IA-64 Compiler Technology

David Sehr, Jay Bharadwaj, Jim Pierce, Priti Shrivastav (speaker), Carole Dulong

*Microcomputer Software Lab*
Introduction

IA-32 compiler optimizations

- Profile Guidance (PGOPTI)
- Inter-procedural Optimizer (IPO)
- Machine independent optimizations
- Vectorizer
- Machine dependent optimizations (Strength reduction, Code Scheduling)

IA-64 compilers are focused on optimizations increasing instruction level parallelism
Compiler Optimizations designed to achieve the best performance on IA-64

Agenda

- Overview of Intel’s IA-64 Compiler
- Machine independent optimizations
- IA-64 code samples
- IA-64 specific optimizations

IA-64 Compiler technology is ready, and exploits the architecture
Overview of Intel’s IA-64 compiler

Compiler Architecture

- C++ Front End
- FORTRAN 90 Front End

- Profile Guidance (PGOPTI)
- Inter-procedural Optimizer (IPO)
- High-Level Optimizer (HLO)
- Global Optimizer (IL0)
- Code Generator (ECG)
Overview of Intel’s IA-64 compiler
Global Optimizer (IL0)

- Loop Unrolling
- Register Variable Detection
- Constant Prop.
- Convert Opt.
- Strength Reduction
- Copy Propagation
- Re-association
- Redundancy Elim.
- Dead Store Elim.
- Dead Code Elim.
Overview of Intel’s IA-64 compiler

Code generator (ECG)

- Machine instruction lowering
- Software Pipelining (rotating registers, loop branches)
- Predication (parallel compares)
- Global scheduling (control and data speculation, multi-way branches)
- Block ordering (branch hints)
- Global register allocation (register stack, ALAT, UNAT)
- Function splitting (I cache and TLB locality)
Compiler Optimizations designed to achieve the best performance on IA-64

Agenda

- Overview of Intel’s IA-64 Compiler
- Machine independent optimizations
- IA-64 code samples
- IA-64 specific optimizations
Machine independent optimizations

- Profile guided optimizations (PGOPTI)
- Inter-procedural optimizations (IPO)

The larger the compilation scope, the better the performance
Machine independent optimizations
Profile guided optimizations (PGOPTI)

**Execution profiling:**
- Profile feedback
  - Compile once with counters inserted.
  - Run the profiled program with a sample input set.
  - Compile again, using the counter values.

**Its effects:**
- Branch probabilities guide
  - Code generator region.
  - Predication regions.
- Execution frequencies guide
  - Procedure inlining/partial inlining
  - Code placement
  - Speculation heuristics

Profile information key to get best out of predication and speculation techniques
Machine independent optimizations
Inter-procedural optimizations (IPO)

Procedure Integration:
- **Inlining**
  - Copying a function body into a call site.
- **Partial inlining**
  - Copy the *hot* portion of a function into a call site.
  - Remainder becomes a *splinter* function.
- **Cloning**
  - Specializing a function to a specific class of call sites.

*Its effects:*
- Exact interprocedural information for a specific call site.
- Larger code region to schedule
  - Spreads function boundaries

Larger scope enables better scheduling and register allocation
Machine independent optimizations
Inter-procedural optimizations (IPO)
Procedure integration: Inlining

**BEFORE:**
void func1()
{
    int i;
    for (i=0;…)
        func2(i);
}

void func2(int x)
{
    a[x] = 1.0;
}

**AFTER:**
void func1()
{
    int i;
    for (i=0;…)
        a[i] = 1.0
}

Eliminates function call overhead
Machine independent optimizations
Inter-procedural optimizations (IPO)
Procedure integration: Partial inlining

BEFORE:
void func(tree *p) {
    func2(p);
}

void func2(tree *p) {
    if (p->left == NULL)
        return;
    ..........  
    printf(...);
}

AFTER:
void func(tree *p) {
    if (p->left == NULL)
        return;
    else {
        splinter(p);
    }
}
void splinter(tree *p) {
    ..........  
    printf(...);
}
**Machine independent optimizations**

**Inter-procedural optimizations (IPO)**

**Procedure integration: Cloning**

**BEFORE:**
```c
void func1()
{
    func2(1, n);
    func2(1, m);
    func2(i, m);
}
void func2(int i, int j)
{
    if (i == 0)
        // do something
    else
        // do something else
}
```

**AFTER:**
```c
void func1()
{
    func2_0(1, n);
    func2_0(1, m);
    func2(i, m);
}
void func2_0(int i, int j)
{
    // do something
}
```

**Eliminate branches**
Machine independent optimizations
Inter-procedural optimizations (IPO)

*Inter-procedural Analysis:*
- Alias propagation
- Mod/ref propagation
- Constant propagation

*ITS EFFECTS:*
- Alias propagation
  - Indirect call conversion
- Better disambiguation
  - Better scheduling, speculation
- Mod/ref propagation
  - Improves registerization
- Constant propagation
  - Makes constant parameters and globals explicit
  - Remove unnecessary conditional code

IPO improves many classical scalar optimizations
Compiler Optimizations designed to achieve the best performance on IA-64

Agenda

- Overview of Intel’s IA-64 Compiler
- Machine independent optimizations
- IA-64 code samples
- IA-64 specific optimizations
IA-64 code samples

- Strength reduction
- Post increments

Techniques to leverage IA-64 architecture features
IA-64 code samples
Strength Reduction

WHEN IT APPLIES:
- One operand is a loop invariant.
- Other operand is an induction variable.

ITS EFFECTS:
- Multiplication is replaced by addition.
- Better chances for post-increment creation.

Strength reduction key for optimizations of loops
IA-64 code samples
Post-Increment
Loads and Stores

**WHEN IT APPLIES:**
- Memory address is modified after memory reference.

**ITS EFFECTS:**
- Addition proceeds in parallel with load/store.
- No extra instructions required for induction variable update.

Post increment increases parallelism
SOURCE:
FUNCTION FUNC(A,B,N)
REAL A(N,N), B(N,N)
DO I = 1, N
   A(1,I) = B(1,I)
RETURN
END

Algorithm:
B(1,I) = base of B +
size of element * (I-1)

ASSEMBLY:
… r35 = base of B
loop:
setf.sig f32 = r32
setf.sig f33 = r33
xma f34 = f32,f33
getf.sig r34 = f34
…
add r36 = r34, r35
ld f34 = [r36]
…
br …
IA-64 code samples
Strength Reduction

**BEFORE:**

- \( \ldots \) \( r_{35} = \text{base of } B \)
- loop:
  - setf.sig \( f_{32} = r_{32} \)
  - setf.sig \( f_{33} = r_{33} \)
  - xma \( f_{34} = f_{32}, f_{33} \)
  - getf.sig \( r_{34} = f_{34} \)
  - \( \ldots \)
- add \( r_{36} = r_{34}, r_{35} \)
- ld \( f_{34} = [r_{36}] \)
- \( \ldots \)
- br \( \ldots \)

**AFTER:**

- \( \ldots \) \( r_{36} = \text{base of } B \)
- \( \ldots \) \( r_{35} = \text{stride of } B \)
- loop:
  - ld \( f_{34} = [r_{36}] \)
  - add \( r_{36} = r_{36}, r_{35} \)
  - \( \ldots \)
  - br \( \ldots \)

Strength reduction replaces multiplication with additions
IA-64 code samples
Strength Reduction+post increment

**BEFORE:**

```
... r36 = base of B
... r35 = stride of B
loop:
  ld f34 = [r36]
  add r36 = r36, r35
... br ...
```

**AFTER:**

```
... r36 = base of B
... r35 = stride of B
loop:
  ld f34 = [r36],r35
... br ...
```

Post increment replaces addition
Compiler Optimizations designed to achieve the best performance on IA-64

Agenda

- Overview of Intel’s IA-64 Compiler
- Machine independent optimizations
- IA-64 code samples
- Machine specific optimizations
Machine specific optimizations

- Predication
- Software Pipelining
Machine specific optimizations

Predication

Objective
- Eliminate hard to predict branches.
- Increase parallelism
- Reduces critical path

A Side effect
- Reduce register usage (as compared to speculation)

Eliminates branches, increases parallelism
Machine specific optimizations

Predication with parallel compare

Objective
- Reduce control height
- Reduce number of branches

If (a>b && b>c)
then Y
else X

Parallel compare reduces control height
Machine specific optimizations
Multi way branches with Predication

- Use Multi way branches
  - Speculate compare (i.e. move above branch)
  - Avoids predicate initialization
  - More than 1 branch in a single cycle
  - Allows n-way branching

If (a>b && b>c)
then Y
else X

Predication and Multi-way increase ILP
Machine specific optimizations
Software Pipelining: Loop Example

Convert string to uppercase

```c
for (i=0, i< len, i++) {
    if (IS_LOWER_CASE(line[i]))
        newline[i] = CNVT_TO_UPPERCASE(line[i]);
    else
        newline[i] = line[i];
}
```

After macro expansion

```c
for (i=0, i< len, i++) {
    if (line[i] >= 'a' && line[i] <= 'z')
        newline[i] = line[i]-32;
    else
        newline[i] = line[i];
}
```

Typical integer-type loop
Machine specific optimizations
Software Pipelining

- Overlapping execution of different loop iterations

vs.

Whole loop computation in one cycle

Overlap iterations for parallelism
Machine specific optimizations
Software Pipelining

Cycle
1   ld
2   ld  cmps
3   ld  cmps  sub
4   ld  cmps  sub  st  **Kernel**

Input
ld  cmps  sub  st

Output
Machine specific optimizations
Software Pipelining: introducing Rotating Registers

- GR 32-127, FR 32-127 can rotate
- Separate Rotating Register Base for each: GRs, FRs
- Loop branches decrement all register rotating bases (RRB)
- Instructions contain a “virtual” register number
  - RRB + virtual register number = physical register number.
Machine specific optimizations

Software Pipelining: Pipelined Loop

Kernel code

loop:

s1
  ld r34 = [ra], 1
  cmp p1 = true

s2
  cmp.and p1 = (r35>96)
  cmp.and p1 = (r35<123)

s3 (p1) sub r36 = r36, 32

s4
  st [rb] = r37, 1

br.ctop loop
Machine specific optimizations
Software Pipelining: Fill the pipe ...

Execute prologue stage

Kernel code

loop:
  ld r34 = [ra], 1
  cmp p1 = true
  cmp.and p1 = (r35>96)
  cmp.and p1 = (r35<123)
  (p1) sub r36 = r36, 32
  st [rb] = r37, 1
  br.ctop loop

Intel Labs
Machine specific optimizations
Software Pipelining: Fill the pipe ...

Perform a loop branch
• Decrement lc
• Rotate registers by decrementing RRB
Machine specific optimizations
Software Pipelining: Fill the pipe ...

Execute prologue stage

Kernel code

loop:
  ld r34 = [ra], 1
  cmp       p1 = true
  cmp.and  p1 = (r35>96)
  cmp.and  p1 = (r35<123)
  (p1) sub r36 = r36, 32
  st [rb] = r37, 1
  br.ctop loop
Machine specific optimizations
Software Pipelining: Fill the pipe ...

Execute prologue stage

Kernel code

loop:
  ld r34 = [ra], 1
  cmp    p1 = true
  cmp.and p1 = (r35>96)
  cmp.and p1 = (r35<123)
(p1) sub r36 = r36, 32
  st [rb] = r37, 1
  br.ctop loop

Intel Labs
Machine specific optimizations
Software Pipelining: Execute the Kernel

Physical register file

Virtual register

Execute kernel
Whole iteration per cycle

Kernel code

loop:
  ld r34 = [ra], 1
  cmp         p1 = true
  cmp.and  p1 = (r35>96)
  cmp.and  p1 = (r35<123)
  (p1) sub r36 = r36, 32
  st [rb] = r37, 1
  br.ctop loop

Intel Labs
Machine specific optimizations
Software Pipelining: Execute the Kernel

Execute kernel
Whole iteration per cycle

Kernel code

loop:
  ld r34 = [ra], 1
  cmp p1 = true
  cmp.and p1 = (r35>96)
  cmp.and p1 = (r35<123)
  (p1) sub r36 = r36, 32
  st [rb] = r37, 1
  br.ctop loop

Physical register file

r36 = l
r37 = B
r32 = _
r33 = O
Machine specific optimizations
Software Pipelining: Execute the Kernel

Execute kernel
Whole iteration per cycle

Kernel code

loop:

    ld  r34 = [ra], 1
    cmp         p1 = true
    cmp.and  p1 = (r35>96)
    cmp.and  p1 = (r35<123)
    sub r36 = r36, 32
    st [rb] = r37, 1
    br.ctop loop

Physical register file

Virtual register file

RRB = -5
r34 = u
r35 = l
r36 = B
r37 = _
Machine specific optimizations
Software Pipelining:

- IA-64 features that make this possible
  - Full Predication
  - Special branch handling features
  - Register rotation: removes loop copy overhead
- Traditional architectures use loop unrolling
  - High overhead: extra code for loop body

Especially Useful for Integer Code with Small Number of Loop Iterations
Summary

- IA-64 compilers use existing compiler technologies
  - Inter procedure optimizations to increase the compilation score
  - Profile guided optimizations to make efficient use of predication and speculation
- Compilers take advantage of IA-64 architecture with new compiler technology
  - Predication to remove branches, and increase parallelism
  - Architecture features allow significant speed up in software pipelined loops
# IA-64 Compiler Status

*Early access versions planned for Q1 ‘00*
*Engaging with additional tools vendors in Q3 ‘99*

*Production compilers in sync with Itanium™ system availability*
Call To Action

- IA-64 compiler technology is ready and able. Get started on IA-64 today.

- Review the IA-64 Track software sessions at http://developer.intel.com/design/ia64/devinfo.htm
  - “Getting Software Ready for IA-64”
    - Learn how to get the best performance from your apps
    - Learn what you can do to get started today

- OSV Breakout Sessions
  - Get the latest OS status from the vendors themselves
  - Learn more details on OS tools availability and optimizations