Compiling for the Itanium™ Architecture

Wei Li
Principal Engineer and Manager
Itanium Compiler Development
Intel® Compiler Lab
wei.li@intel.com
Dec 11, 2001
Agenda

• Compiler Overview
• Profiling
• Interprocedural Optimizer
• Disambiguator
• High-level Optimizer
• Parallelizer
• Scalar Optimizer
• Code Generator
• References (recent publications for more details)
Compiler Goals

• Best performance compiler on Intel Architectures
  – Tightly coupled with the latest architectures/micro-architectures
  – Incorporating the best and latest compiler technologies
  – Innovation to push the technology boundaries (patents, conference and journal papers)

• Product compiler
  – Good product stability
  – Reasonable compile-time
Intel Compiler Architecture

C++
Front End

FORTRAN 95
Front End

Profiler

Interprocedural analysis and optimizations: inlining, constant prop, whole program detect, mod/ref, points-to

Loop optimizations: data deps, prefetch, scalar repl, unroll/interchange/fusion/dist, auto-parallel/OpenMP

Disambiguation:
types, array, pointer, structure, directives

Global scalar optimizations: partial redundancy elim, dead store elim, strength reduction, dead code elim

Code generation: predication, software pipelining, global scheduling, register allocation, code generation
Intermediate Languages

- C++ FE
- F95 FE
- Pre-IPO Opts.
- High-level Loop Opts.
- Parallel Opts.
- Scalar Opts.
- IPF CG
IL0 Language

- Almost-syntactic representation
  - Statements, expression trees
- Static single assignment (SSA) use-def
- Some phases use syntactic info, such as
  - If-nesting
  - Goto/label
  - Loops, etc.
- Easily convert to or from syntactic form
Agenda

• Compiler Overview
• **Profiling**
• Interprocedural Optimizer
• Disambiguator
• High-level Optimizer
• Parallelizer
• Scalar Optimizer
• Code Generator
• References
Profile-based Optimizations

- New Itanium™ processor features enable aggressive optimizations and scheduling.

- The highest performance can be achieved with the most accurate profiling information.
  - Characterize the execution behavior of the program
  - Use the profile information to guide optimizations
Using Profile

- Branch probability guide (examples)
  - Predication region
  - Scheduling region
  - Speculation
  - Code placement

- Execution frequency guide (examples)
  - Procedure inlining/partial inlining
  - Loop optimizations
  - Software pipelining
Profile Generation

• Static profiling
  – Program-based heuristics (e.g. loop branch, pointer, call, opcode, loop exit, return, store, loop header, guard, error) for branch probabilities.
  – Estimation of execution frequencies on basic blocks and edges from probabilities and control-flow graph.

• Dynamic profiling
  – Program instrumentation (compile once with counters inserted).
  – Run the instrumented program with a sample input set.
Dynamic Profile Compilation

- `c` -> `-Qprof_gen -c` -> `obj` (instrumented)
- `obj` -> `link` -> `exe` (instrumented executable)
- `exe` -> `merge` -> `dpi` (profile guided optimizations)
- `c` -> `-Qprof_use -c` -> `obj`
Value Profiling

- Strength reduction of “heavy” operations
  - Divides, shifts
- Indirect function calls
  - Pull out n most frequent call targets
  - Specialized for virtual functions
    - Test for pointer to vtbl eliminates indirection
- Being extended to
  - Loop trip counts
  - Memset/memcpy length and alignment
**Agenda**

- Compiler Overview
- Profiling
- **Interprocedural Optimizer**
- Disambiguator
- High-level Optimizer
- Parallelizer
- Scalar Optimizer
- Code Generator
- References
Interprocedure Optimizer Overview

• Inlining
  – Increase scope of optimization and instruction scheduling

• Constant propagation

• Whole program detection
  – Understand the behavior of every function

• Points-to analysis
  – Better disambiguation of pointer-based code

• Mod/ref Analysis
  – Fewer kills due to modification (PRE) or reference (PDSE)

• Procedure placement
  – Cache locality
IP/IPO Pass1

C/C++ FE ➔ F90

Module ➔ IL1/IL0 Interface/Glue ➔ PGOPTI ➔ IP Summary ➔ ILIO ➔ .il

Routine ➔ Module ➔ Routine ➔ Summary
IP/IPO Pass2
Whole Program Detection

- Determines whether all procedures have known behavior
  - Procedures for which IL is present
  - Procedures known to library database. e.g. no side effect in sin, cos
- Enables variety of optimizations
  - Points-to analysis
  - Global address taken analysis: indirect vs. global variable whose address not taken
  - Malloc combining, etc.
Points-to Analysis

- Compile-time estimation of potential pointer targets
- Enable accurate memory disambiguation

\[
\begin{align*}
S: & \quad a = \& x; \quad \text{(a, x)} \\
T: & \quad b = \text{malloc}(); \quad \text{(b, allocT)} \\
U: & \quad c = a; \quad \text{(c, x)} \\
V: & \quad *a = 5; \\
W: & \quad y = b->i;
\end{align*}
\]

- Statements V and W do not conflict
Whole Program Points-to Analysis

• Based on Andersen’s approach

• Off-line substitution to speed up analysis
  – [RountevChandra:PLDI 2000]

• Recently incorporated the ideas from “Ultra-fast Aliasing Analysis Using CLA”
  – [HeintzeTardieu: PLDI 2001]

• Analysis time:
  – 20 seconds max for any benchmark
  – 600Mhz Pentiumlll
Enhanced Features

• Distinguishes between fields of a structure type
  – a->f and a->g are considered accesses to fields f and g

• Does not distinguish between instances of a given structure type
  – a->f and b->f are considered accesses to the same location f
  – Exception: Allocating or taking address of entire structure

• Distinguishing fields instead of instances is a big win

• Identification of simple user-defined memory allocators
  – SSA-based analysis, prove allocator is malloc-like
Function Placement

**SOURCE:**
```c
void func1()
{
    ...
    func3();
    ...
}
void func2()
{
    // really big function
    // not called frequently
}
void func3()
{
}
```

**PAGE LAYOUT:**
- **without**
  - func1
  - func2
  - func3
- **with**
  - func1
  - func3
  - page boundary
  - func2
Agenda

- Compiler overview
- Profiling
- Interprocedural optimizer
- Disambiguator
- High-level Optimizer
- Parallelizer
- Scalar optimizer
- Code Generator
- References
Disambiguator

• A single, unified framework containing many methods
• Intraprocedural methods
  – Direct memory references
  – Indirect references without points-to
  – Simple base + offset analysis
  – Local points-to analysis
  – Array data dependence analysis
• Interprocedural methods
  – Global address taken analysis
  – Whole program points-to analysis
• Methods requiring user assertion
  – Type-based disambiguation
Intraprocedural Methods

• Direct memory references
  – Different objects
  – Different fields of same structure

• Indirect references without points-to
  – What cannot be pointed-to
  – Local variable whose address not taken
  – Dereference of unmodified parameter vs. stack local

• Simple base + offset analysis
  – Analyze address expressions for base + offset
  – If same base, compare offsets
  – Different fields from same base pointer
  – Array accesses with constant subscripts
  – Unstructured addressing
Intraprocedural Methods Continued

• Local points-to analysis (lpt)
  – Intraprocedural version of whole program points-to

• Array data dependence analysis (array)
  – Distinguish different elements of same array
  – Statically or dynamically allocated arrays
Agenda

- Compiler overview
- Profiling
- Interprocedural optimizer
- Disambiguator
- High-level Optimizer
- Parallelizer
- Scalar optimizer
- Code Generator
- References
High-level Optimizer Overview

- Data dependence analysis
  - Array subscript analysis
- Eliminating memory operations
  - Scalar replacement, unroll-and-jam, register blocking
- Cache optimizations
  - Loop interchange, fusion, distribution, blocking, data layout
- Overlapping memory latency
  - Data prefetch
- Instruction-level parallelism
  - Unrolling, interchange, distribution
- Bandwidth optimization
  - Load-pair, array contraction
## HLO Architecture

**HLO Driver:** phase ordering, global cost analysis.

### Optimizations (cost model and algorithms):
loop nest optimization for cache locality, cross-loop optimization for cache locality, register blocking, load and store elimination, load-pair, data prefetch, data padding for cache conflict, unrolling for ILP, balancing computation and memory, memory binding for bandwidth, cache hints for load/store, cache hints for swp, data transformations, array contraction, memory directives, reduction, transform for ILP, transform for SWP, transform for vectorization, transform for multiprocessor.

### Analysis:
data dependence analysis, reuse analysis, array section analysis, live range analysis, induction variable analysis, register pressure, swp schedule.

### Basic Transformations:
linear transformations (interchange, skewing, reversal, scaling), fusion, distribution, unrolling, blocking, unroll-and-jam, stripmining, collapsing, peeling, scalar replacement, data padding, array and structure remapping, prefetch.

### HLO IR (data structures, loop recovery, linear expressions)
Scalar Replacement with Rotating Registers

- Eliminate loads and stores for array references by exploiting data reuse carried by loops.

```plaintext
for i = 6, 100
end_for

Init t1, t2, t3, t4, t5
for i = 6, 100
    t0 = t5 + B[i]
    a[i] = t0
    t5 = t4; t4 = t3;
    t3 = t2; t2 = t1;
    t1 = t0
end_for
```

No explicit moves

Explicit moves required

```
.b1:
    (p16) ld4          r32 = [r3], 4
    (p18) add         r35 = r40,  r34
    (p19) st4          [r2] = r36, 4
    br.ctop .b1
```
Cache Optimizations

• Reduce reuse distance with techniques like blocking, linear transformations, fusion, distribution, data layout optimizations.
• Multi-level memory hierarchy hides memory latency, when good data reuse.

```
for i = 1, 1000  
  for j = 1, 1000  
    for k = 1, 1000  
      end_for  
    end_for  
  end_for
```
Selective Prefetch with Predication

• Selective prefetching:
  – If an access has spatial locality, only the first access to the same cache line will incur a miss - the condition is \((i \mod \text{line\_size}) = 0\)
  – For temporal locality, only the first access will incur a cache miss - the condition is \(i = 0\)
  – If an access has group locality and is not the leading reference, there is no cache miss
  – If an access has no locality, it will miss in every iteration

• On traditional architectures, having an if-condition in the innermost loop is too expensive

• If-statements converted into predicates
Using Rotating Address Registers

for \( j = 1, n \)
\[
\begin{align*}
  r_{43} &= \text{address of } D(2+k,j) \\
  r_{44} &= \text{address of } C(1+k,j)
\end{align*}
\]
for \( i = 1, m \)
\[
\begin{align*}
  \text{prefetch } [r_{44}] \\
  r_{42} &= r_{44} + 16 \\
  r_{44} &= r_{43} \\
  r_{43} &= r_{42} \\
  C(i,j) &= D(i-1,j) + D(i+1,j)
\end{align*}
\]
end_for
end_for

Uses just one prefetch instruction for multiple arrays. Save memory slots. No predicate computation.
Prefetch Hints

- 4 types of hints - T0, NT1, NT2, and NTA
- T0 - temporal locality at level 1
- NT1/NT2 - no temporal locality at level 1/2
- NTA - no temporal locality at all levels
- Can take advantage of these hints along with hints for loads and stores to orchestrate data movement across the cache hierarchy
Aggregating Loads: Load-pairs

- Loop unrolling for memory aligned paired loads. Data alignment and loop remainder issues.
- Result: high bandwidth and reduced demand for memory issue slots.

```
for i = 1, 1000
    y[i] = y[i] + a * x[i]
end_for
```

```
for i = 1, 1000, 2
    t1, t2 = ldfpd(x[i], x[i+1])
    t3, t4 = ldfpd(y[i], y[i+1])
    y[i] = t3 + a * t1
    y[i+1] = t4 + a * t2
end_for
```
Bandwidth Optimizations

• Reducing memory footprint with array contraction. Requires complex compiler analysis.
• Result: reduced memory bus traffic.

```
for j = 2, N-1
    for i = 2, N-1
        E[i, j] = A[i-1, j] + B[i+1, j]
    end_for
end_for
for I = 2, N-1
    D[I, 2] = 1/E[I, 2]
end_for
for v = 3, N-1
    for u = 2, N-1
        r = D[u, v-1]
        D[u, v] = 1/(E[u, v]+r*C[u, v-1])
    end_for
end_for
```

Benefits: eliminated an array to reduce bandwidth requirement.
Agenda

- Compiler overview
- Profiling
- Interprocedural optimizer
- Disambiguator
- High-level Optimizer
- **Parallelizer**
- Scalar optimizer
- Code Generator
- References
Parallelizer Overview

- OpenMP directives
- Auto-parallelization
- Multi-threading code generation
  - Calling runtime routine to fork threads
  - Runtime library provided by KAI
- Support for Assure, a tool to help debug OpenMP/threaded programs
VOID foo()
{
    int A[1000], B[1000], C[1000], X[1000], I, NUM;
    /* parallel region */
#pragma omp parallel private(NUM) shared(X, A, B, C)
    {
        NUM = omp_get_num_threads();
        #pragma omp for private(I)  /* work-sharing for loop */
            for (I = 0; I< 1000; I++)
            {
                /* assume that the routine “bar” has no side effects */
                X[I] = bar(A[I], B[I], C[I], NUM);
            }
    }
}
MT-Code Generation Example

Parallelizer Generate “FORK” call to enable runtime library to create threads entering threaded entry generated by parallelizer

Threaded Entry: __par_region, data set: A, B, C, X
runtime Initialization calls: get LB, UB, ST
NUM = omp_get_num_threads()
for (I=LB, I<=UB, I++) { X[I] = bar(A[I], B[I], C[I], NUM); }
Runtime Synchronization calls

Thread-0  Thread-1  Thread-2  Thread-3
Automatic Parallelization

- MT-Code generation and privatization modules shared between OpenMP and auto-parallelization.
- PAROPT analyzer finds scalars where every use is dominated by a def in the loop – candidate for privatization
  - E.g. scalar x is to be made private
    
    ```c
    for (i=0; i<n; i++) {
        x = a(i) + bb(i);
        ...
        c(i) = x + dd(i);
    }
    ```
- PAROPT analyzer finds loop-carried-dependence-free loops and marks them as parallel loops.
Advanced MT-Code Generation

• Runtime Dependence Test based MT-Code
• Multi-Versioning
• Profile-Guided MT-Code Generation
• Threshold-Controlling
Agenda

- Compiler overview
- Profiling
- Interprocedural optimizer
- Disambiguator
- High-level Optimizer
- Parallelizer
- Scalar optimizer
- Code Generator
- References
Scalar Optimizer Overview

• SSA-based optimizations
  – PRE
  – Strength reduction
  – Sign/zero extend optimization
  – Copy propagation
  – Constant propagation
  – Dead code elimination

• Redundant conditional elimination

• Instruction recognition
  – FMA
SSA PRE

- Combined loop invariant, CSE
- Memory-to-register conversion
- Extended with hoisting
  - Reduce code size
- Profile-based speculative PRE
Partial Redundancy Elimination

- A computation is *partially redundant* if its result is available on *some* paths.
- Remove partial redundancy by hoisting the computation to the not available path.
Anticipability and Safety

- A computation is *anticipated* at a statement $S$ if it occurs on *all* the paths from $S$ to the program exit.
- Hoisting anticipated computation is safe and won’t increase any path length.
Control Speculation

• The load is only *partially anticipated* at Loop-entry.
• Speculatively hoist the load and propagate the speculative value. Compensation after kill.
Combining Data/Control Speculation

- Control and data speculation are applied at the same time.
Strength Reduction

- Important to exploits features of architecture
  - Strength reduction and substitution extremely important for memory addressing with only register indirect addressing
  - Strength reduction important to avoid integer multiplies
- SSA based algorithm allows easier to maintain design
- Register pressure
- Does sign/zero extension removal for induction variables
- Vital to expose opportunities for post-increments
Sign/zero Extend Optimization

- Important for 32-bit applications
- Especially important for pointer-intensive applications
- Enables better scheduling of memory operations
- Enables better use of post-increment instructions
- Precise, bitwise model of each IL0 operator
- Forward dataflow
  - Infers properties of producers
- Backward dataflow
  - Infers properties of consumers
- Used to remove sign/zero extension
Agenda

• Compiler overview
• Profiling
• Interprocedural optimizer
• Disambiguator
• High-level Optimizer
• Parallelizer
• Scalar optimizer
• Code Generator
• References
Code Generator Overview

• Predication
  – If-conversion to use regular, unconditional and parallel compares

• Software pipelining
  – Uses rotating registers, stage predicates, loop branches

• Global scheduling
  – Instruction scheduling across basic block boundaries
  – Uses control and data speculation, predication, post-increments, multiway branches

• Global register allocation
  – Allocate registers for predicated code
  – Region-based

• Driven by information provided by machine model
Predication

- When branch misprediction rate is high, it is better to predicate
- Predication creates more ILP
- Predication has the potential cost of increasing the critical path length
- Reduce code size by eliminate branches
- Better code packing in bundles
- Techniques
  - If-conversion
  - Parallel compare to reduce control height
If-conversion for Predication

- Identifying region of basic blocks based on resource requirement and profitability (branch misprediction rate, misprediction cost, and parallelism)
- Result: a single predicated basic block

\[
\begin{align*}
  a &< b \\
  s &= s + a \\
  s &= s + b \\
  *p &= s \\
  \text{cmp.lt p1,p2} &= a,b \\
  (p1) s &= s + a \\
  (p2) s &= s + b \\
  *p &= s
\end{align*}
\]
Software Pipelining

- Exploit parallelism across iterations without code bloat
- Architectural features used:
  - Rotating registers
  - Rotating predicates (stage predicates)
  - Predication to collapse control flow
  - Counted loop
  - While loop
  - Control and data speculation
Overlapping Loop Iterations

- Exploit parallelism across loop iterations.
- Result: kernel-only code without code expansion.

L1: ld4 r4  = [r5], 4;  // 0
    add r7  = r4, r9;  // 2
    st4 [r6] = r7, 4    // 3
    br.cloop L1;;

L1:
(p16) ld r32  = [r5], 4  // Cycle 0
(p18) add r35  = r34, r9 // Cycle 0
(p19) st4 [r6] = r36, 4  // Cycle 0
    br.ctop L1;;  // Cycle 0
Modulo Scheduling

- Find the minimum II (initiation interval) with a valid modulo schedule
  - II – constant interval between the start of successive iterations
  - Resource II -- from machine model’s estimation
  - Recurrence II -- from critical dependence cycles
- Scheduling from MinII
  - Determine instruction scheduling priority
  - Use data dependence constraints to find a legal range
  - Place in the earliest clock cycle satisfying resource constraints
  - Sometimes schedule latest for minimizing register pressure
  - Unschedule some instructions to make room if the above attempt is not successful
  - Increase II if unscheduled instructions over budget
Global Code Scheduling

- Acyclic subgraph with, single/multiple entry/exit containing
  - Basic block nodes
  - Nested nodes from lower levels in hierarchy
    - Loops, or already scheduled regions
    - Contain data flow summary information
- Formation and scheduling of regions tightly coupled
- Uses architectural features
  - Large number of registers
  - Control and data speculation, checks and recovery code
  - Multiway branches
Wavefront Scheduling

• Wavefront is a set of acyclic scheduling region nodes that
  – Represents scheduling boundary
    • Nodes above wavefront (already scheduled)
    • Nodes on wavefront (being scheduled)
    • Nodes below wavefront (not yet scheduled)
  – Is a strongly independent cut set
    • Every path in scheduling region flows through exactly one node on the wavefront
  – Relaxes the constraint of scheduling one block at a time

• Wavefront advancement can be constrained so compensation satisfies SCP (single copy on a path)
Wavefront Scheduling (Example)
Global Register Allocation

• Incorporating architectural features
  – Large number of registers
  – Predication
  – Data speculation
• Region-based allocation
• Graph-coloring (Chaitin) with optimistic coloring (Briggs)
• Register coalescing
• Simple Rematerialization
• ALAT management
• Predicate-aware allocator
Minimize ALAT Conflict

- Assign registers to the live ranges of the advanced loads to eliminate possible conflict in ALAT (Advanced Load Address Table)
Predicate-aware Register Allocation

- Use the same register for two separate computations in the presence of predication, even if there is live-range overlap without considering predicates.
Modeling The Machine

• Objective: Provide micro-architecture information to the rest of the compiler
  – Machine characteristics read from microarchitecture description file, which is used by simulator and other tools.
Agenda

- Compiler overview
- Profiling
- Interprocedural optimizer
- Disambiguator
- High-level Optimizer
- Parallelizer
- Scalar optimizer
- Code Generator
- References
Recent Publications from the group

• “An overview of the Intel IA-64 compiler”, Intel Tech Journal, Q4’99
• “An advanced optimizer for IA-64”, IEEE Micro, no6’00
• “The IA-64 Electron code generator”, IEEE Micro, no5’00
• “On the importance of points-to analysis and other memory disambiguation methods for C programs”, PLDI’01
• “Optimizing software data prefetches with rotating registers”, PACT’01
• “Wavefront Scheduling: path based data representation and scheduling of subgraphs”, MICRO’99
• “Software pipelining of loops with early exits for the Itanium architecture”, EPIC1 workshop, MICRO’01
• Itanium™ Compiler tutorials at ISCA’00, Hotchips’99, PACT’99
Summary

• Best performance compiler on Intel Architectures
  – Tightly coupled with the latest micro-architectures.
  – Incorporating the best and latest compiler technologies
  – Innovation to push the technology boundaries

• Product compiler used by ISV’s
  – Good product stability
  – Reasonable compile-time