



Micro-36 Tutorial

Open Research Compiler (ORC): Proliferation of Technologies and Tools

Co-organizers:

Roy Ju*, Pen-Chung Yew+, Ruiqi Lian**, Lixia Liu*, Tin-Fook Ngai*,
Robert Cohn*, Costin Iancu**

*Intel Corp, **Chinese Academy of Science,
+Univ. of Minnesota, ++ Lawrence Berkeley Lab

Presented at the 36th International Symposium on Microarchitecture
(Micro-36)

San Diego, CA
December 1, 2003



Agenda

- Overview of ORC Features and ORC 2.1
- Alias and Dependence Profiling and Enabled Optimizations
- Pin – Binary Instrumentation Tool
- Speculative Parallel Threading
- Unified Parallel C





Overview of ORC



ORC

- Achieved objective: be a leading open source IPF (IA-64) compiler to the research community
- Initiated by Intel Microprocessor Research Labs (MRL) in Q3 '00
- Joint efforts between Intel MRL and Chinese Academy of Sciences
- To download at <http://ipf-orc.sourceforge.net/>
- User community and support:
 - ipf-orc-support@lists.sourceforge.net
 - open64-devel@lists.sourceforge.net

* IPF for Itanium Processor Family in this presentation



The ORC Project

- Based on the Pro64 open source compiler released by SGI
 - Retargeted from the MIPSPro product compiler
- ORC development efforts started in Q4 2000
- Four ORC releases so far
 - ORC 2.1 released in July '03
 - ORC 2.0 released in Jan '03
 - ORC 1.1 released in July '02
 - ORC 1.0 released in Jan '02
- Summary of major ORC features :
 - Largely redesigned CG
 - Enhanced IPA and WOPT
 - Various enhancements to boost performance
 - Tools and other functionality

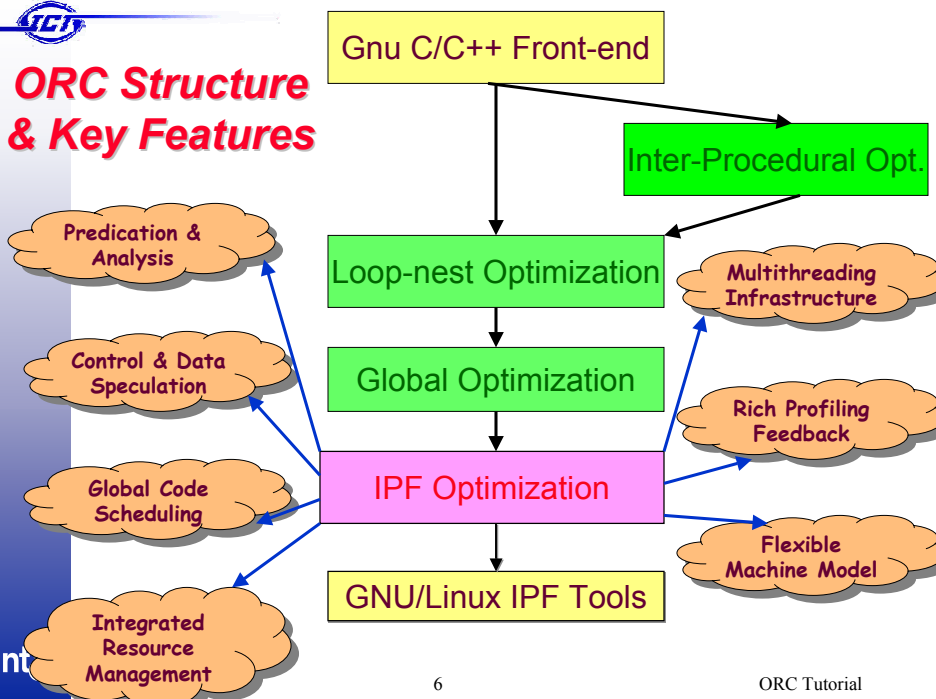


5

ORC Tutorial



ORC Structure & Key Features

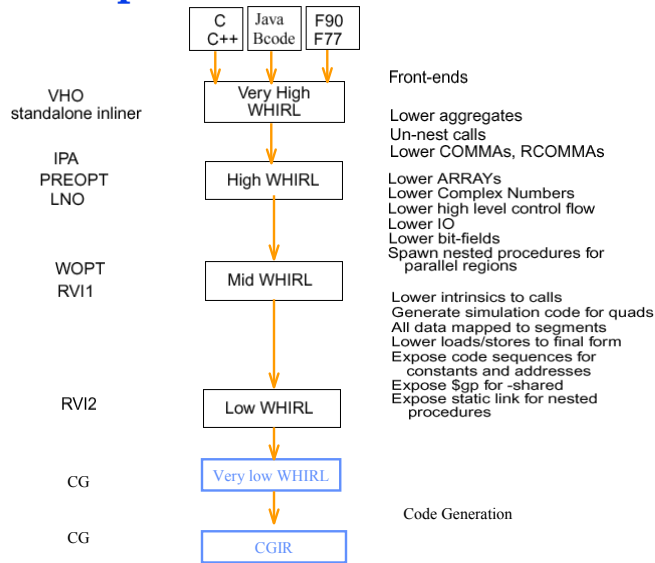


6

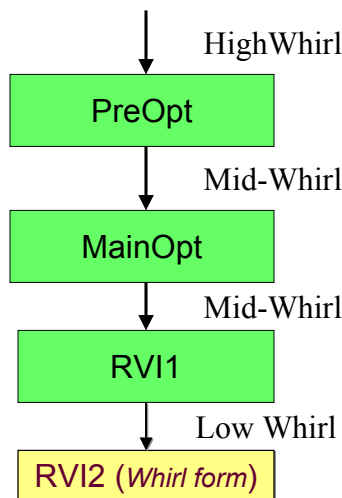
ORC Tutorial



Flow of Open64

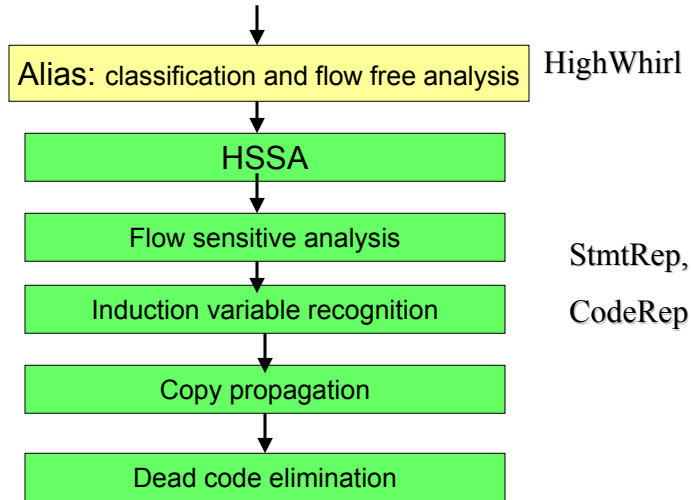


Flow of Global Optimizer (WOPT)

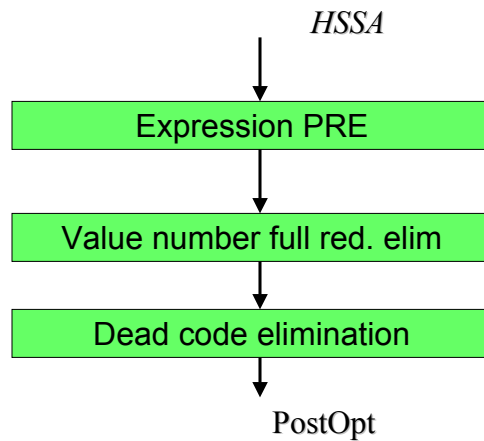




Major Components of Preopt

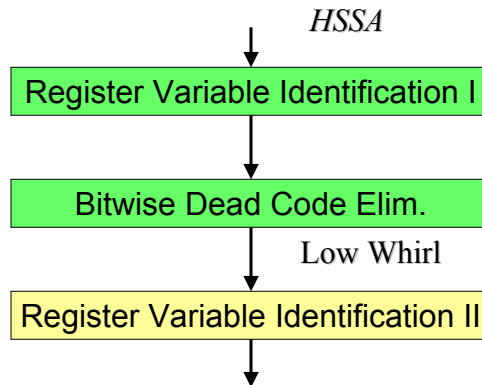


Major components of MainOpt





Major components of PostOpt



Loop Nest Optimizer - LNO

- Works on High Whirl
- Optimizations performed
 - Loop transformations for memory hierarchy
 - Automatic parallelization
 - Array privatization
 - Cache line optimizations
 - Data prefetch/memory opt
 - OpenMP support



Loop Nest Optimization

- Assumes “*preopt*” to normalize loops and code preparation
- Fast and efficient array data dependency analysis
- Based on unimodular transformations
- Passes array dependency information to code generation phase through “*MAP*”



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

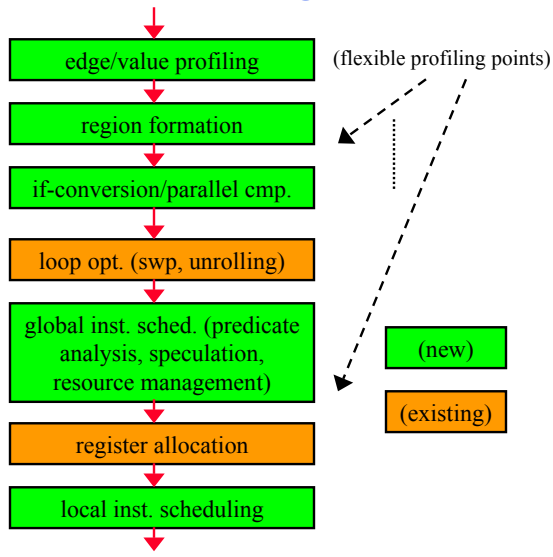
- Perform transformation based on
 - Info collected during analysis
 - Data promotion
 - Constant propagation
 - Indirect call to direct call
 - Assigned once globals
 - Decisions made during analysis
 - Inlining
 - Common padding and splitting



Code Generation

- Has been a major focus in ORC and 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

Major Phase Ordering in CG



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, i.e. frequency weighted cycles of each basic block.
 - Need pre-generated feedback information.



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.



ORC 2.1 Features

- Focusing on Itanium2-specific optimizations
- Tuning existing optimizations for Itanium2
- Inter-procedural allocation stacked registers
 - Published at ICS '03
- Pin – an IPF binary instrumentation tool



Itanium2 Optimizations

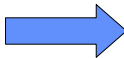
- More resources and more flexible dispersal rule
- Shorter latency and smaller penalty
- Cache optimization
 - Bank Conflict
 - Cache Line Conflict
 - Prefetch distance
- Single Cycle Loop
 - Unrolling
- Speculation of predicated chk



Dispersal rule

- Example:

```
{.mmi
  nop.m 0
  nop.m 0
  nop.i 0
}{ .mmi
  st
  st
  nop.i 0
}
```



```
{.mmi
  st
  st
  nop.i 0
}
```



Cache Conflict

- Various conflicts
 - L2 Bank conflict: load/load store/store load/store
 - L1 Cache Line conflict: store/load store/store
- Example in mcf for L2 bank conflict
- load/load conflict
between node->pred and node->child, node-> orientation & node->basic_arc, can't be issued in same cycle.

```
while( node )    {
    if( node->orientation == UP )
        node->potential = node->basic_arc->cost + node->pred->potential;
    else /* == DOWN */
        { node->potential = node->pred->potential - node->basic_arc->cost;
        }
    tmp = node;
    node = node->child;
}
```

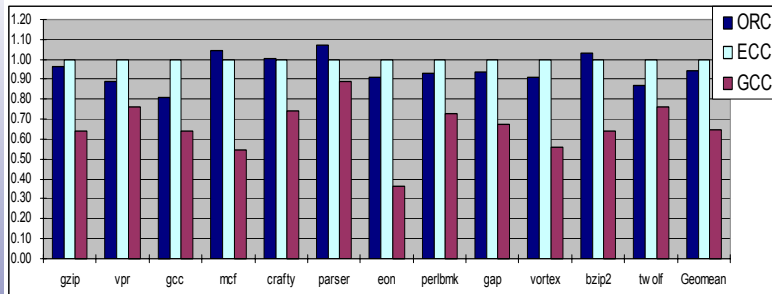


Performance Disclaimer

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.



INT Performance on Itanium2/Linux



- Testing environment:
 - 4-way 900 MHz Itanium2 , 3M L3 Cache, 1G Mem, RH 7.2
- ORC 2.1 at 5% from Ecc 7.0 and 30% ahead of Gcc 3.1

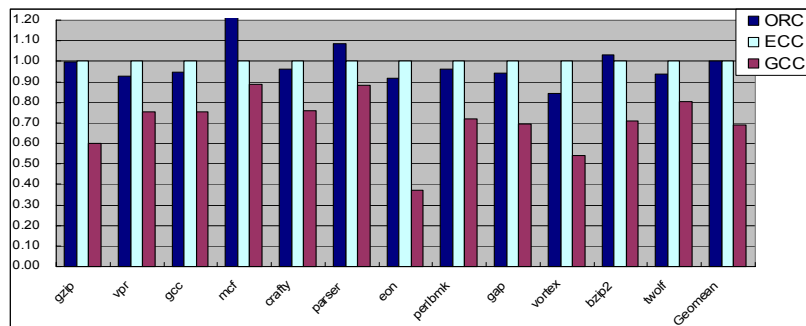


25

ORC Tutorial



INT Performance on Itanium/Linux



- Testing environment:
 - HP i2000 WS: 733 MHz Itanium, 2M L3 Cache, 1G Mem, RH 7.2
 - Compiled with SPEC base options for Ecc 7.0 and -O3 for Gcc 3.1
 - Linked with standard libraries on Linux
- ORC 2.1 on par with ECC 7.0 and 30% ahead of Gcc 3.1

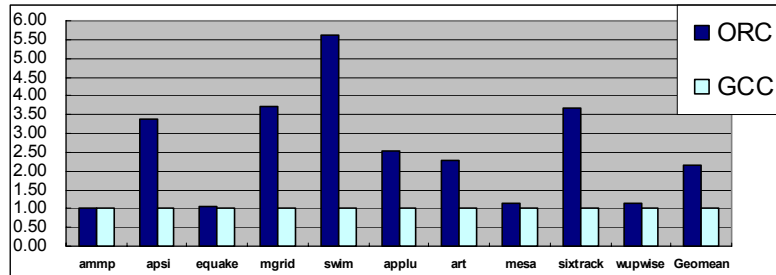


26

ORC Tutorial



FP Performance on Itanium2/Linux



- Testing environment:
 - 4-way 900 MHz Itanium2 , 16K L1 DCache, 16K L1 ICache, 256K L2 Cache, 3M L3 Cache, 1G Mem, RH 7.2
 - Compiled with the “-O3” option for both ORC 2.1 and Gcc 3.1
 - Linked with standard libraries on Linux
 - Fortran 90 cases are not included



ORC 2.1 about twice of Gcc 3.1 on the FTN77 & C FP performance at -O3

27

ORC Tutorial



ORC Future Plan

- Will be less on IPF performance centric features
 - Have achieved its performance goal
- Working on upgrading the ORC front-end to GNU C & C++ 3.2 as well as the build compiler
 - anyone interested in an pre-release of work in progress?
- May merge certain major user contributions in future releases
- The Intel and CAS ORC teams use ORC for various research topics
 - Publications listed on the web site
- To help organize a more active user community



28

ORC Tutorial



Open64/ORC User Activities

- > 4000 downloads since ORC 1.0
- A worldwide Open64/ORC user community
- Adopted by many academic research groups worldwide
 - Visible in publications
- Regular tutorials & user forums
 - Micro34, PLDI02, PACT02, Micro35, CGO03, Micro36
- Prof. G. Gao organizing another user forum in '04
- Looking for funding to better organize the community and establish the mechanism to coordinate contributions

Alias and Data Dependence Profiling

Pen-Chung Yew
Department of Computer Science and
Engineering
University of Minnesota

1

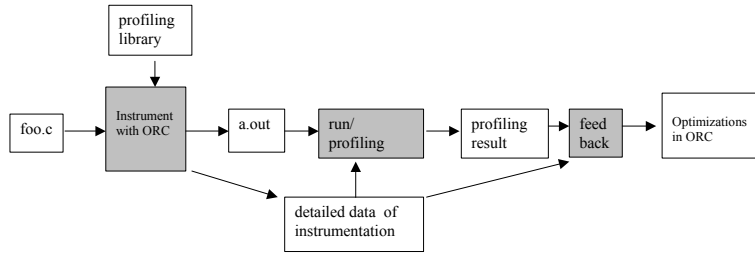
Outline

- Instrumentation-based alias profiling
- Instrumentation-based data dependence profiling
- Techniques to reduce profiling overhead
- Data speculation using profiling information
- Summary

2

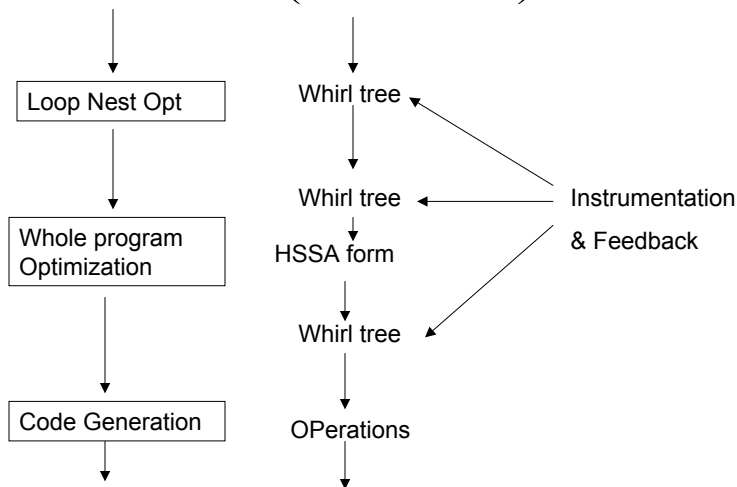
Instrumentation-Based Profiling

- Three steps: instrumentation, profiling and feedback



3

Intel's Open Research Compiler (ORC v2.0)



4

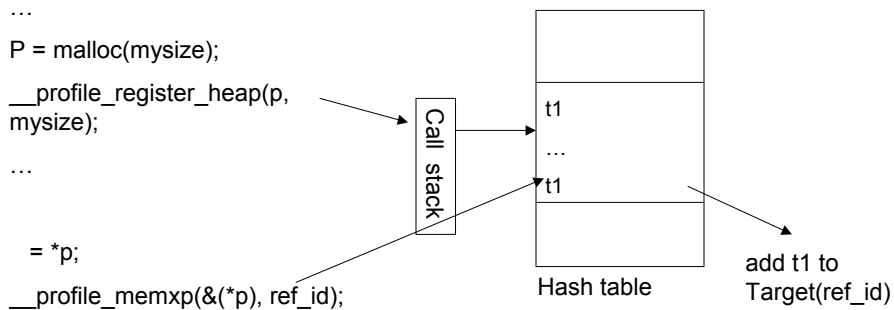
Alias Profiling

- Target set for indirect references
 - variables or heap blocks
- Read/write set for function calls
- Additional information about a target
 - Probability: $(\# \text{ occurrences of a target}) / (\# \text{ occurrences of this reference})$
 - Field tag
 - Calling context

5

How to Perform Alias Profiling

- Simulate naming schemes: variables and heap objects
- Calculate the points-to set by address



6

Feedback of Alias Profiling

- Whirl nodes are mapped back by the traversal order of procedure body
- Variables are also mapped back by the traversal order of symbol table
- Target sets of references are recorded with references

7

Determine Aliases Using Profile

- Alias relation based on profile can be determined by checking the intersection of the target sets

– un-reached references: unknown

– Further infer the probability

$\text{target}(\text{ref}) = \{(v_i, p_i), i=0, n\}$, where v_i is the variable and p_i is its probability

$\text{Is_Aliased_by_Profile}(\text{ref1}, \text{ref2}) =$

$\min(\text{sum_p}(\text{ref1}, \text{ref2}), \text{sum_p}(\text{ref2}, \text{ref1}));$

$\text{sum_p}(\text{ref1}, \text{ref2}) =$

$\sum p_i$, where, $(v_i, p_i) \in \text{target}(\text{ref1})$ and $(v_i, x) \in \text{target}(\text{ref2})$

8

Example of Feedback

Assume:

two references, WN *ref1, *ref2

Target(ref1) = {(a, 10%), (b, 80%), (c, 10%)}

Target(ref2) = {(b, 20%), (c, 20%), (d, 60%)}

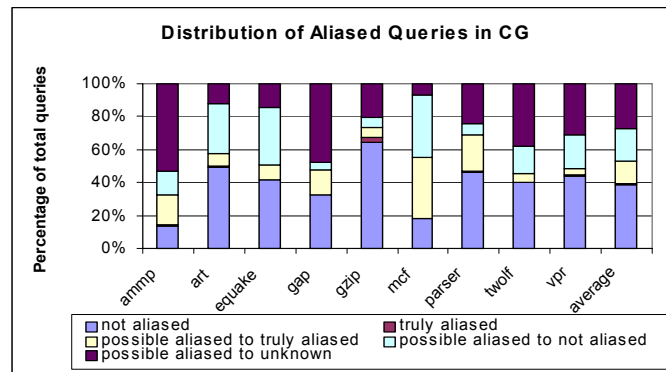
Result:

sump(ref1, ref2) = 80%+10% = 90%

sum_p(ref2, ref1) = 20%+20% = 40%

Is_Alias_by_Profile(ref1, ref2) = 40%

Alias Profiling vs. Static Analysis



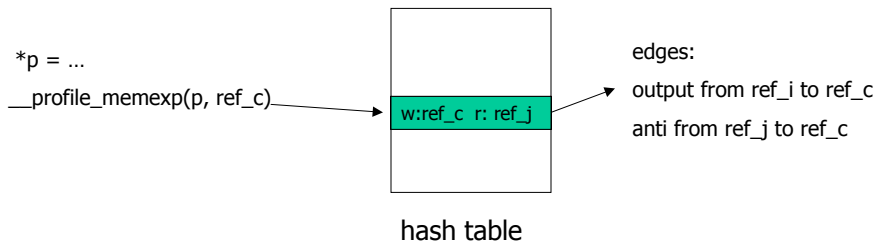
Data Dependence Profiling

- Data dependence edges among memory references and function calls
- Detail information
 - type: flow, anti, output, or input
 - probability: frequency of occurrence
- When loops are targeted
 - dependence distance: limited

11

How to Perform Data Dependence Profiling

- Use hashing to speedup the pair-wise address comparing
- Detect a data dependence edge by comparing the latest read and write to an address stored in the hashed entry
- Overwrite the latest read or write in the hashed entry



12

DD Profiling for Function Calls

- Dependence edges across procedures cannot be directly used by compilers
- Record the calling context to find the proper procedure call sites.
- Example:

P() {	An edge from a reference in Q to a
Q();	reference in R is detected by profiling
...	
R();	This edge should be translated into the
}	edge for call Q to call R in procedure P
	with the help of calling context

13

DD Profiling for Loops

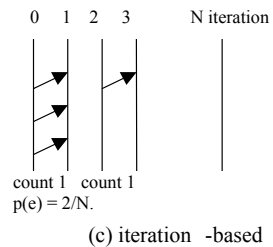
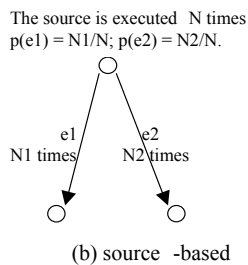
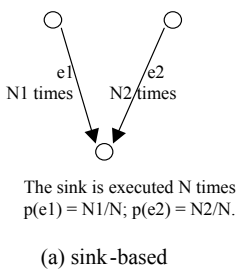
- Each loop has an iteration counter and each loop nest has an iteration vector (IV)
- Record the iteration vector in the hashed entry associated with the reference ID
- When a dependence edge is detected,
distance vector = current IV – recorded IV

14

Different Definitions of Probability

- Occurrence-based probability for dependences in procedures
 - sink: $(\# \text{occurrence of edge}) / (\# \text{occurrence of sink})$
 - source: $(\# \text{occurrence of edge}) / (\# \text{occurrence of source})$
- Iteration-based probability for dependences in loops
 - $(\# \text{iteration in which the edge occurs}) / (\# \text{iteration})$

Examples of Probability

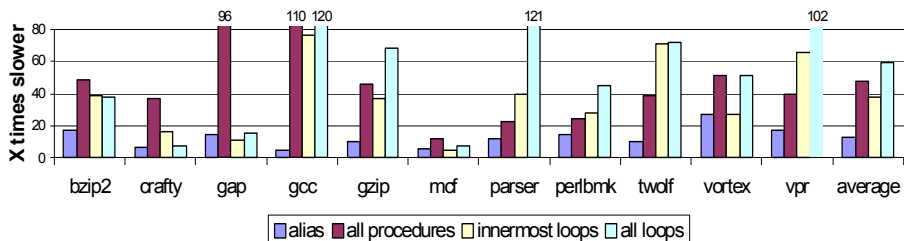


Alias Profiling vs. DD Profiling

- Output: points-to set vs. dependence edges
- Representation: location factor SSA-form vs. dependence graph or statement factor SSA-form
- Precision: name based vs. address based
- Maintenance: easy vs. difficult
- Complexity: linear vs. square

17

Overhead of Profiling



- Compiler: ORC version 2.0
- Machine: Itanium2, 900 MHz and 2G memory
- Benchmarks: SPEC CPU2000 Int
- Instrumentation optimization has been done

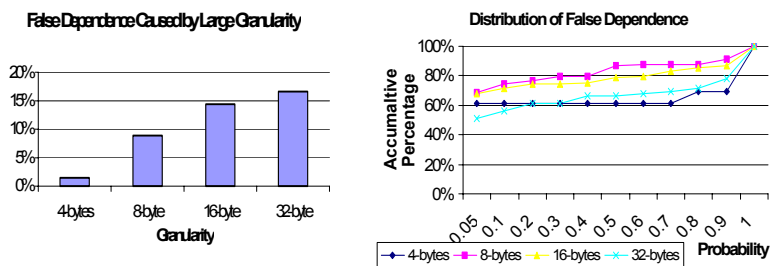
18

Techniques to Further Reduce Overhead

- Reduce the space requirement by hashing
- Larger granularity of address
 - Smaller iteration counter
- Sampling
 - Sample the snap shots of procedures or loops instead of individual references
 - Use instrumentation-based sampling framework
 - Switch at procedures or loops

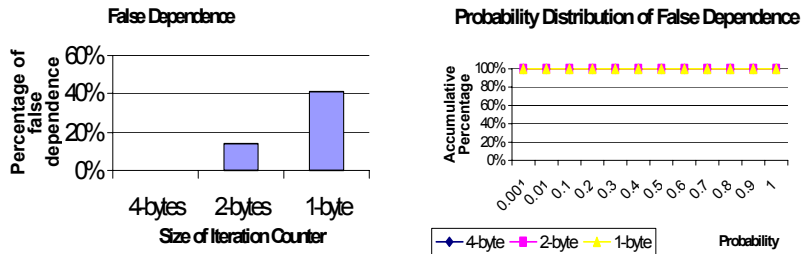
19

Granularity of Address Used in Hashing Function



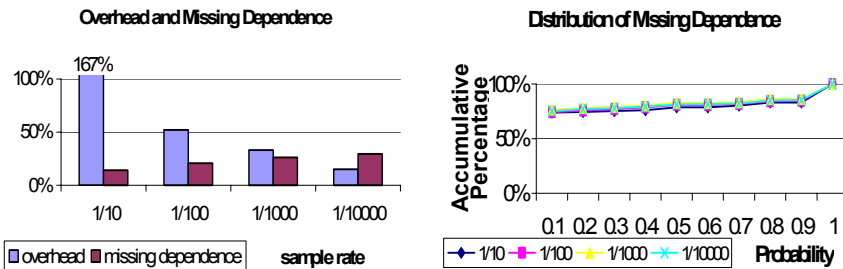
20

Size of Iteration Counter



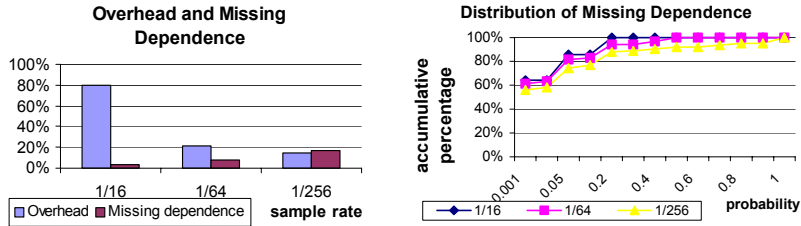
21

Sampling in DD Profiling for Procedures



22

Sampling in DD Profiling for Loops



23

Partial Redundancy Elimination with Data Speculation

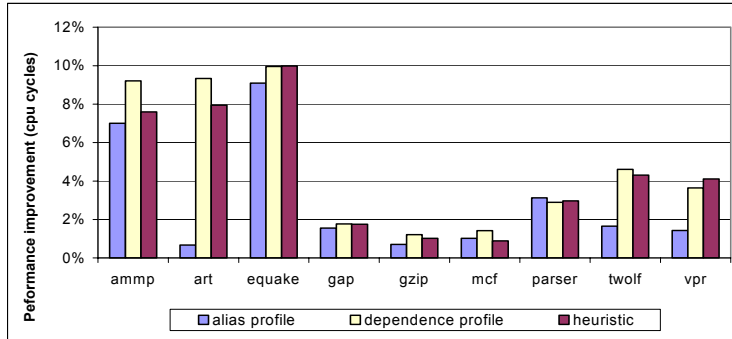
- Use ALAT in Intel's IA-64 architecture
- A simple example:

load p		ld.a r14=p
store *q	→	st [q]=r15
load p		ld.c r14=p

- **Benefits:**
 - ld.c can be turned into a nop
 - expose more redundant expressions, such as *p, or p+a

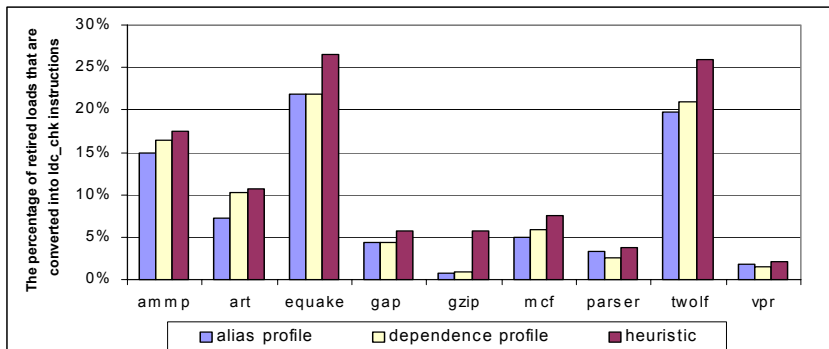
24

Performance Improvement



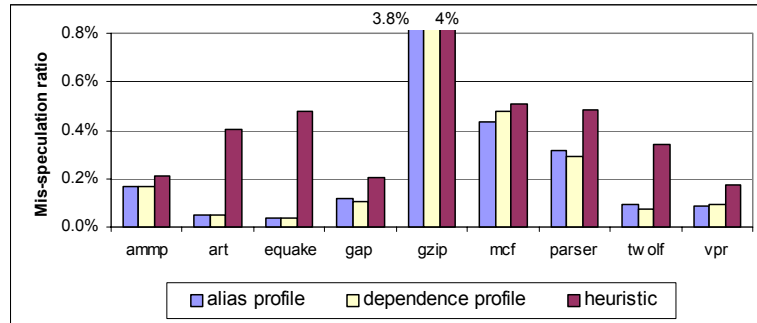
25

Reduction in Load Operations



26

Mis-speculation Rates



27

DD Profiling Guided Scheduling

- Collect DD profile with *train* input
- Feed back profile to dependence graph for code scheduling
- Mark edges with probability less than 2% as candidates for speculation.
- Perform code motion not limited by heuristics
- Run result code with *ref* input

28

Code Scheduling

	Performance	advanced loads in all loads	ALAT failure rate
equake	8.36%	18.80%	0.34%
art	32.34%	31.32%	0.88%
mesa	12.23%	9.14%	0.09%
bzip	0.34%	6.81%	6.98%
gzip	0.00%	2.45%	0.11%
parser	2.36%	2.92%	2.00%
vortex	1.94%	4.62%	0.13%
Average	8.22%	10.8%	1.5%

29

Summary

- Instrumentation-based alias and DD profiling can be performed with low overhead
- The type of information collected should be determined by the “consumer” of such information
- Alias and DD profiles can be used effectively to support data speculation
- Alias and DD profiles can also be used to support speculative thread generation

30

Instrumentation of IPF/Linux Programs with Pin

<http://www.intel.com/software/products/opensource/tools1/inst/>

Robert Cohn
MMDC
Intel

1

What Does Pin Stand For?

- **P**in **I**s **N**ot an acronym
- Pin is based on the post link optimizer spike
 - Use dynamic code generation to make a less intrusive profile guided optimization and instrumentation system
 - Pin is a small spike

2

Instrumentation

```
Max = 0;
for (p = head; p; p = p->next)
{
    printf("%i\n", Loop);
    if (p->value > max)
    {
        printf("%i\n", max);
        max = p->value;
    }
}
```

Dynamically defined

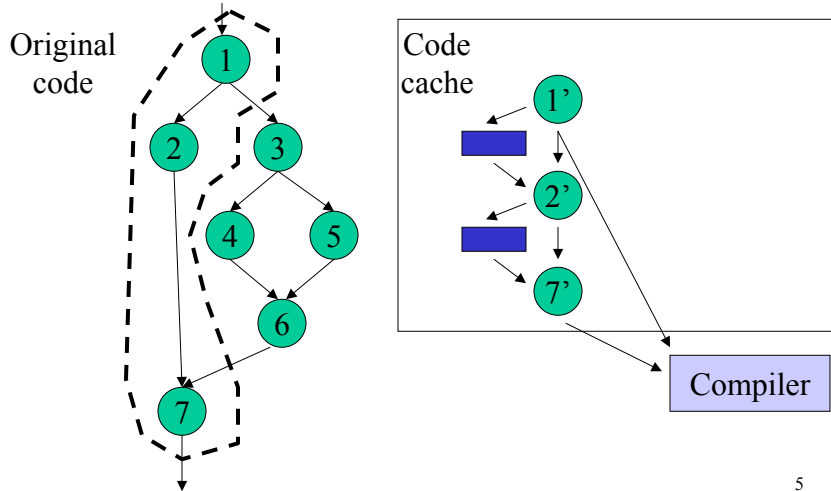
3

What Can You Do With Instrumentation?

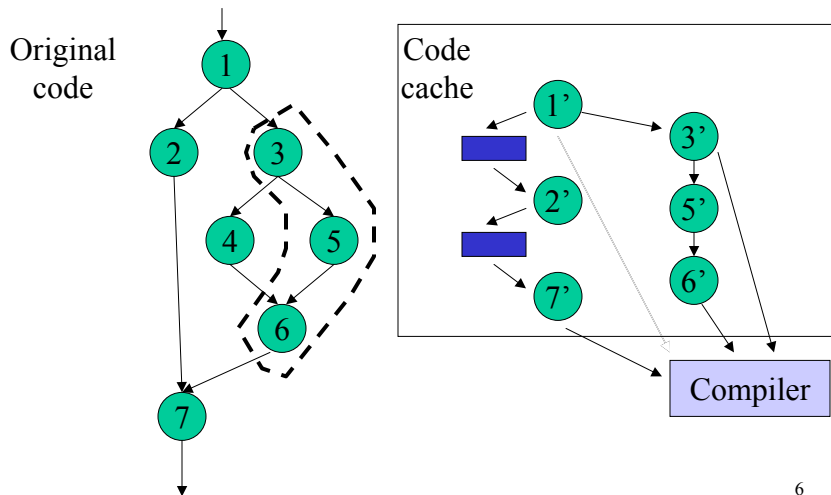
- Profiler for optimization:
 - Basic block count
 - Value profile
- Micro architectural study
 - Instrument branches to simulate branch predictor
 - Generate traces
- Bug checking
 - Find references to uninitialized, unallocated data

4

Execution Drives Instrumentation



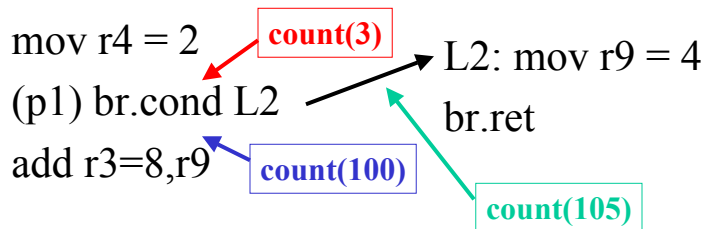
Execution Drives Instrumentation



Inserting Instrumentation

Relative to an instruction:

1. Before
2. After
3. Taken edge of branch



7

Analysis Routines

- Instead of inserting IPF instructions, user inserts calls to **analysis routine**
 - User specified arguments
 - E.g. Increment counter, record memory address, ...
- Written in C, ASM, etc.
- Optimizations like inlining, register allocation, and scheduling make it efficient

8

Instrumentation Routines

- **Instrumentation routine** walks list of instructions, and inserts calls to analysis routines
- User writes instrumentation routine
- Pin invokes instrumentation routine when placing new instructions in code cache
- Repeated execution uses already instrumented code in code cache

9

Example: Instruction Count

```
[rscohn1@shli0005 Tests]$ hello
Hello world
[rscohn1@shli0005 Tests]$ icount -- hello
Hello world
ICount 496890
[rscohn1@shli0005 Tests]$
```

10

Example: Instruction Count

```
counter++;  
mov r2 = 2  
counter++;  
add r3 = 4, r3  
counter++;  
(p2) br.cond L1  
counter++;  
add r4 = 8, r4  
counter++;  
br.cond L2
```

11

```
#include <stdio.h>  
#include "pinstr.H"  
  
UINT64 icount=0;  
  
// Analysis Routine  
void docount() { icount++; }  
  
// Instrumentation Routine  
void Instruction(INS ins)  
{  
    PIN_InsertCall(IPOINT_BEFORE, ins,  
                  (AFUNPTR)docount, IARG_END);  
}  
  
VOID Fini()  
{  
    fprintf(stderr, "ICount %lld\n", icount);  
}  
  
int main(int argc, char *argv[])  
{  
    PIN_AddInstrumentInstructionFunction(Instruction);  
    PIN_AddFiniFunction(Fini);  
    PIN_StartProgram();  
}
```

12

Example: Instruction Trace

```
[rscohn1@shli0005 Trace]$ itrace -e hello
Hello world
[rscohn1@shli0005 Trace]$ head prog.trace
0x200000000000045c0
0x200000000000045c1
0x200000000000045c2
0x200000000000045d0
0x200000000000045d2
0x200000000000045e0
0x200000000000045e1
0x200000000000045e2
[rscohn1@shli0005 Trace]$
```

13

Example: Instruction Trace

```
traceInst(ip);
mov r2 = 2
traceInst(ip);
add r3 = 4, r3
traceInst(ip);
(p2) br.cond L1
traceInst(ip);
add r4 = 8, r4
traceInst(ip);
br.cond L2
```

14

```

#include <stdio.h>
#include "pinstr.H"
FILE *traceFile;
void traceInst(long * ipsyll){
    fprintf(traceFile, "%p\n", ipsyll);
}
void Instruction(INS ins){
    PIN_InsertCall(IPOINT_BEFORE, ins,
        (AFUNPTR)traceInst, IARG_IP_SLOT, IARG_END);
}
int main(int argc, char *argv[])
{
    PIN_AddInstrumentInstructionFunction(Instruction);
    traceFile = fopen("prog.trace", "w");
    PIN_StartProgram();
}

```

15

Arguments to Analysis Routine

- IARG_UINT8, ..., IARG_UINT64
- IARG_REG_VALUE <register name>
- IARG_IP_SLOT
- IARG_BRANCH_TAKEN
- IARG_BRANCH_TARGET_ADDRESS
- IARG_THREAD_ID
- IARG_IN_SIGNAL

16

More Advanced Tools

- Instruction cache simulation: replace itrace analysis function
- Data cache: like icache, but instrument loads/stores and pass effective address
- Malloc/Free trace: instrument entry/exit points
- Detect out of bound stack references
 - Instrument instructions that move stack pointer
 - Instrument loads/stores to check in bound

17

Example: Faster Instruction Count

```
counter++;  
mov r2 = 2  
counter++;  
add r3 = 4, r3  
counter++ 3;  
(p2) br.cond L1  
counter++;  
add r4 = 8, r4  
counter++ 2;  
br.cond L2
```

18

Sequences

- List of instructions that is only entered from top

Program:	Sequence 1:	Sequence 2:
<code>mov r2 = 2</code>	<code>mov r2 = 2</code>	
<code>L2:</code>	<code>add r3 = 4, r3</code>	<code>add r3 = 4, r3</code>
<code>add r3 = 4, r3</code>	<code>add r4 = 8, r4</code>	<code>add r4 = 8, r4</code>
<code>add r4 = 8, r4</code>	<code>br.cond L2</code>	<code>br.cond L2</code>
<code>br.cond L2</code>		

19

```
void docount(UINT64 c) { icount += c; }
void Sequence(INS head) {
    INS ins;
    INS last = INS_INVALID();
    UINT64 count = 0;
    for (ins = head; ins != INS_INVALID(); ins = INS_Next(ins)) {
        count++;
        switch(INS_Category(ins)) {
            case TYPE_CAT_BRANCH: case TYPE_CAT_CBRANCH:
            case TYPE_CAT_JUMP: case TYPE_CAT_CJUMP:
            case TYPE_CAT_CHECK: case TYPE_CAT_BREAK:
                PIN_InsertCall(IPOINT_BEFORE, ins,
                    (AFUNPTR)docount, IARG_UINT64, count, IARG_END);
                count = 0;
                break;
        }
        last = ins;
    }
    PIN_InsertCall(IPOINT_AFTER, last,
        (AFUNPTR)docount, IARG_UINT64, count, IARG_END);
}
```

20

Instruction Information Accessed at Instrumentation Time

1. **INS_Category(INS)**
2. **INS_Address(INS)**
3. **INS_Regr1, INS_Regr2, INS_Regr3, ...**
4. **INS_Next(INS), INS_Prev(INS)**
5. **INS_BraType(INS)**
6. **INS_SizeType(INS)**
7. **INS_Stop(INS)**

21

Callbacks

- Call backs for instrumentation
 - PIN_AddInstrumentInstructionFunction
 - PIN_AddInstrumentSequenceFunction
- Other callbacks
 - PIN_AddImageLoadFunction
 - PIN_AddThreadBeforeFunction
 - PIN_AddThreadAfterFunction
 - PIN_AddFiniFunction: last thread exits

22

Instrumentation is Transparent

- When application looks at itself, sees same:
 - Code addresses
 - Data addresses
 - Memory contents
- Don't want to change behavior, expose latent bugs
- When instrumentation looks at application, sees original application:
 - Code addresses
 - Data addresses
 - Memory contents
- Observe original behavior

23

Pin Instruments All Code

- Execution driven instrumentation:
 - Shared libraries
 - Dynamically generated code
- Self modifying code
 - Instrumented first time executed
 - Pin does not detect code has been modified

24

Dynamic Instrumentation

- While program is running:
 - Instrumentation can be turned on/off
 - Code cache can be invalidated
 - Reinstrumented the next time it is executed
 - Pin can detach and run application native
- Use this for fast skip

25

Advanced Topics

- Symbol table
- Altering program behavior
- Threads
- Signals
- Debugging

26

Symbol Table/Image

- Query:
 - Address \leftrightarrow symbol name
 - Address \Rightarrow image name (e.g. libc.so)
 - Address \Rightarrow source file, line number
- Instrumentation:
 - Procedure before/after
 - PIN_InsertCall(IPOINT_BEFORE, Sym, Afun, IARG_REG_VALUE, REG_REG_GP_ARG0, IARG_END)
 - Before: at entry point
 - After: immediately before return is executed
 - Catch image load

27

Alter Program Behavior

- Analysis routines can write application memory
- Replace one procedure with another
 - E.g. replace library calls
 - PIN_ReplaceProcedureByAddress(address, funptr);
 - Replaces function at address with funptr

28

Alter Program Behavior

- Change values of registers:
 - PIN_InsertCall(IPOINT_BEFORE, ins, zap, IARG_RETURN_VALUES, IARG_REG_VALUE, REG_G03, IARG_END);
 - Return value of function zap is written to r3

29

Alter Program Behavior

- Change instructions:
 - Ld8 r3=[r4]
 - Becomes:
 - Ld8 r3=[r9]
 - INS_RegR1Set(ins, REG_G09)
- Pin provides virtual registers to instrumentation:
 - REG_INST_GP0 – REG_INST_GP9

30

Threads

- Pin is thread safe
- Pin assumes your tool is not thread safe
 - Tell pin how many threads your tool can handle
 - `PIN_SetMaxThreads(INT)`
- Make your tool thread safe
 - Instrumentation code: guarded by single lock
 - Analysis code – protect global data structures
 - Lock
 - Use pin provided routines, not pthreads
 - Thread local storage
 - Use `IARG_THREAD_ID` to index into array

31

Signals

- Signal safe code
 - Re-entrant (only stack data)
 - Block signals (requires system call)
- Pin is signal safe
- Make you tool signal safe
 - Instrumentation code: signals are blocked
 - Analysis code
 - Don't instrument signal path
 - `-native_signals`
 - `IARG_IN_SIGNAL` to dynamically detect in signal

32

Debugging

- Instrumentation code
 - Use gdb
- Analysis code
 - Pin dynamically optimizes analysis code to reduce cost of instrumentation (up to 10x)
 - Disable optimization to use gdb
 - `icount -p pin -O0 -- /bin/ls`
 - Otherwise, use `printf`

Speculative Parallel Threading Compiler

Tin-Fook Ngai *

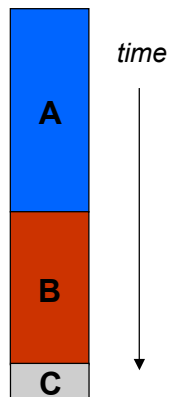
William Chen, Zhao-Hui Du, Xiao-Feng Li,
Chu-Choew Lim*, Chen Yang, Qingyu Zhao
Intel China Research Center
* *Microprocessor Technology Labs*

Intel Corporation

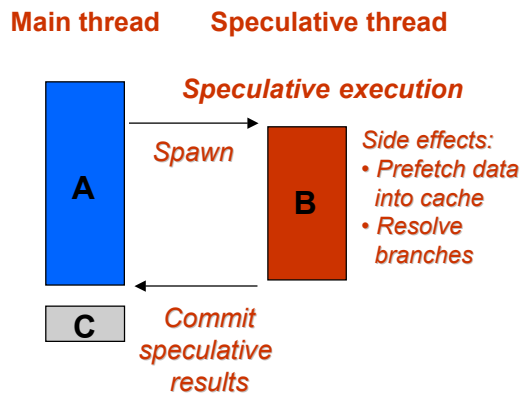


Speculative Parallel Threading

Original program execution:



SPT execution:



Hardware Supports for SPT

- Special SPT 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

Speculation and Parallelism

- SPT performs both control and data speculation
- SPT achieves both computation and data/memory parallelism
- Types of SPTs
 - Procedure continuation SPT
 - **Loop iteration SPT**
 - Loop continuation SPT
 - Region run-ahead SPT

⇒ **Compiler is an essential means**

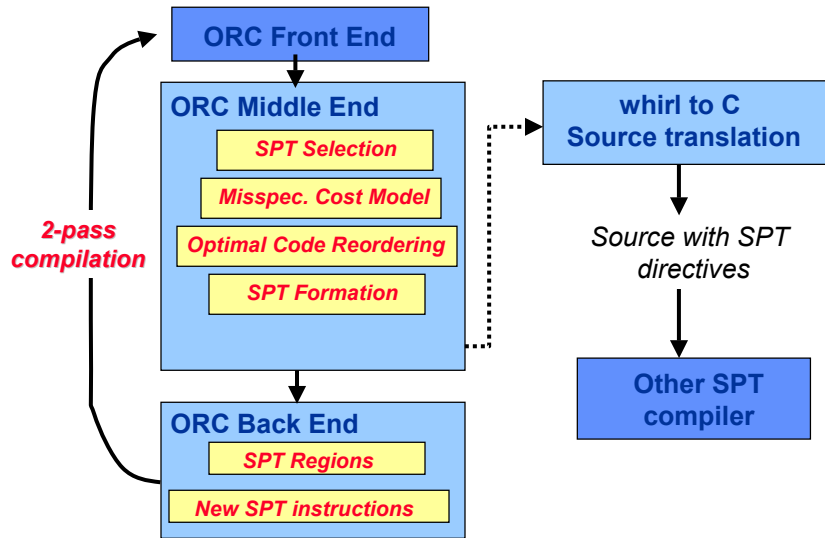
Major SPT Compiler Research Issues

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

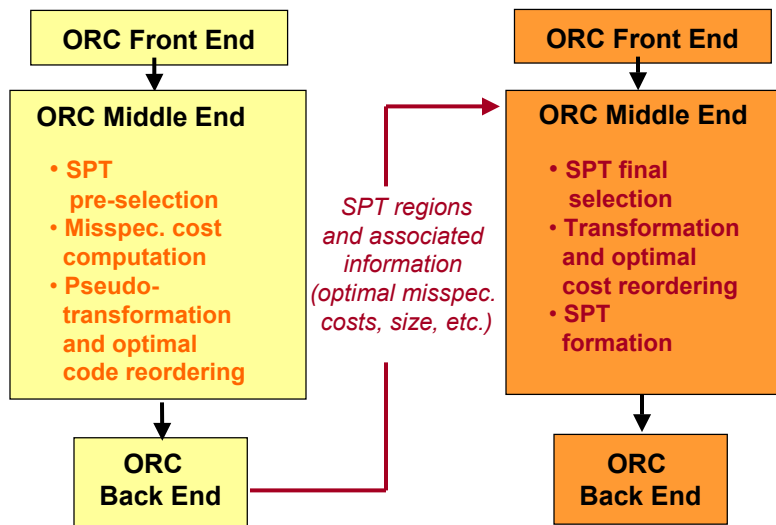
PSL SPT Compiler

- Cost-driven speculative parallelization
 - Use misspeculation cost to drive
 - SPT transformation
 - SPT optimization
 - SPT loop selection
- A 2-pass SPT compilation framework
- Prototyped on ORC (IPF) compiler

Our SPT Compiler Platform

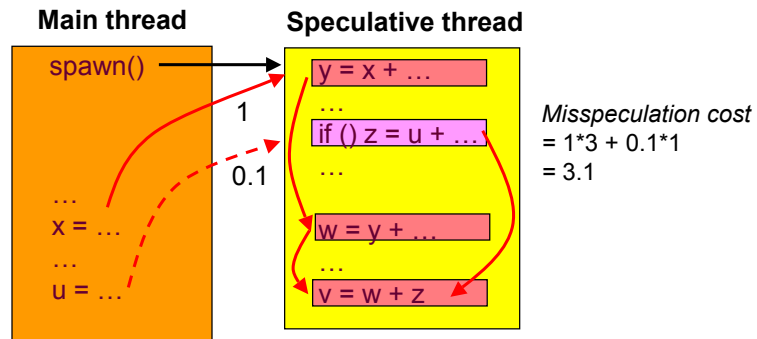


2-Pass SPT Compilation



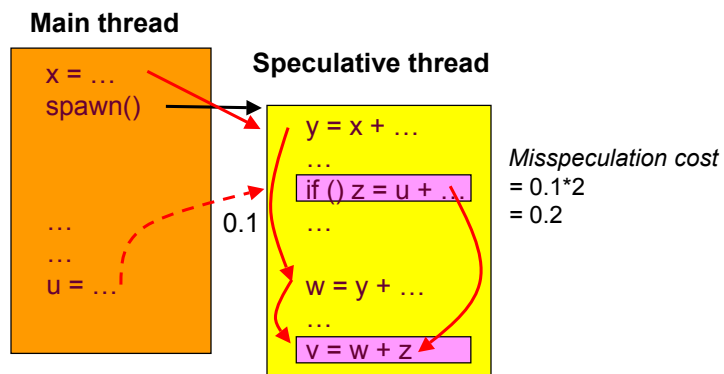
SPT Cost Model

- Based on data/control flow dependences and estimated execution probabilities and costs



Optimal Code Reordering

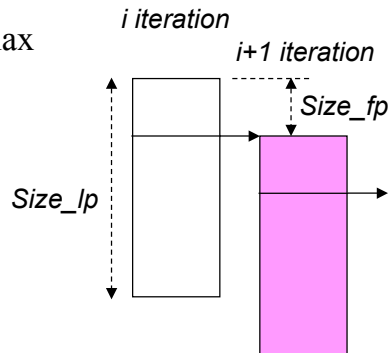
- Reorder code before SPT spawn point to minimize misspeculation cost
- Reduce critical path to spawn SPT threads early



SPT Loop Selection

SPT loop selection criteria:

- Loop body size $> L_{min}$
- Misspeculation cost $< C_{max}$
- Sequential component $< P_{max}$
 $Size_{fp} / Size_{lp} < P_{max}$
- Hardware constraints:
Loop body size $< L_{max}$



Optimal Partitioning for SPT Loops

Problem description:

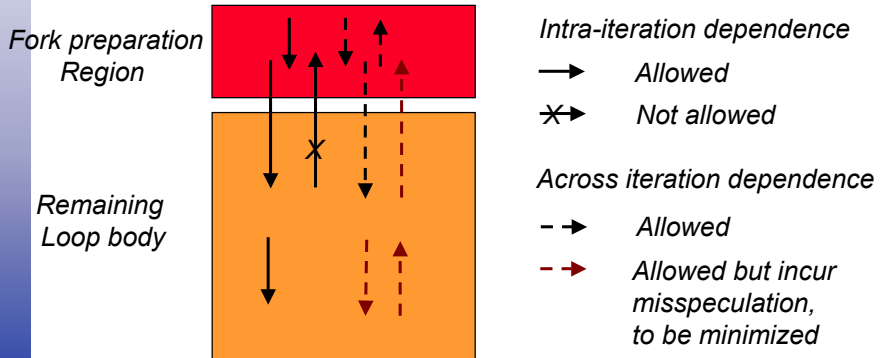
Given a dependence graph of the loop body, find an optimal partitioning, P , such that

- Misspec_cost(P) is minimal

subject to the constraints:

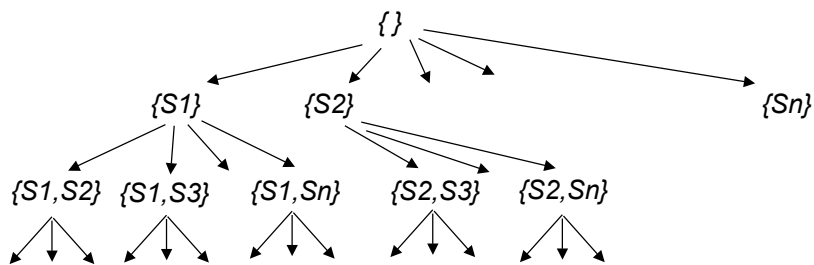
- No unsafe/illegal code reordering
- The fork prep region size $< Size_{fp_max}$

A feasible partition:



Branch and Bound Search for Optimal Partitioning:

- Search tree pruned by fork prep region size
- Cost bounded by min. misspec. cost of a search node
 - Statements are ordered and no preceding statements can be moved into a fork prep region to form a new partition node



Other Enabling Techniques

- Loop unrolling
 - To increase loop size of small loops
- Dependence profiling
 - To obtain more accurate dependence probabilities
- Software value prediction
 - To predict and use critical values w/o hardware support

Dependence Profiling

- Dependence probabilities are essential information for good speculation
 - Used in our misspeculation cost computation
 - Profiling provides a convenient and reliable means to obtain accurate dependence probabilities
- Use the dependence profiling tool provided by U. of Minnesota
 - Instrument and profile every memory references in loops and function calls
 - Feedback profiling information and annotate the dependence graph with dependence probabilities

Software Value Prediction

- Selective value profiling on critical dependences
- Value pattern analysis
- Predictor, check and recovery code generation

```
while (x) {  
    foo(x);  
    x=bar(x);  
}
```



```
pred_x = x;  
while (x) {  
    SPT1:  
    x = pred_x;  
    pred_x = x + 2;  
    SPT_FORK(SPT1);  
    foo(x);  
    x = bar(x);  
    if (x != pred_x) {  
        pred_x = x;  
    }  
}
```

ORC Implementation (1)

ORC Middle End

- A new SPT 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
 - Annotate dependence graph with dependence probabilities from dependence profiling
 - Perform software value prediction
 - For each loop candidate, find its SPT optimal partitioning

ORC Implementation (2)

ORC Middle End

- A new SPT phase in mainopt (cont)
 - Perform final SPT loop selection
 - 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 SPT directives as intrinsic calls

ORC Implementation (3)

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
- LNO: Selective loop unrolling
 - Including outer loop unrolling
- Dependence profiling
 - Instrument and feedback after LNO, before WOPT
 - Propagate to WOPT, from Whirl to SSA

ORC Implementation (4)

ORC Backend

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

ORC Implementation (5)

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

Evaluation

- Simulation of a SPT architecture
 - A 2-core tightly coupled multiprocessor
 - In-order IPF cores
 - One core for the main thread and the other for the speculative thread
 - Shared caches and memory
 - 1024-entry buffer to hold speculative execution results
 - Main thread commits correct speculation results and re-executed misspeculated instructions
 - Itanium2 processor and cache configuration

Speculative Parallel Loops In Spec2000Int

