Lin

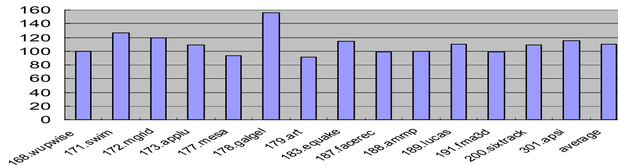
Institute of HPC
Dept. of CS&T, Tsinghua University

Agenda

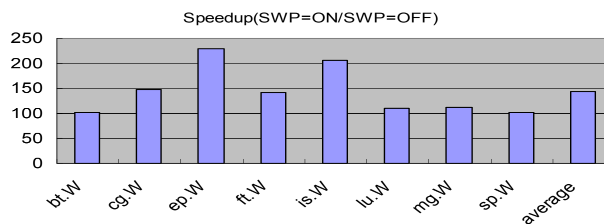
- ❖ Research on ILP
  - ➢ Performance of SWP
  - ➢ Loops Fail to SWP (SWP Failure)
  - ➢ Solutions to SWP Failure
- ❖ Research on TLP
  - ➢ OpenMP Research
  - ➢ Project Overview
  - ➢ Some Experiments and Results

ILP-Performance of SWP

- ❖ A speedup of 10% (SPEC fp2000)



- ❖ A speedup of 48% (NPB 2.3-serial)



ILP-SWP Failure

- ❖ Statistics
  - ➢ 4.7% in SPEC fp2000 (3,000 loops)
  - ➢ 6.8% in NPB (400 loops)
- ❖ Cause of SWP failure
  - ➢ Loop too big
    - – 37% in SPECfp, 20% in NPB-serial
  - ➢ Non-rotating register not enough (general register)
    - – 63% in SPECfp, 80% in NPB-serial
    - – Base update TN accounts for non-rotating register pressure // `ld4 r1=[r2],4`

ILP-Solutions to SWP Failure(RSU)

- Solution 1-Register Sensitive Unrolling(RSU)
  - Why unrolling?
    - Increasing the number of instructions visible to compiler
    - Enabling factional MII(Minimum Initiation Interval)
      - 5 additions / 2 adders ➔ $MII=\lceil 5/2 \rceil=3$ cycles
      - Unroll 2 times ➔ $MII=\lceil 10/2 \rceil=5$ cycles
    - Increasing register requirements
  - Limiting unrolling factor(*K*)
    - Recalculating *K* According to register requirements
    - Increased SWP-ed loops
    - May lead to performance decline due to poor schedule caused by improper unrolling factor

ILP-Solutions to SWP Failure(SRA)

- Solution 2-Stacked Register Allocation(SRA)
  - Currently SWP can only use static register as non-rotating register

  - Allocate stacked register to TNs which need Non-rotating register

ILP-Conclusion

- SWP failure is mainly caused by insufficient non-rotating registers
- RSU & SRA both provide doable solutions to SWP failure
- RSU may lead to performance decline due to improper unrolling factor, and sometimes can not solve SWP failure problem
- SRA performs better than RSU by allocating general register more efficiently

TLP-Project Overview

- ORC based OpenMP compiler
  - Goal
    - Develop an compiler that produces multithreaded programs that can explore intra-die(Hyper threading) and inter-dies (SMP) TLP
  - Motivation
    - Study characteristics of OpenMP itself: how to build up a high performance implementation
    - Find optimizing opportunity for OpenMP implementation and codes written in OpenMP: find out suitable program forms to carry on source-level optimization
    - A testbed for parallelization: compiling for the SMP clusters

TLP-Project Plan

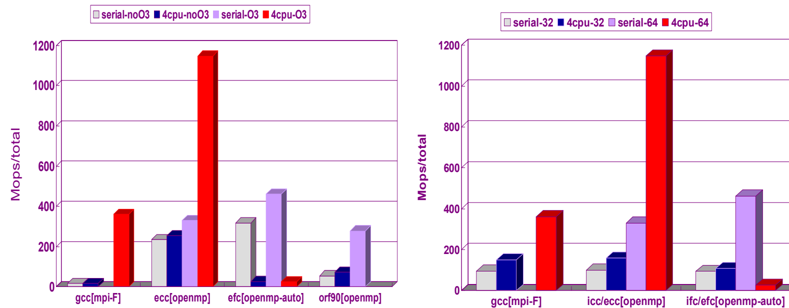
- ❖ Framework building
  - ➢ A base module support on selected Fortran 77 OpenMP directives
  - ➢ Works relatively well
- ❖ Extending to OpenMP Fortran API v1.0
- ❖ Extending to OpenMP Fortran API v2.0
- ❖ Fortran(v2.0)/C frontend building
- ❖ Debugging phase for stability and performance
  - ➢ Regression test and performance tuning

TLP-Project Progress

- ❖ OpenMP tranform module
  - ➢ Have the same function as original SGI module, except for workshare/threadprivate handling(done)
- ❖ Fortran90 FE
  - ➢ In working. Tuning for final release
- ❖ C FE
  - ➢ The Lex/Yacc part has been done
  - ➢ The WHIRL transformation module is in working
- ❖ TO DO
  - ➢ Profiling support for OpenMP performance tuning
  - ➢ Source-level OpenMP parallelization and optimizing techniques

TLP-Experiments and Results

❖ An example-BT.W



Platform (Fig. of the right column)
xx-32 : 4 * Xeon 700MHz, 1G Mem, 1.24G network, Linux 2.4.7-10smp
xx-64 : 4 * Itanium2 900MHz, 4G Mem, 1000M network, Linux 2.4.9-18smp

TLP-Experiments and Results

❖ Conclusion

- Compiler optimization is critical for IA-64 based platform
- Parallel compiler is still constrained by its own ability
- The performance of Itanium2 SMP + ecc + OpenMP could be very good
- With the best optimizing effort by both programmer and compiler, the performance of Itanium2 SMP is 6~10 fold over that of Xeon SMP
- The performance per MHz (floating point) of Itanium2 SMP is 4.5~7.5 fold over that of Xeon SMP

The End

Thanks !

For more information, please mail to:

linhaibo99@mails.tsinghua.edu.cn(ILP)

chenyj99@mails.tsinghua.edu.cn(TLP)

Research Activities in Academia

- University groups using ORC/Open64 as reported in PACT02 tutorial and in the past
  - U. Del, U. Minn, U. Ghent, Georgia Tech, U. Maryland,
  - Rice U., TsingHua U., Peking U., Alberta U., CAS, U. Houston, Princeton
- More universities and research groups:
  - MIT, Prof. Saman Amarasinghe
    - Predication and scheduling
  - Lawrence Berkeley Lab
    - Global address space language (e.g. upc)
- Adjoint Compiler Technology & Standard project
  - Fortran 95 Automatic Differentiation Tool

Known Publications Based on ORC/Open64

- Speculative Register Promotion Using Advanced Load Address Table (ALAT)
  To appear in the CGO-1 Conference, '03.
- EPIC Instruction Scheduling Based on Optimal Approaches
  1st Annual Workshop on EPIC Architectures and Compiler Technology
- SSA Predicated Execution Code Scheduling on SSA form for Itanium
- Maximizing Pipelined Function Units Usage for Minimum Power Software Pipelining
  20-th Int'l Conf. On Computer Design, 02
- A near-optimal instruction scheduler for a tightly constrained variable instruction set embedded processor
  1st Int'l Conf. On Compilers, Architectures, and Synthesis for Embedded Systems, 02
- Engineering a C compiler for the Cognigine cgn1600 Network Processor
  Network Processor Conferences, 2002
- Effective Compilation Support for Variable Instruction Set Architecture
  Int'l Conf. On Parallel Architecture and Compilation Techniques, 02
- Reuse Distance-Based Cache Hints Selection
  8th Int'l Euro-Par Conf. , 02
- SCALEA: A Performance Analysis Tool for Distribution and Parallel Programs
  8th Int'l Euro-Par Conf. , 02
- More …

Checkins and Merges on the Plan

- ORC2.0 early Jan 03 (ICT/ICRC-Intel/PSL-Intel)
- ORC2.0 merge into Open64 soon afterward (U. of Delaware)
- OpenMP support for 2.96 gcc, orcf90 (TsingHua U.)
  - V1.0 early next year
  - V2.0 soon afterward
- U. of Minnesota merge early 03 discussed
- Other merges waiting?

Upcoming User Forum

- On the plan
  - During CGO Conference
  - Late March, ’03 in San Francisco
  - Agenda still open
  - Would like to get suggestions

Future Plan

- Continued activities at Intel
  - Programming Systems Lab
    - Speculative multithread compilation and microarchitecture
    - Streaming data compilation
  - Barcelona Lab/UPC
    - Multithread compilation
- Continued activity at Chinese Academy of Sciences
  - Streaming data and network processor compilation

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