Compiler Technology on IA-64

Part I:	IA-64 Compiler Overview	(Wei Li)
Part II:	Compiler Optimizations	(Wei Li)
Part III:	Backend Technologies	(Jesse Fang)
Part IV:	Dynamic Optimizations	(Jesse Fang)

Part I: IA-64 Compiler Overview

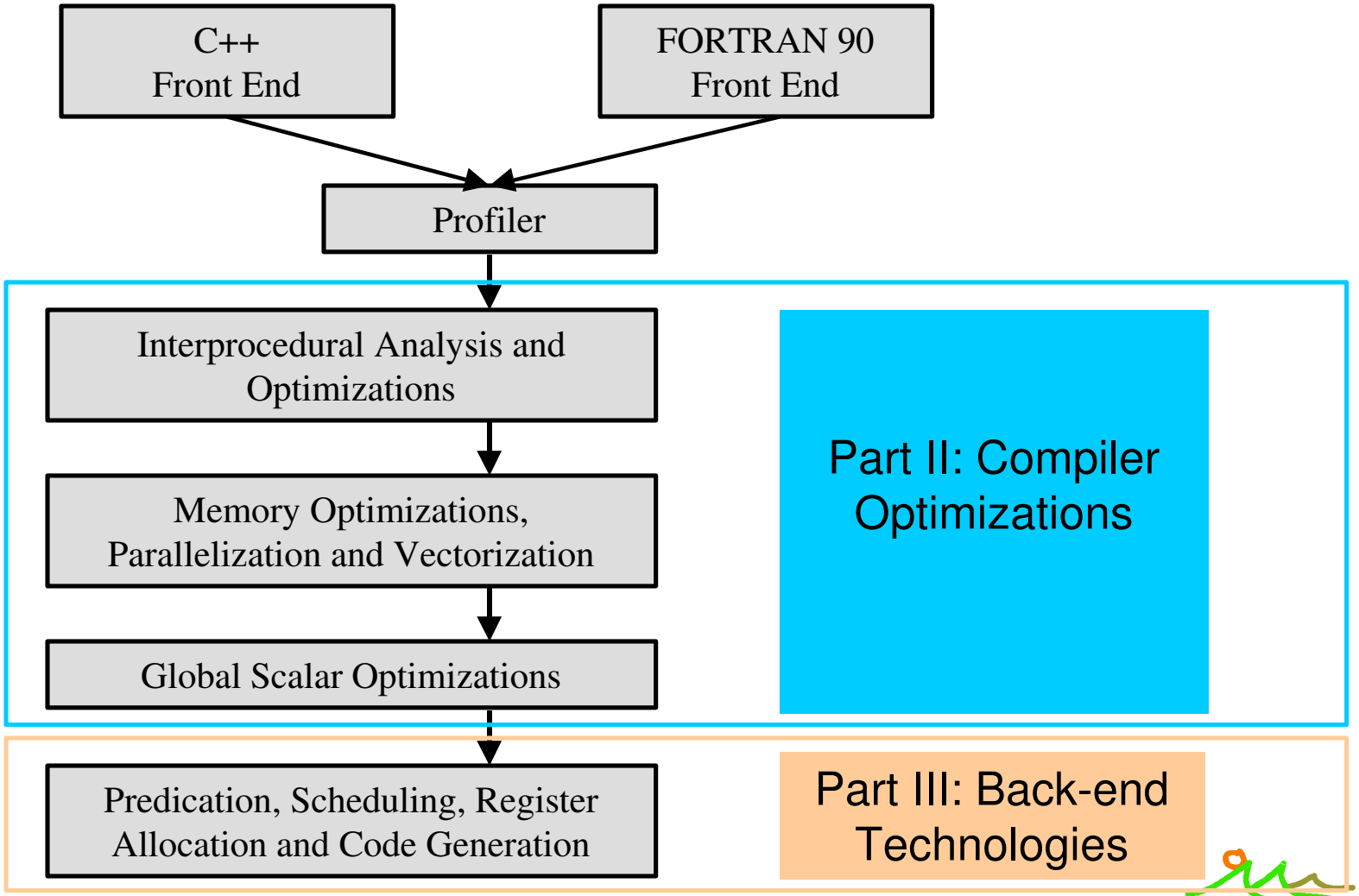
Wei Li

Microcomputer Software Labs

Overview

- **Compiler Architecture**
- **Profiling**

IA-64 Compiler Architecture



Profile-based Optimizations

- **New IA-64 features enable aggressive optimizations and scheduling based on profiling information.**
- **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**

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 from probabilities and control-flow graph.
- Dynamic profiling
  - Program instrumentation (compile once with counters inserted).
  - Run the instrumented program with a sample input set.

Using Profile

- Profile used in many parts of the IA-64 compiler
- Branch probability guide
  - Predication region
  - Scheduling region
  - Speculation
  - Code placement
- Execution frequency guide
  - Procedure inlining/partial inlining
  - Loop optimizations
  - Software pipelining
  - Register allocation

Part II: Compiler Optimizations on IA-64

Rakesh Krishnaiyer

Dattraya Kulkarni

Wei Li

John Ng

David Sehr

Peng Tu (Tensilica)

Microcomputer Software Labs

Overview

- **A partial list of optimizations that can make use of the IA-64 features**
  - **Procedure Inlining**
  - **Interprocedural Analysis**
  - **Code Placement**
  - **Data Dependence Analysis for Speculation**
  - **Eliminating Memory Operations**
  - **Cache Optimizations**
  - **Overlapping Memory Latency**
  - **Parallelization and Vectorization**
  - **Loop Unrolling for ILP**
  - **Partial Redundancy Elimination**
- **and more ……**

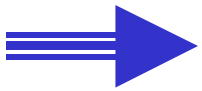
Procedure Inlining

- Benefits:
  - Interprocedural information for a specific call site.
  - Larger code region for optimizations (such as loop transformations)
  - Larger code region to schedule
- Cost:
  - Code size increases
- Selective integration (partial inlining and cloning) to reduce code size growth.

Inlining

- Inlining a function body into a call site.

```
void func1()
{
    int i;
    for (i=0;…)
        func2(i);
}
void func2(int x)
{
    a[x] = 1.0;
}
```



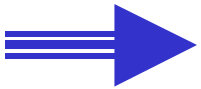
```
void func1()
{
    int i;
    for (i=0;…)
        a[i] = 1.0
}
```

Partial Inlining

- Copy the *hot* portion of a function into a call site.
- Remainder becomes a *splinter* function.

```
void foo(x)
{
 if (P(x))
   a hot region here
 else
   a large cold region here.
}

void main()
{
 call foo(y)
}
```



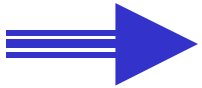
```
void foo_cold(x)
{
 a large cold region here.
}

void main()
{
 if (P(y))
   a hot region here
 else
   call foo_cold(y)
}
```

Cloning

- Specializing a function to a specific class of call sites.

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



```
void func1()
{
    func2_0(n);
    func2_0(m);
    func2(i, m);
}

void func2_0(int j)
{
    // do something
}
```

Interprocedural Analysis and Optimization

- Use of Alias Analysis
  - Indirect call conversion
  - Better disambiguation
- Use of Mod/ref Analysis
  - Fewer *kills* due to unknown modification (PRE) or reference (PDSE)
- Constant propagation
  - Makes constant parameters and globals explicit.
  - Enables cloning.
  - Remove unnecessary conditional code.
  - Improves data dependence analysis.

Code Placement

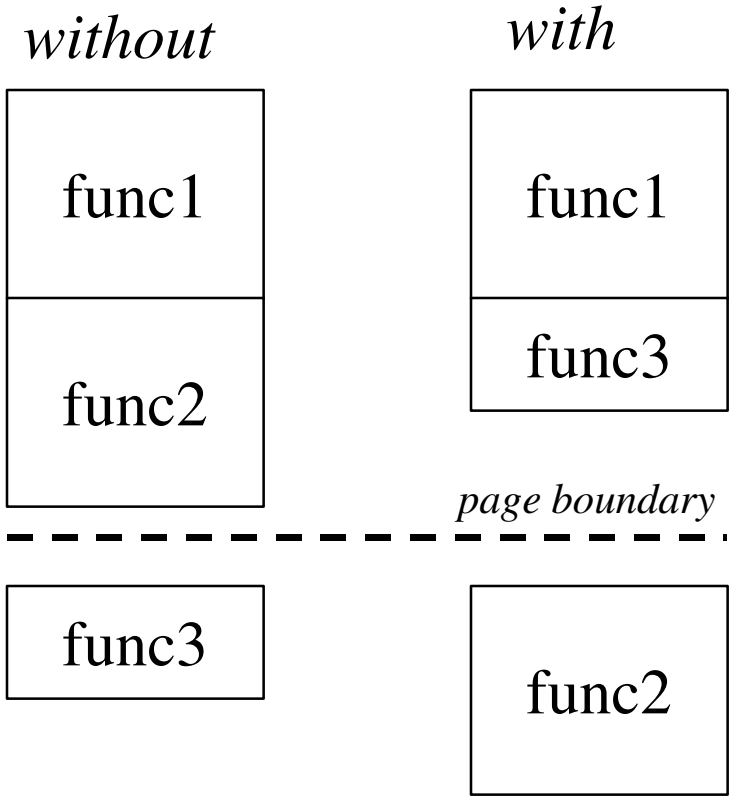
- Instruction locality optimizations
- Block ordering
  - Lays out blocks to take best advantage of branch heuristics.
  - Groups *hot* blocks together
  - Places cold blocks at function end
- Function placement
  - Places functions near callers and callees.
- Function splitting
  - Moves *cold* portions of all functions into a special cold segment

Function Placement

SOURCE:

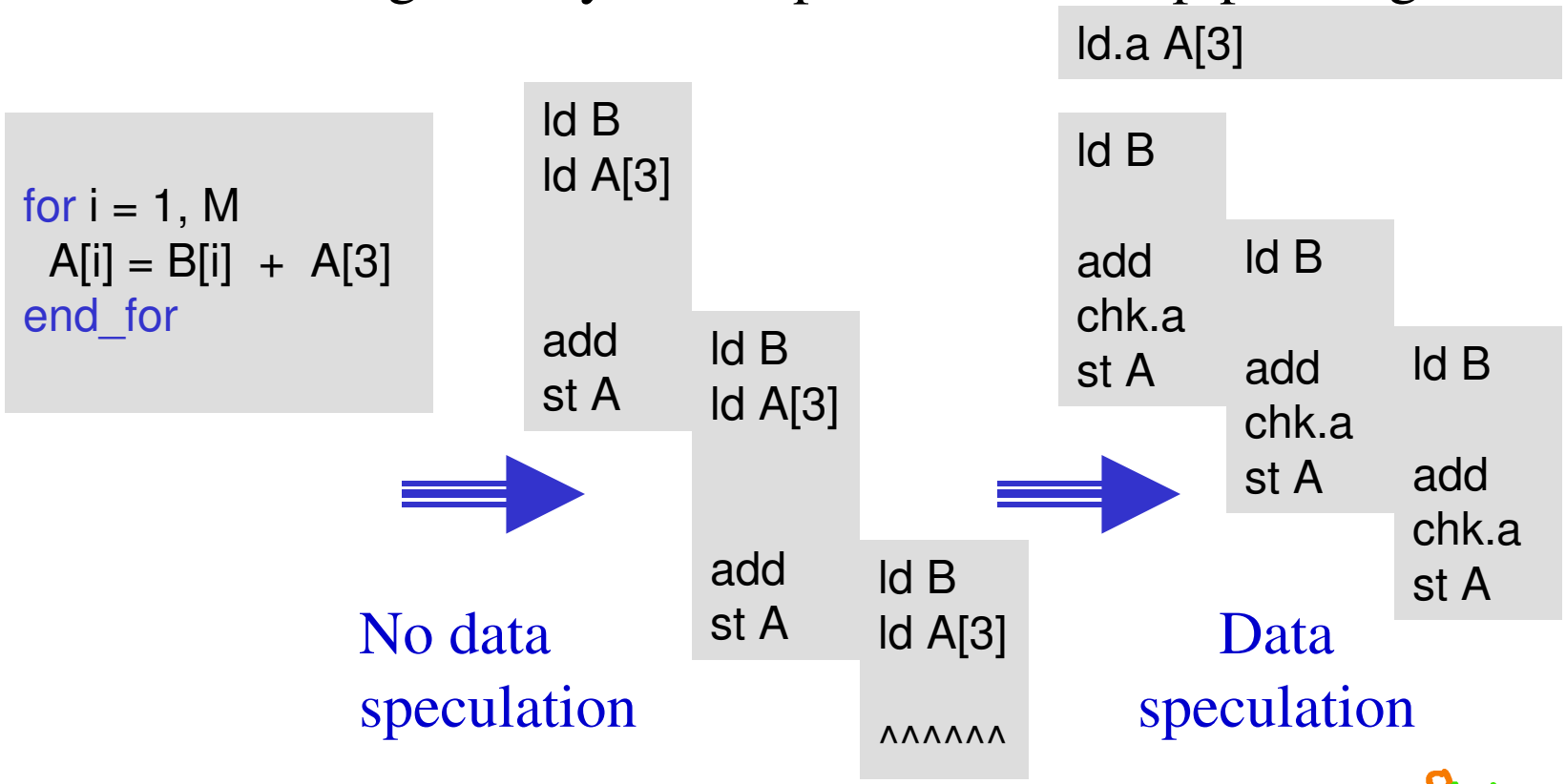
```
void func1()
{
        ...
        func3();
        ...
}
void func2()
{
        // really big function
        // not called frequently
}
void func3()
{
}
```

PAGE LAYOUT:



Data Dependence Analysis for Speculation

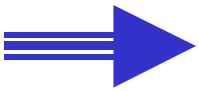
- Hoist loads above possibly conflicting stores. Applied with *probabilistic data dependence analysis*.
- Result: hiding latency and improve software pipelining.



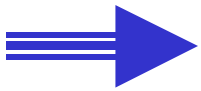
Computing Probability

- Compute the number of occurrences of the dependence.
- Estimate the number of dynamic pairs of ld/st or st/st., e.g. the size of the iteration space.

```
for i = 1, M
  A[i] = B[i]  +  A[3]
end_for
```



The number of solutions for A[i]=A[3] is 1.



The number of pairs (A[i], A[3]) is M.

The dependence probability is 1/M.

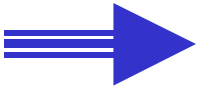
Eliminating Memory Operations

- Problem:
  - Increasingly large gap between processor and memory speed.
  - Register allocation
    - only handles scalars, not array references.
    - doesn't exploit data reuse carried by loops.
- Solutions
  - Register Blocking
  - Load and Store Elimination
- IA-64 advantage: large register set

Register Blocking

• Turn loop carried data reuse into loop independent data reuse, usually in the same basic block.

```
for j = 1, 2*M
  for i = 1, 2*N
    A[i, j] = A[i-1, j]  +  A[i-1, j-1]
  end_for
end_for
```

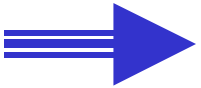


```
for j = 1, 2*M, 2
  for i = 1, 2*N, 2
    A[i, j] = A[i-1,j] + A[i-1,j-1]
    A[i, j+1] = A[i-1,j+1] + A[i-1,j]
    A[i+1, j] = A[i, j] + A[i, j-1]
    A[i+1, j+1] = A[i, j+1] + A[i, j]
  end_for
end_for
```

Virtual Register Allocation

- Turn array references into scalars.
- Result: memory operations reduced to register load/store.

```
for j = 1, 2*M
  for i = 1, 2*N
    A[i, j] = A[i-1, j]  +  A[i-1, j-1]
  end_for
end_for
```



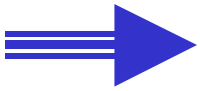
```
for j = 1, 2*M, 2
  for i = 1, 2*N, 2
    r1 = A[i-1,j]
    r2 = r1 + A[i-1,j-1]
    A[i, j] = r2
    r3 = A[i-1,j+1] + r1
    A[i, j+1] = r3
    A[i+1, j] = r2 + A[i, j-1]
    A[i+1, j+1] = r3 + r2
  end_for
end_for
```

Large register file allows more aggressive blocking. E.g. from 8MN to 4MN loads.

Load and Store Elimination

- Eliminate loads and stores for array references by exploiting data reuse carried by loops, a.k.a *scalar replacement.*
- Result: more memory operations reduced into register load/store.

```
for i = 2, N+1
        = A[i-1] + 1
   A[i] =
end_for
```



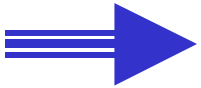
```
t1 = A[1]
for i = 2, N+1
      = t1 + 1
  t1 =
  A[i] = t1
end_for
```

Benefits: reduced 2N to N memory operations.

Unroll-and-Jam

- Turn outer-loop carried data reuse into inner-loop data reuse.
- Result: expose more opportunity for scalar replacement.

```
for j = 1, 2*M
  for i = 1, N
    A[i, j] = A[i-1, j]  +  A[i-1, j-1]
  end_for
end_for
```



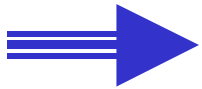
```
for j = 1, 2*M, 2
  for i = 1, N
    A[i, j] = A[i-1,j] + A[i-1,j-1]

    A[i, j+1] = A[i-1,j+1] + A[i-1,j]
  end_for
end_for
```

More Load and Store Elimination

• *Scalar replacement* may introduce extra copying operations to cover the reuse distance.

```
for j = 1, 2*M, 2
  for i = 1, N
    A[i, j] = A[i-1,j] + A[i-1,j-1]
    A[i, j+1] = A[i-1,j+1] + A[i-1,j]
  end_for
end_for
```



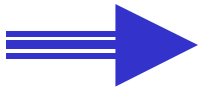
```
for j = 1, 2*M, 2
   t1 = A[0, j]
   t2 = A[0, j+1]
  for i = 1, N
    t0 = t1 + A[i-1, j-1]
    A[i, j] = t0
    t2 = t3 + t1
    A[i, j+1] = t2
    t3 = t2
    t1 = t0
  end_for
end_for
```

Benefits: reduced 3MN to MN loads, but introduced extra moves.

Rotating Registers

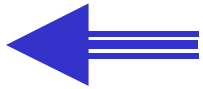
- Automatic register renaming
- Result: anti-dependence broken.

```
for i = 6, 100
  A[i] = A[i-5] + B[i]
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
```

Explicit moves required



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

No explicit moves

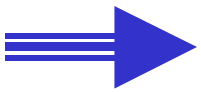
Cache Optimizations

- Goal: hit the closest cache as much as possible.
- Reduce cache misses
  - Capacity misses
  - Conflict misses
  - Cold misses
- Solutions
  - Locality Optimization
  - Explicit Memory Hierarchy Control

Cache Locality 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
      A[i, j, k] = A[i, j, k]  +  B[i, k, j]
    end_for
  end_for
end_for
```

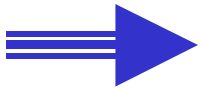


```
for v = 1, 1000, 20
  for u = 1, 1000, 20
    for k = v, v+20
      for j = u, u+20
        for i = 1, 1000
          A[i, j, k] = A[i, j, k]
                       +  B[i, k, j]
        end_for
      end_for
    end_for
  end_for
end_for
```

Cache Hints for Streaming Data

- Placement of data at desired level of the memory hierarchy, using compile-time *data reuse analysis*.
- Result: avoid cache pollution with streaming data.

```
for i = 1, 1000
  for j = 1, 1000
    C[i] = C[i] + A[i, j]  *  B[j]
  end_for
end_for
```



```
for j = 1, 1000
  for i = 1, 1000
    t = A[i,j].nta
    C[i] =  C[i] + t * B[j]
  end_for
end_for
```

Allows high cache performance for multimedia applications.

Overview: what is covered so far

- *Procedure Inlining*
- *Interprocedural Analysis*
- *Code Placement*
- *Data Dependence Analysis for Speculation*
- *Eliminating Memory Operations*
- *Cache Optimizations*
- **Overlapping Memory Latency**
- **Parallelization and Vectorization**
- **Loop Unrolling for ILP**
- **Partial Redundancy Elimination**

Overlapping Memory Latency

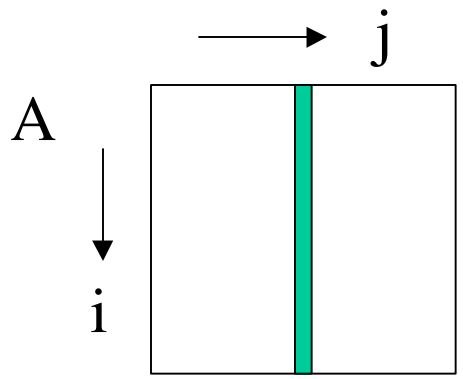
- If the latency is long, start early.
- Use computation to overlap latency.
- Techniques
  - Data Prefetching and Predication
  - Control Speculation
  - Data Speculation

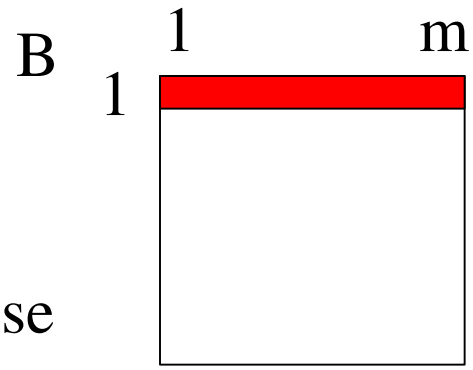
Prefetching Guidelines

- Avoid unnecessary prefetches - prefetch only refs that are predicted to suffer cache misses
- Avoid ineffective prefetches - don’t prefetch too early or too late
- Extra instructions - prefetch instructions + instructions that generate the addresses
- Takes up memory slots - may increase II for a software-pipelined loop
- May cause more stress on the memory and cache subsystems

Reuse Analysis

```
for j = 1, n
  for i = 1, m
    A[i, j] = B[1, i] +  B[1, i+1]
  end_for
end_for
```





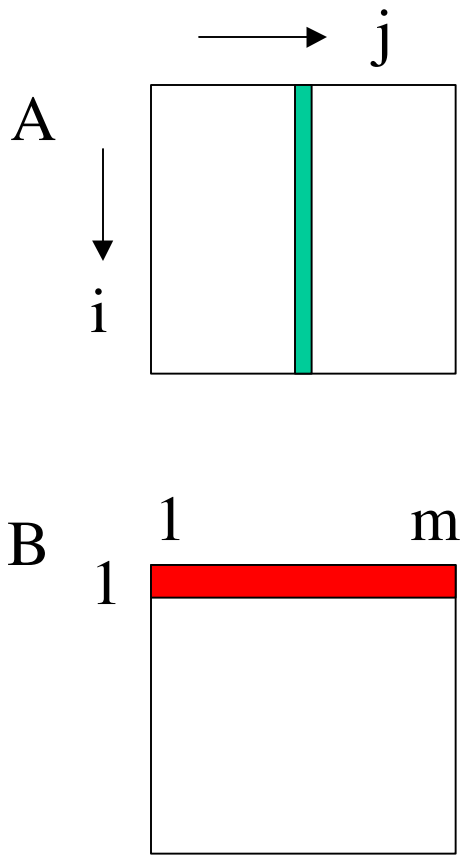
A(i,j) has spatial reuse
B(1,i+1) has temporal reuse
B(1,i) and B(1,i+1) have group reuse

Prefetch Predicates

- If an access has spatial locality, only the first access to the same cache line will incur a miss - predicate is (i mod line_size) = 0
- For temporal locality, only the first access will incur a cache miss - predicate 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

Example Code with Prefetches

```
for j = 1, n
  for i = 1, m
    A[i, j] = B[1, i] +  B[1, i+1]
    if (iand(i,7) == 0)
      prefetch (A(i+k,j))
    if (j == 1)
      prefetch (B(1,i+t))
  end_for
end_for
```



Assumed CLS = 8 words.

k and t are prefetch distance values

Predication vs. Unroll

- In traditional architectures, having an if-condition in the innermost loop is too expensive
- Loop splitting is performed to remove the conditionals - unrolling, stripmining, or peeling; Problem: code expansion
- In IA-64, predication can help
- If-statements converted into predicates

Assembly Code with Predication

```
.b1_2:
 (p17)    ldfd          f32=[r8],8
 (p20)    fma.d         f38=f37,f1,f35
 (p17)    and           r38=7,r35
 (p26)    lfetch        [r37]
 (p17)    add           r34=1,r35 ;;
 (p17)    cmp4.eq.unc p23,p24=r38,r0
 (p17)    add           r32=8,r33
 (p17)    cmp.le        p16,p0=r34,r3 ;;
 (p22)    stfd          [r2]=f40,8
 (p17)    add           r35=8,r36
 (p24)    cmp4.eq.unc  p25,p0=1,r38
 (p23)    lfetch        [r33]
 (p16)    br.wtop.dptk .b1_2 ;;
```

```
for j = 1, n
  for i = 1, m
    C[i, j] = D[i-1, j] +  D[i+1, j]
    if (iand(i,7) == 0)
      prefetch (C(i+k,j))
    if (iand(i,7) == 1)
      prefetch (D(i+k+1,j))
  end_for
end_for
```

Prefetch Hints

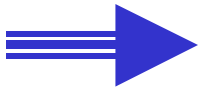
- 4 types of hints - T0, NT1, NT2, and NTA
- T0 - temporal locality at level 1
- NT1/NT2 - no temporal locality at level 1/2
- NTA - no temporal locality at all levels
- Can take advantage of these hints along with hints for loads and stores to orchestrate data movement across the cache hierarchy

Data Prefetching

• Adding prefetching instructions using *selective prefetching*. Reuse analysis for the best level in the hierarchy.

• Result: data is in cache, when used. Eliminated redundant prefetches.

```
for i = 1, M
  for j = 1, N
    A[j, i] = B[0, j]  +  B[0, j+1]
  end_for
end_for
```



```
for i = 1, M
  for j = 1, N
    A[j, i] = B[0, j]  +  B[0, j+1]
    if (mod(j,8) == 0)
      lfetch.nt1(A[j+d, i])
    if (i == 1)
      lfetch.nt1(B[0, j+d])
  end_for
end_for
```

Predication eliminates the need for extensive loop unrolling.
Cache hints.

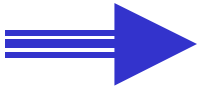
Parallelization and Vectorization

- High architecture bandwidth
- Vectorization
- Techniques
  - Transformation for Parallel Instructions
  - Transformation for Load Pairs
  - Transformation for Parallelism/SWP

Transformation for Parallel Instructions

- Vector operations. Traditional vectorization techniques apply here.
- Result: multiple elements are processed in parallel.

```
for j = 1, 1000
  z[j] = x[j] + y[j]
end_for
```

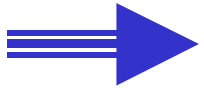


```
for j = 1, 1000, 2
  z[j:j+1] = x[j:j+1] + y[j:j+1]
end_for
```

Transformation for 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
```

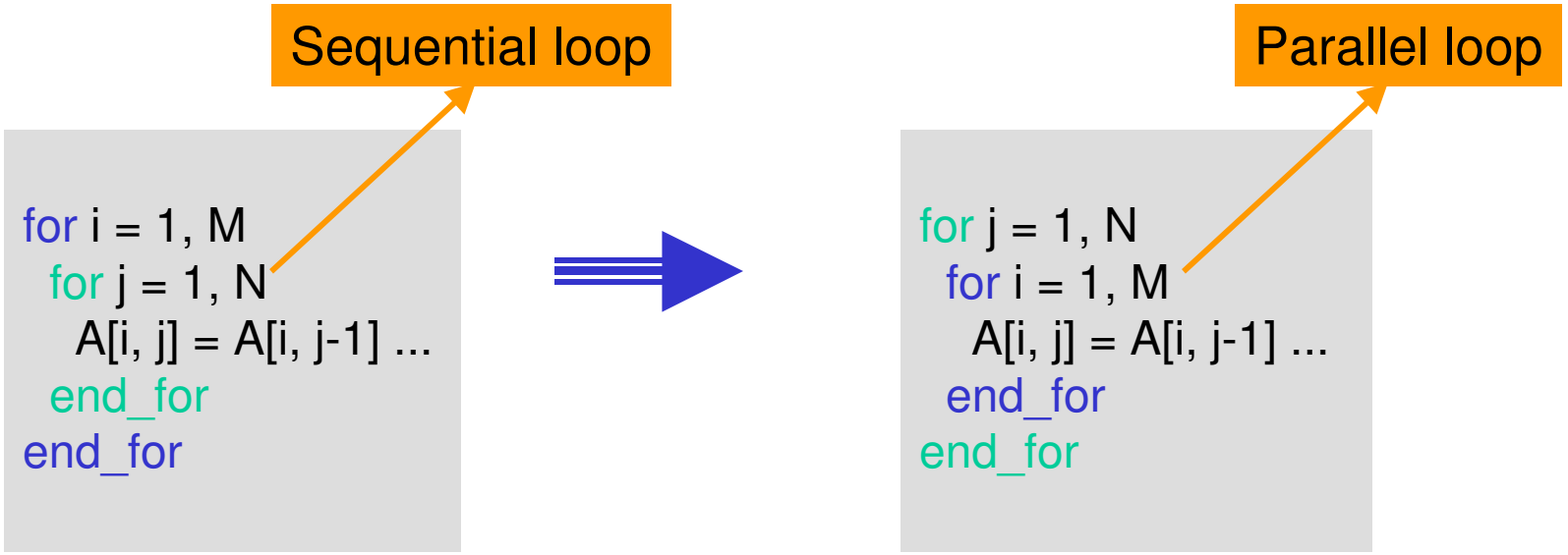


```
for i = 1, 1000, 2
  t1, t2 = ldfpd(x[i],x[i+1])
  t3, t4 = ldfpd(y[i],y[i+1])
  y[i] = t3 + a*t1
  y[i+1] = t4 + a*t2
end_for
```

IA-64 support for load-pair and early exit from software pipelining.

Transformation for Parallelism

- Loop transformation for fine-grained parallelism, e.g. software pipelining and vectorization.
- Loop transformation for coarse-grained parallelism.

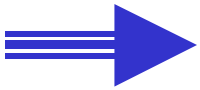


Increased parallelism for software pipelining.

Loop Unrolling for ILP

- Larger scheduling regions. Fewer dynamic branches.
- Code size may increase.

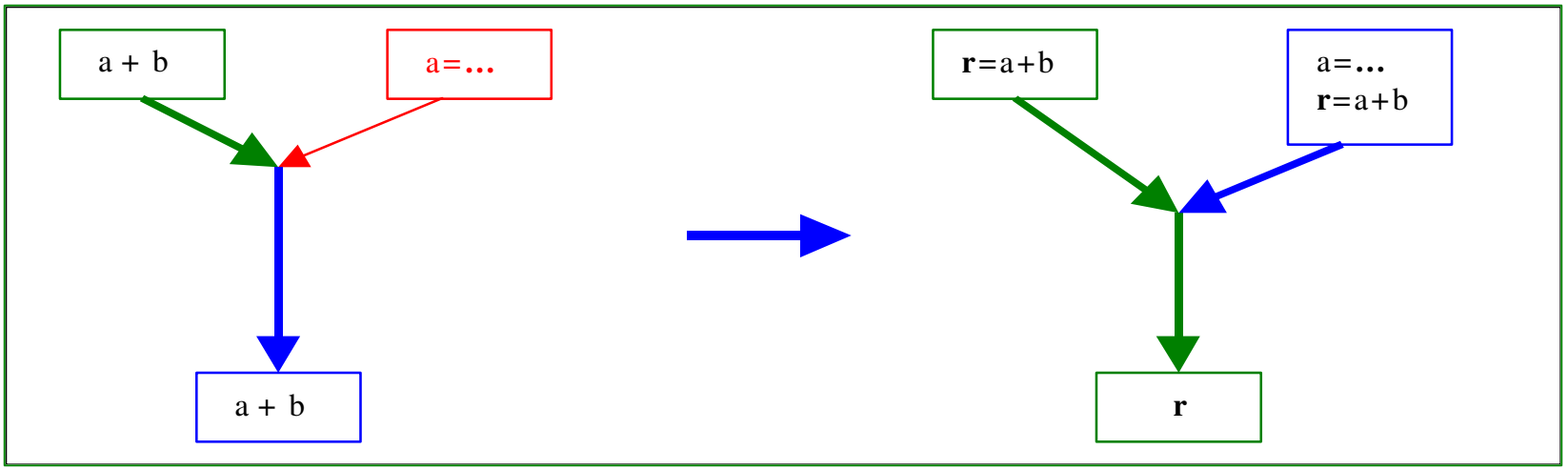
```
/* N = 2M */
for (i = 0; i<N; i++)
  a[i] = b[i];
  *p = ...
end_for
```



```
t = ld.a N;
for (i = 0; i<t/2; i=i+2)
  a[i] = b[i];
  *p = …
  t = ld.c  N
  a[i+1] = b[i+1];
  *p = …
  t = ld.c N
end_for
```

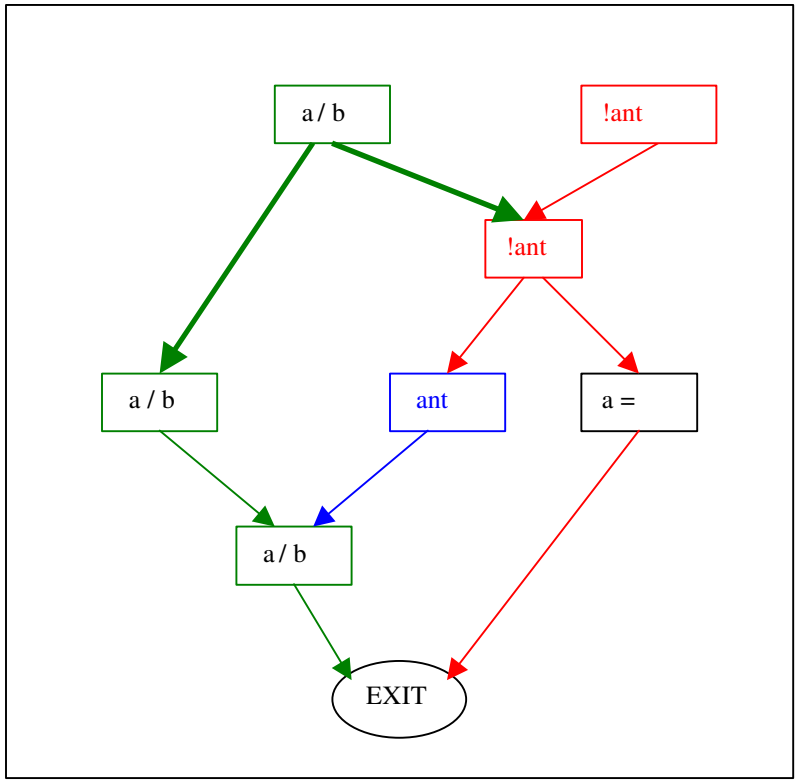
IA-64 support for data speculation.

Partial Redundancy Elimination



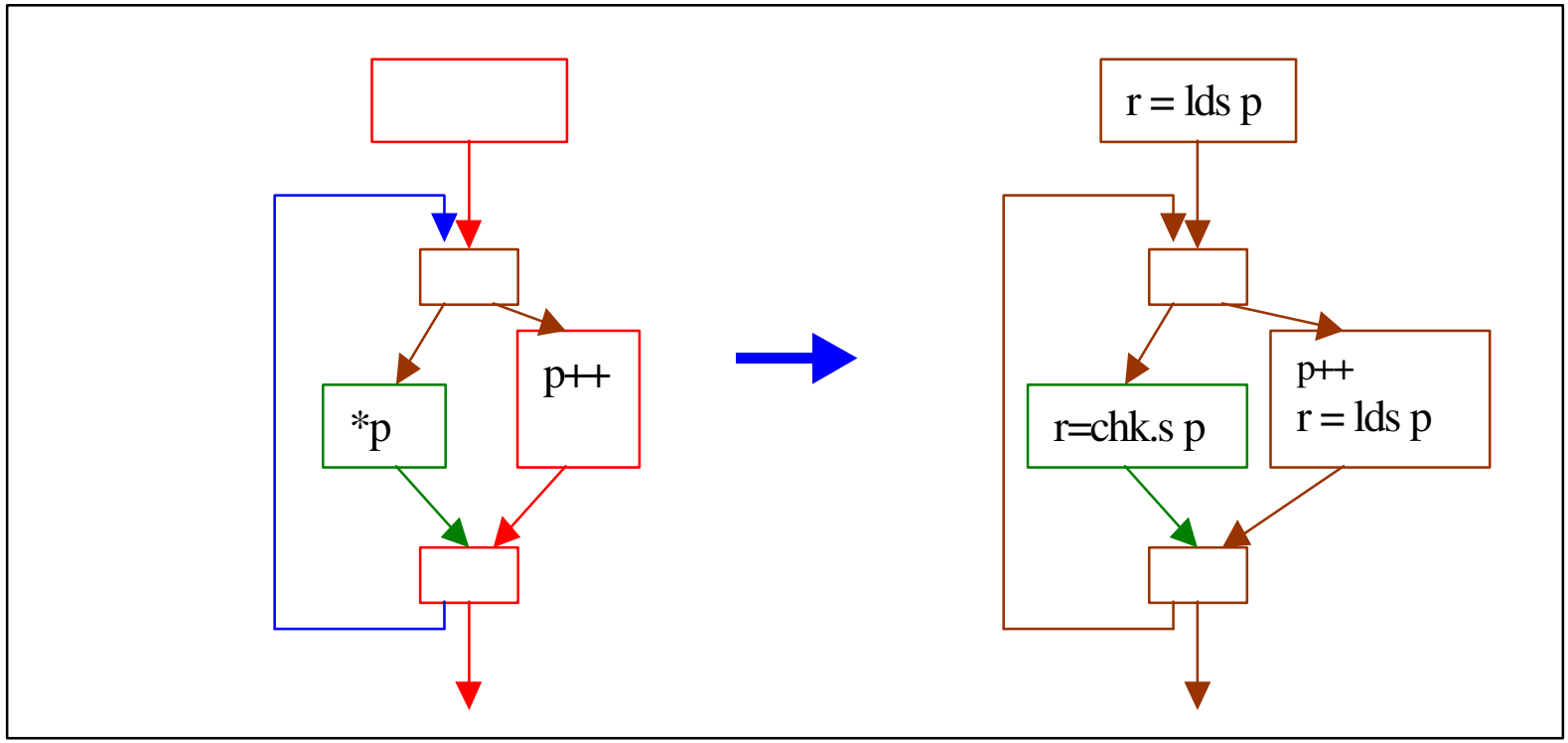
- A computation is *partially redundant* if its result is available on *some* paths.
- Remove partial redundancy by hoisting the computation to the not available path.

Anticipability and Safety



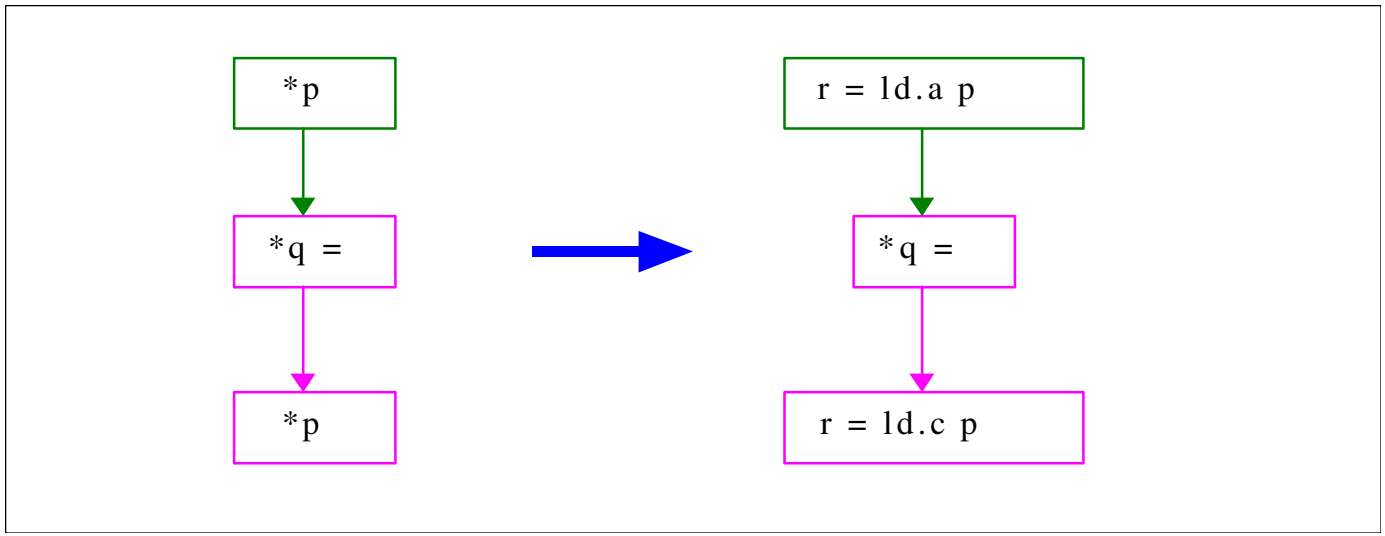
- A computation is *anticipated* at a statement S if it occurs on *all* the paths from S to the program exit.
- Hoisting anticipated computation is safe and won't increase any path length.

Control Speculation



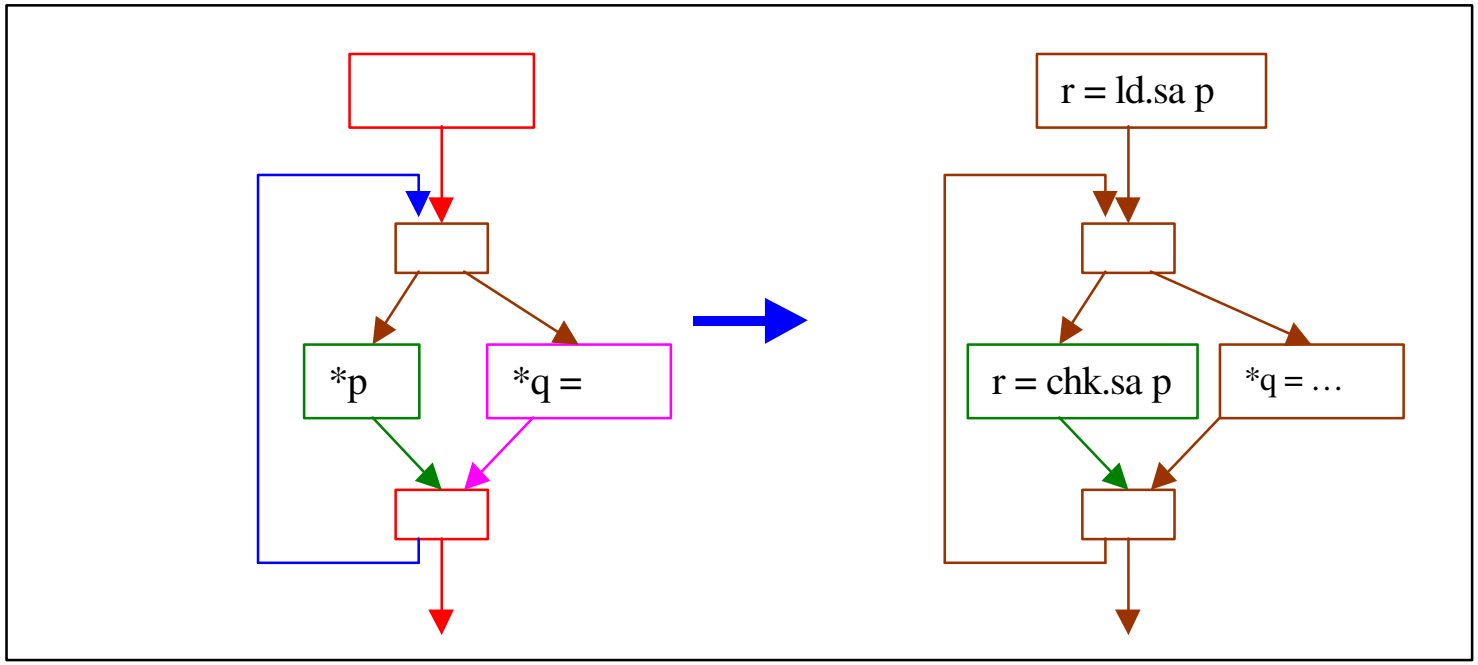
- The load is only *partially anticipated* at Loop-entry.
- Speculatively hoist the load and propagate the speculative value. Compensation after kill.

Injury and Data Speculation



- Ambiguous kill with low probability may be treated as an *injury* to an available expression.
- A cheaper repair code (ld.c) is applied to the injured value.

Combining Data/Control Speculation



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

Summary

- **Part I: IA-64 Compiler Overview**
- **Part II: Compiler Optimizations on IA-64**
  - **Procedure Inlining**
  - **Interprocedural Analysis**
  - **Code Placement**
  - **Data Dependence Analysis for Speculation**
  - **Eliminating Memory Operations**
  - **Cache Optimizations**
  - **Overlapping Memory Latency**
  - **Parallelization and Vectorization**
  - **Loop Unrolling for ILP**
  - **Partial Redundancy Elimination**