Software Optimization and Performance Analysis on Intel® Itanium® 2 Processors

David Levinthal
Senior Software Engineer
Intel Corp

Feb 23 2005
Agenda

- **Intel® Itanium® Processor Architecture**
- **Role of Compilers in EPIC**
- **Performance monitoring on Intel® Itanium® 2 Processors**
- **Performance Analysis Methodology**

- **Backup**
  - Resources
  - Architecture review
  - Software pipelining and loop unrolling
  - Non IEEE math functions
  - Full collection of tree diagrams

*Many Comments are Specific to Loop Dominated Codes*
Intel® Itanium® 2 Processor Memory System

External Memory

L3
1-9MB
12-24-way
128B lines
12-17 CLKS

L2
256KB
8-way
128B lines
5-7 CLKS
Banked

L1I
16KB
64B lines
1 CLK

L1D
16KB
64B lines
1 CLK

128 FP Registers
128 General Registers
Core Pipeline
(functional units)

L3 Associativities and Latencies are a function of the L3 size

8.53 GB/s Bandwidth requires 533 MHZ chipset, 6.4 GB/s with 400 MHZ chipset

Intel, the Intel logo, Itanium and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other names and brands may be claimed as the property of others.
Massive Register Set

128 Integer Registers
- NaT 63 0
- GR0
- GR1
- GR31
- GR32
- GR127

128 FP Registers
- 81 81
- FR0 + 0.0
- FR1 + 1.0
- FR31
- FR32
- FR127
- FR127
- 32 Static
- 96 Framed, Rotating

64 Predicate Registers
- PR0
- PR1
- PR15
- PR16
- PR63
- 32 Static
- 96 Rotating

8 Branch Registers
- BR0 63 bit 0
- BR7
- 16 Static
- 48 Rotating

Large number of registers enables flexibility and performance

Intel, the Intel logo, Itanium and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.
Register Stack

GRs 0-31 are global to all procedures
GRs 32-127 stacked registers, local to each procedure
Only GRs implement a register stack
  The FRs, PRs, and BRs are global to all procedures
Register Stack Engine (RSE)
  Upon stack overflow/underflow, registers are saved/restored to/from a backing store transparently

Optimizes the Call/Return Mechanism
Register Stack

Call changes frame to contain only the caller’s output

Alloc sets the frame region to the desired size

- Three architecture parameters: local, output, and rotating
- Return restores the stack frame of the caller

Avoids register spill/fill upon procedure call/return
Speculative Instructions replace Out of Order Engine

Data Speculation

Traditional Architecture

instr 1
instr 2
...
store
load

Barrier

Intel® Itanium® Architecture

ld.a
instr 1
instr 2
...
store

ld.c/chk.a

// Entry in ALAT

// Conflict?

// Check ALAT

Control Speculation

Traditional Architecture

instr 1
instr 2
...
branch
load

Barrier

Intel® Itanium® Architecture

ld.s
instr 1
instr 2
...
branch
chk.s

// Detect exception

Propagate

// Deliver exception

*Other names and brands may be claimed as the property of others.
Core Pipeline

• Efficiently schedules functional units and register resources
  – Front End and Back End coupled by Instruction Buffer
  – Front End and Back End run asynchronously
  – Front End collects instructions and formats them
  – Back End makes sure that instructions and data arrive at functional unit in synch
    • Issue instructions to functional units
    • Resolve exceptions and branch mispredictions
    • Interact with cache and functional unit micropipelines

• Detailed performance analysis reveals the precise pipeline-algorithm-data interaction

• For best performance, design your algorithm and data structures to interact with the core pipeline smoothly
Intel® Itanium® 2 Processor
8 stage Pipeline

- Two stage Front End (FE) gets and formats instructions from L1I cache or the Instruction Streaming Buffer
  - FE loads pipeline Instruction Buffer which stages instructions for Back End
- 6 stage Pipeline Back End
  - Expands the templates (EXP)
  - Prepares registers for access by the instructions (REN)
  - Loads data from registers to functional units (REG)
  - EXE stage invokes instructions and routes output from single cycle ALUs back to REG stage as needed
  - DET stage detects micropipeline stalls, exceptions and branch mispredictions and flushes the pipeline
  - WRB stage writes output of functional units to registers
Intel® Itanium® 2 Processor
8 stage Pipeline

Pipeline Front End

Instruction Buffer

Pipeline Back End

FPU micropipeline

L1D Micropipeline

L1D Cache
Cache Structure

• Very high speed memory for data that gets reused
• Organized into “cache lines”
  – Access of a single element brings in enough adjacent elements to fill the line (64/128 consecutive bytes)
  – Underlying assumption that if you need one element you will need its neighbors soon
• Cache lines are organized into “associative sets” or “ways”
  – Greater associativity allows the hardware more flexibility in cache line replacement algorithms
L1D Cache Structure

<table>
<thead>
<tr>
<th>Intel® Itanium® 2 Processor 16 KB 4-way associative L1 Data Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Integer Data)</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
<tr>
<td>64 byte cache line</td>
</tr>
</tbody>
</table>

L1D has one cycle latency for integer loads:
- All cacheable integer loads go through L1D (write through)
- Use L1D micro pipeline to access general register file
- Cache line row determined by address bits 6 through 11
- 64 associative sets
### L2 Unified Cache

#### Bank Structure

- 256KB, 128 byte cache lines, 8 way associativity
- Each associative set is 1KB, 256 associative sets
- Bank structure allows fast transfers from/to large Intel® Itanium® 2 Processor L2 Cache
- 16 banks each 16 bytes wide

**16 banks cover 256 bytes = 2 cache lines**

<table>
<thead>
<tr>
<th>Line</th>
<th>1</th>
<th>Of</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>3</td>
<td>Of</td>
<td>8</td>
</tr>
<tr>
<td>Line</td>
<td>5</td>
<td>Of</td>
<td>8</td>
</tr>
<tr>
<td>Line</td>
<td>7</td>
<td>Of</td>
<td>8</td>
</tr>
</tbody>
</table>

**cache lines**
L2 Unified Cache Bank Structure

- 256KB, 128 byte cache lines, 8 way associativity
- Each associative set is 1KB, 256 associative sets
- Bank structure allows fast transfers from/to large Intel® Itanium® 2 Processor L2 Cache
- 16 banks each 16 bytes wide
  16 banks cover 256 bytes = 2 cache lines

<table>
<thead>
<tr>
<th>Line</th>
<th>Of</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bank 0 is a column of 1024 16 byte elements
Intel® Itanium® 2 Processor
L2 Cache Access

• L2 data access controlled by 32 entry queue (OzQ) and allows out of order data return
  – FP data loaded to FP register file directly from L2
• Minimum integer latency is 5 cycles
• Minimum floating point latency is 6 cycles
• Latency is increased by:
  – Cache miss
  – Bank conflicts cause OzQ cancels (measured to add 6 cycles)
  – Multiple misses and misses to lines being updated will cause OzQ recirculates (measured to add ~17 cycles @ 1Ghz processor)
    • Only one data access is escalated to L3 and the system bus, the others recirculate
Virtual to Physical Address Translation and DTLB Hierarchy

- Access to physical memory requires virtual to physical address translation
  - OS creates Virtual Hash Page Table (VHPT) to map these translations
- Hardware Page Walker (HPW) accesses VHPT to enable physical memory access
- Two Level Data Translation Lookaside Buffer (DTLB) act as caches for VHPT data
  - DTLB entries required for a cache hit
- Accessing many pages simultaneously may access more DTLB entries than are available
  - Cache misses and DTLB updates cause memory access stalls

Data Access must use DTLB Hierarchy Efficiently
DTLB Hierarchy

- **L1 DTLB**: 32 entries
- **L1 Data Cache**: Hit 1 cycle latency
DTLB Hierarchy

L1 DTLB
32 entries

L1 Data Cache

L2 DTLB
128 total entries

OS

L2 Cache

load

Miss

Transfer

Update Cache Line

Hit

9 cycle Latency
DTLB Hierarchy

- L1 DTLB: 32 entries
  - Load
  - Miss
  - Transfer
- L1 Data Cache
- Update Cache Line
- Hit: 22 cycle Latency

- L2 DTLB: 128 total entries
  - Transfer
  - Miss
- VHPT Pages
- OS
- HPW
- VHPT 8 bytes/entry
- OS managed

*Other names and brands may be claimed as the property of others.*
Agenda

• **Software pipelining and loop unrolling**
• Role of Compilers in EPIC
• Performance monitoring on Intel® Itanium® 2 Processors
• Performance Analysis Methodology

• Backup
  – Resources
  – Architecture review
  – Non IEEE math functions
  – Full collection of tree diagrams

Many Comments are Specific to Loop Dominated Codes
**Intel® Itanium® 2 Processor**  
**Software Pipelining**

Intel® Itanium® 2 Processors invoke software loop pipelining through rotating registers  
Absorbing latency through multiple loop iterations  
Predication creates pipeline prologues and epilogues  
Single instruction stream for  
Pipeline loading (prologue)  
Loop body (kernel)  
Pipeline draining (epilogue)
Rotating Registers

For (l=0; l<MAX; l++) sum += a[l];

Loop:
  ldfs f32 = [r32], 4
  fma f46 = f50, f1, f38
  br.ctop.sptk loop;;
Rotating Registers

For \( I = 0; I < \text{MAX}; I++ \) sum += \( a[I] \);

Loop:

- \( \text{ldfs f32 = [r32]}, 4 \)
- \( \text{fma f46 = f50, f1, f38} \)
- \( \text{br.ctop.sptk loop} ;; \)
Rotating Registers

For(\(i=0; i<\text{MAX}; i++\)) \(\text{sum} += a[i]\);

Loop:

\[
\text{ldfs } f32 = [r32], 4
\]
\[
\text{fma } f46 = f50, f1, f38
\]
\[
\text{br.ctop.sptk loop;}
\]
Predication Creates Prologue and Epilogue

For(I=0;I<MAX;I++)sum += a[I];

Loop:

(p16) ldfs f32 = [r32],4
(p22) fma f46 = f50,f1,f38
br.ctop.sptk loop;;
Predication Creates Prologue and Epilogue

For(I=0;I<MAX;I++)sum += a[I];

Loop:
(p16) ldfs f32 = [r32],4
(p22) fma f46 = f50,f1,f38
br.ctop.sptk loop;;

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p16</td>
<td>1</td>
</tr>
<tr>
<td>p17</td>
<td>1</td>
</tr>
<tr>
<td>p18</td>
<td>1</td>
</tr>
<tr>
<td>p19</td>
<td>0</td>
</tr>
<tr>
<td>p20</td>
<td>0</td>
</tr>
<tr>
<td>p21</td>
<td>0</td>
</tr>
<tr>
<td>p22</td>
<td>0</td>
</tr>
<tr>
<td>p23</td>
<td>0</td>
</tr>
</tbody>
</table>
Predication Creates Prologue and Epilogue

For(I=0;I<MAX;I++)sum += a[I];

Loop:
(p16) ldfs f32 = [r32],4
(p22) fma f46 = f50,f1,f38
br.ctop.sptk loop;;

<table>
<thead>
<tr>
<th>p16</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>p17</td>
<td>1</td>
</tr>
<tr>
<td>p18</td>
<td>1</td>
</tr>
<tr>
<td>p19</td>
<td>1</td>
</tr>
<tr>
<td>p20</td>
<td>1</td>
</tr>
<tr>
<td>p21</td>
<td>1</td>
</tr>
<tr>
<td>p22</td>
<td>1</td>
</tr>
<tr>
<td>p23</td>
<td>0</td>
</tr>
</tbody>
</table>
Unroll and SWP are the keys to performance in HPC codes

• Performance requires parallelism
  – Requires loop unrolling
  – Requires SWP to hide latency

• Dominant reason compiler fails to do this automatically is pointer disambiguation
Loop Unrolling

Loop unrolling allows two main benefits

• More effective use of functional units
  – More data and operations manipulated/iteration

• Data Reuse
  – Some elements in the kernel expressions will remain in registers and only loaded once

For example in a matrix multiply the right most matrix is the one loaded $N^3$ times

Reusing elements is the key to reducing main memory access
Limits of Loop Unrolling

There are three main limits to loop unrolling

• Functional units
  – Two Floating point units on Intel® Itanium® 2 Processors

• Cache sizes
  – Unrolling may force more cache usage
  – Each cached row (8b * N = 8*1024b) uses 8KB

• Register resources (< 96 rotating registers)
  – MM requires 3 GR and 3 FP registers/line
  – Times unrolling
  – Times depth of pipelining

This limits unrolling to 8 – 16 times
Identifying well Optimized Loops from Assembler Listings

Identifying SW Pipelining

- A SW pipelined loop will have multiple stages
  - Minimally reading data, calculating and storing results
  - Loop ends with a br.ctop, br.cexit or br.wtop, br.wexit
- Stages are denoted by the predicate registers that enable them
  - Therefore there should be several different predicates used in the loop
Identifying well Optimized Loops from Assembler Listings

• SW Pipelining is used to absorb latency
  – Scheduled latency is number of cycles/iteration (number of ;; times the difference between ld predicates and consumption (ex fma) predicates

• Be suspicious of pipelined loops that use advanced loads (ex ld4.a)
  – Pointers were not disambiguated
  – Limited optimization was invoked
Agenda

- Role of Compilers in EPIC
- Performance monitoring on Intel® Itanium® 2 Processors
- Performance Analysis Methodology

- Backup
  - Resources
  - Architecture review
  - Software pipelining and loop unrolling
  - Non IEEE math functions
  - Full collection of tree diagrams

Many Comments are Specific to Loop Dominated Codes
The Role of compilers

• Intel® Itanium® Processor instruction scheduling is explicit
  – What you see is what you get
  – Speculative and prefetch instructions replace O-O-O engine

• Sophisticated compilers are essential
  – Needed immediately to enable new architecture’s acceptance
  – Excellent compiler instruction scheduling
Compiler Development and Performance Monitoring

- Compiler generated performance = effective instruction scheduling
- Compiler development requires powerful performance monitoring capabilities
  - Quickly identify performance improvement opportunities
- Intel® Itanium® processor family has a radical performance monitoring architecture
  - Enabling rapid software performance development

Intel® Itanium® 2 Processor Performance Monitoring Unit was Explicitly Designed to Enable SW Optimization and Compiler Development
Agenda

• Role of Compilers in EPIC
• **Performance monitoring on Intel® Itanium® 2 Processors**
• Performance Analysis Methodology

• Backup
  – Resources
  – Architecture review
  – Software pipelining and loop unrolling
  – Non IEEE math functions
  – Full collection of tree diagrams
Performance Analysis on Intel® Itanium® 2 Processors

• Cycle accounting identifies processor pipeline stalls
  – Performance Monitoring Units (PMU) can determine exactly how CPU cycles are used
• Vtune™ Analyzer determines where (IP) cycles are being consumed but also exactly why
  – Exact cause and location of execution inefficiency can be identified

Unique Features of the Intel® Itanium® Processor Family
Event Based Sampling with VTune™ Performance Analyzer

• Overflow of an Intel® Itanium® 2 Processor Performance Monitoring Unit (PMU) counter raises a fault
  – OS handles the fault
  – Exception handler is the VTune™ Analyzer driver

• Intel® Itanium® 2 Processor PMU has 4 counters
  – It can count 4 events simultaneously
    • Some events cannot be counted at the same time
  – ~ 500 performance events can be counted
    Ignores privilege selection and programming of branch event
  – VTune™ Analyzer groups events into compatible sets
    • Takes multiple runs to collect all the data
Event Based Sampling with VTune™
Performance Analyzer

- VTune™ Analyzer “samples” various data with its driver
  - IIP, time stamp, Process ID, Thread ID + other relevant data
    - Interrupt IP has added inaccuracy from OS response time (SKID)
- Driver sets/resets the PMU counter to overflow value – “Sample After Value” (SAV)
  - results in a fault being raised every “SAV” events
  - Driver records IIP of the fault + “other data”
- VTune™ Analyzer has 2 modes for setting the SAV
  - Default: SAV calibration run is used to determine an SAV/event that results in interrupt rate of 1Khz/event
    - Second run takes performance event samples
  - User can set the SAV by hand and disable the calibration
    - Requires half the “runs”
    - Preferred mode of the author
Agenda

• Role of Compilers in EPIC
• Performance monitoring on Intel® Itanium® 2 Processors
• **Performance Analysis Methodology**

• Backup
  – Resources
  – Architecture review
  – Software pipelining and loop unrolling
  – Non IEEE math functions
  – Full collection of tree diagrams
Follow the Cycle Accounting Tree

Cycles Retiring Instr

CPU_CYCLES

BACK_END_BUBBLE.ALL

Cycle Accounting

Stall Cycles are the Basis of the Analysis
Follow the Cycle Accounting Tree

Cycle Accounting

Stall Cycles Decompose to Architectural Cause

- CPU_CYCLES
- BACK_END_BUBBLE.ALL
  - BACK_END_BUBBLE.FE
  - BE_RSE_BUBBLE
  - BE_EXE_BUBBLE
  - BE_L1D_FPU_BUBBLE
  - BE_FLUSH_BUBBLE

Decreasing Priority

Cycles Retiring Instr

Intel, the Intel logo, Itanium and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.*
Follow the Cycle Accounting Tree

Cycles Retiring Instr

CPU_CYCLES

BACK_END_BUBBLE.ALL

BE_FLUSH_BUBBLE

BE_L1D_FPU_BUBBLE

BE_EXE_BUBBLE

BE_RSE_BUBBLE

BACK_END_BUBBLE.FE

Compile with Profile Guided Feedback

Gather More Detail

Reduce Register Usage

Compile with Profile Guided Feedback

Cycle Accounting

Source of Stalls Prescribes Solution

*Other names and brands may be claimed as the property of others.*
Follow the Cycle Accounting Tree

Precision of Detail allows Precision of Optimization
Cycle Accounting Tree Identifies the Relevant Architectural Events

Cycles Retiring Instr

CPU_CYCLES

BACK_END_BUBBLE.ALL

BE_FLUSH_BUBBLE

BE_L1D_FPU_BUBBLE

BE_EXE_BUBBLE

BE_RSE_BUBBLE

BACK_END_BUBBLE.FE

L1D_Read_Misses

L2_Data_References

L3_ Reads.Data_read.
  All

L3_ Reads.Data_read.
  Hit

L3_ Reads.Data_read.
  Miss

L2_OzQ_Cancel1.
  Bank_Conf

L2_Force_Recirc

Dear_Latency_ge_32

Architectural Events

Stall Cycles are Associated with Architectural Events
Methodology, Step by Step

• Determine most important modules
  – Amdahl’s Law always wins
• Identify nature of execution inefficiency
  – Use cycle accounting tree to determine dominant sources
• Collect data on relevant architectural events
  – Dominant source of stall cycles defines which ones
  – Weight events by penalties to determine which are dominant sources
• Apply the appropriate optimization technique
• Iterate in this manner until performance is acceptable

Intel Compiler Development uses this Methodology
What is Missing?
Long Latency Loads

• L3 cache misses due to loads (and stores) cause hundreds of stalled cycles
  – L3_reads.data_reads.miss is caused by loads, stores and PREFETCH instructions
• Data ear events sample loads
  – Allow user to select minimum latency (8, 16, 32, 64…cycles)
  – Minimum latency > 32/64 identifies unprefetched data loads
  – The author’s averaging over a few loop dominated codes suggests a sampling fraction of 0.25 – 0.33
    • EAR sampling fraction WILL vary with application

Event Address Register Performance Monitoring is Unique to Intel® Itanium® Processor Family
Using Data EAR events

• EAR events are exact
  – Triggered by load instructions
  – IP recorded by HW
  – Data EAR records actual data delivery latency
    • Uses its power of two threshold only to trigger event for counting
  – VA is recorded by HW

• Allows unambiguous localization of long latency data accesses

• These events are exact in the disassembly view at all optimizations
  – Not in the source view at high optimization
What is Missing?
Long Latency Loads

• Dear_Latency_gt_64 with SAV = 1000
  – Identifies loads with latencies that cause pipeline stalls
  – Dominated by L3 cache misses
    • Identifying unprefetched data

• Penalty Estimate
  Penalty ~ 1000(sav) * 3 (dear sampling fraction) * 300 cycles (assume all are L3 Miss)
  * Number of Samples

(very rough)

Event Address Register Performance Monitoring is Unique to Intel® Itanium® Processor Family
What is Missing?
Bandwidth limitations

- There are many memory bus events
  - ~90
- Loop dominated codes can use Bus_memory.all.self (SAV = 10,000)
  - Take cpu_cycles with SAV = 1,000,000
- Bus cycles/Bus transaction > 4 (6.4GB/sec)
  - 4 bus cycles to transfer a cacheline 32 Bytes/cycle (double pumped)
  - Practical limit around 5.2 due to changing bus direction

\[
\text{Bus_cyc/Bus_trans} = \frac{\text{cpu_cycles} \times (0.200/\text{core_freq_in_Ghz})}{\text{Bus_mem.all.self}} \times 100(\text{sav ratio})
\]

(use sample counts, for 533MHz FSB use 0.267 instead of 0.200)

If This Number Approaches ~6-7, you may have a problem
Overcoming Bandwidth

• Bandwidth limitations in loops have several resolutions
• Decrease cachelines transferred
  – Modify structures so only data being used is retrieved from memory
  – Structures of arrays rather than arrays of structures
• Retrieve identical cachelines as few times as possible
  – Merge loops using same cachelines/data
  – Do as much work with data while you have it in the core
• Hide the latency
  – Merge functional unit limited loops with BW limited loops
    • Requires known relationship between trip counts
    • Ex: merge a loop with a costly function (mod, divide, sqrt) with a loop that zeros an array
The Big Four, The Biggest Bang for the Buck

- CPU_CYCLES (SAV = 1,000,000)
- BACK_END_BUBBLE.ALL (SAV = 1,000,000)
- DEAR_LATENCY_GT_64 (SAV = 1000)
- BUS_MEMORY.ALL.SELF (SAV = 10,000)
  - Disable Calibration
- Identifies
  - Low hanging fruit with back_end_bubble.all
  - Hotspots (cpu_cycles / back_end_bubble.all)
  - Long latency data access/non prefetched data/thread sharing problems
  - Bandwidth limited loops
Agenda

• Role of Compilers in EPIC
• Performance monitoring on Intel® Itanium® 2 Processors
• Performance Analysis Methodology
• Non IEEE math functions

• Backup
  – Resources
  – Architecture review
  – Software pipelining and loop unrolling
  – Full collection of tree diagrams

Many Comments are Specific to Loop Dominated Codes
Divides, Sqrt’s and FP Scoreboard Stalls

- FP Scoreboard dependency stalls result from chained FP ops
- These are mostly due to divides, sqrt’s and complex functions
- There is no “be_exe_bubble.frfr” counter to identify this contribution explicitly.

How do You Identify FP Scoreboard Dependency Stalls?
FMA’s and Newton-Raphson Approximations

- Divides and Sqrt functions are calculated in SW
- These take advantage of “infinite” accuracy of intermediate state of the fma

\[ y = \frac{1 + e}{b} \] provided by frcpa instr
\[ d = -e = 1 - y*b \] fnma
\[ \frac{1}{b} = \frac{y}{1 + e} \] which is what we want

\[ = y \times (1 - e + e^2 - e^3 + e^4 - e^5 \ldots) \]
\[ = y \times (1 + d + d^2 + d^3 + d^4 + d^5 \ldots) \]

Forms the basis of divides and Sqrt’s…
Divides, SQRT’s and “FRFR” stalls

Consider the following assignment loop

```c
void daxpy(int len, double * restrict a, double *b)
{
    int i;
    for(i=0; i<len; i++) a[i] = 1.0/ b[i];
    return;
}
```
SW reciprocals, Sqrt’s in asm

Note that *.s1 FP Environment is always used
Estimating FP Scoreboard Dependency Stalls

- Count frcpa and frsqrta instructions
  - Op code match IA64_Instr_Retired
  - Multiply by approximate penalties
- Use Vtune™ Analyzer to collect stall cycles
  - “Export” source view when viewing disassembler as “csv” file
  - Sum stall cycles associated with fma.s1, fnma.s1 etc type instructions
  - This can be done with simple excel script

Two Techniques Available
Throughput vs Latency algorithms

• By default the 7.0/7.1 compilers use the minimum latency encoding of the division algorithm
• There is another algorithm that has higher latency but allows greater throughput in SWP loops
  – -mP3OPT_ecg_thruput_for_div=T

In a loop like
For(i=0;i<len;i++)a[i]=x/b[i];
Timing decreases from 7.1 to 5.1 cycles/iter

Intel® 8.0/8.1/9.0 Compiler uses throughput algorithm in SWP loops, latency algorithm everywhere else
Non IEEE Compliant Algorithms can Improve Timing (7.0 compiler)

Using simple functions of the type:
\[ \text{for}(i=0; i<\text{len}; i++) a[i] = \text{func}(b[i]); \]

The penalties for divides and Sqrts and the impact of non_ieee algorithms are approximated

<table>
<thead>
<tr>
<th>function</th>
<th>Default</th>
<th>Stall cycles</th>
<th>NoIEEE</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td></td>
<td>1.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>recip</td>
<td></td>
<td>7.1 5.4 cycles/call</td>
<td>4.1</td>
<td>3 cycles/call</td>
</tr>
<tr>
<td>div</td>
<td></td>
<td>7.1 5.4 cycles/call</td>
<td>4.3</td>
<td>2.8 cycles/call</td>
</tr>
<tr>
<td>recip_sqrt</td>
<td></td>
<td>15.1 13.4 cycles/call</td>
<td>8.6</td>
<td>6.5 cycles/call</td>
</tr>
<tr>
<td>sqrt</td>
<td></td>
<td>14.6 12.9 cycles/call</td>
<td>7.6</td>
<td>7 cycles/call</td>
</tr>
</tbody>
</table>

Throughput division encoding took 5.1 cycles/iteration

These numbers were measured with a 1Ghz Intel® Itanium® 2 Processor
Non IEEE Compliant Algorithms can Improve Timing (8.1 compiler)

Using simple functions of the type:

for(I=0;I<len;I++)a[I]=func(b[I]);

The penalties for divides and Sqrts and the impact of non_ieee algorithms are approximated

<table>
<thead>
<tr>
<th>function</th>
<th>Default</th>
<th>~Stall cycles</th>
<th>NoIEEE</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recip</td>
<td>5.1</td>
<td>3.8 cycles/call</td>
<td>3.6</td>
<td>1.5 cycles/call</td>
</tr>
<tr>
<td>div</td>
<td>5.1</td>
<td>3.8 cycles/call</td>
<td>4.1</td>
<td>1.0 cycles/call</td>
</tr>
<tr>
<td>recip_sqrt</td>
<td>7.1</td>
<td>5.8 cycles/call</td>
<td>6.1</td>
<td>1.0 cycles/call</td>
</tr>
<tr>
<td>sqrt</td>
<td>11.0</td>
<td>9.7 cycles/call</td>
<td>7.1</td>
<td>3.9 cycles/call</td>
</tr>
</tbody>
</table>

Throughput division encoding took 5.1 cycles/iteration (now default for loops)

These numbers were measured with a 1.6Ghz Intel® Itanium® 2 processor
Accuracy of Non IEEE Flags

- Accuracy is defined in terms of ULP
  - ulp stands for Unit-in-the-Last-Place
  - Meaning 1 bit in the mantissa
- Standard algorithms are good to better than ½ ULP
- These non IEEE algorithms are good to better than 1 ULP

the error is small and well contained, not like the "dirty" divides/sqrts on some other architectures. where you might only get half of the bits in the significand right
Non IEEE Compliant Algorithms can Improve Timing

There is some loss of accuracy

- But not much!

-IPF_fp_relaxed
Identifying Bank Conflicts

- Identifying exactly which variables are causing bank conflicts can be tricky
- Use VTune Analyzer to identify loops with performance affected by bank conflicts
  - Then use asm view to identify which sets of registers are accessed on the single cycles
- Invoke a debugger that can display all relevant registers
  - Put a break point at beginning of loop
  - Loop through 5-10 iterations
  - Inspect addresses held in the registers accessed on the same cycle
  - Look for values with second hex digit being identical

Virtually Never Happens in 8.1/9.0 Compilers
Unless YOU have 256 byte aligned arrays
Compiler Insights

- (Q) IPO can invoke whole program optimization
  - Automatically disambiguate pointers
  - Other advantages
- Requires entire scope of application
- Shared Objects defeat this
  - No possibility of knowing pointers passed from external call
  - Even for internal only variables
  - DLL import/export syntax MAY provide some relief???
Let the Compiler do the Work

• Going to /O3 optimization will invoke more aggressive optimizations
  – Better instruction scheduling

• **Disambiguate your Pointers!!!!!!**
  • Profile guided feedback (prof_gen/prof_use) and Interprocedural Optimization (IPO) can be very effective in addition

• Use performance monitoring to guide your effort
Intel® Software Development Products

• Intel® Compilers
  – Easiest way to take advantage of Intel Architecture performance features
  – Compatible with leading Windows* and Linux* development environments

• Intel® VTune™ Performance Analyzer
  – Saves time in development cycle by identifying “hot spots” in source code
  – Low intrusion sampling methods provide most accurate results

• Intel® Cluster Tools
  – Optimization of MPI usage with minimal impact

• Intel® MPI Library
  – Optimization of MPI interconnect with minimal impact

• Intel® Performance Libraries
  – Run-time processor detection loads the best code for the processor
  – Reduces the need to write processor-specific code thereby enabling

• Intel® Thread Checker and Thread Profiler
  – Automatic parallel programming error detection and performance analysis

*Other names and brands may be claimed as the property of others.
Summary

• Use SW Development tools in a coordinated fashion
  – Use Performance Analysis to guide Optimization (Cpu_Cycles, Back_End_Bubble)
  – Invoke O3 for routines with significant cycles
    • Disambiguating pointers immediately in those functions
    • Then eliminate remaining long latency loads identified by Data EAR (after O3)
  – Iteratively remove remaining stall cycles

Performance analysis on Intel® Itanium® Processors is powerful!
What you learn there can be applied everywhere