Deep Jam

Conversion of Coarse-Grain Parallelism to Instruction-Level and Vector Parallelism for Irregular Applications

Patrick Carribault[1,3,2] Albert Cohen[2] William Jalby[3]

[1]BULL SA

[2]ALCHEMY Group, INRIA Futurs

[3]LRC ITACA Laboratoy, CEA/DAM and University of Versailles

PACT'05 - Saint Louis, MO - 09/20/05

Introduction

Context:

- Uni-processor
- Loop nest optimizations
- Irregular application
  - Complex control flow
  - Unrestricted memory accesses

Contribution:

- New Transformation: Deep Jam

Preliminary Example

```
for ( i = 0 ; i < n ; i++ ) {
   p = ...;
   q = ...;
   if ( p ) {
      a[i] = ...;
      t[i] = ...;
   }
   while ( q != 0 ) {
      t[i] += a[i] ...;
      q = q << 1 ;
   }
}
```

- Irregular Control
- No outer loop carried flow dependence

- Path profiling
  - Unpredictable
  - Stable trip counts (few variations)

Preliminary Example

```
for ( i = 0 ; i < n ; i++ ) {
   p = ...;
   q = ...;
   if ( p ) {
      a[i] = ...;
      t[i] = ...;
   }
   while ( q != 0 ) {
      t[i] += a[i] ...;
      q = q << 1 ;
   }
}
```

- Itanium 2 Madison 1.5 GHz
- ICC v8.1 Compiler
- Hardware Counters:

CPU Cycles	20,489,000
Instructions	49,340,000
NOPs	19,130,000

- IPC: 2.4

Available Transformations

Performance analysis:

- Limited fine-grain parallelism
- Coarse grain parallelism available

State-of-the-art transformations:

- Operate on the inner loop
- Exploit ILP in "instruction window" for a fixed number of iterations

Beyond bounded window:

- Regular code: classical loop transformations
- More general code: Deep Jam

Deep Jam By Example

- To enlarge the scope: outer loop unrolling (e.g factor of 2)

```
for ( i = 0 ; i < n ; i+=2 ) {
   p = ...;
   q = ...;
   if ( p ) {
      a[i] = ...;
      t[i] = ...;
   }
   while ( q != 0 ) {
      t[i] += a[i] ...;
      q = q << 1 ;
   }
   p = ...;
   q = ...;
   if ( p ) {
      a[i+1] = ...;
      t[i+1] = ...;
   }
   while ( q != 0 ) {
      t[i+1] += a[i+1] ...;
      q = q << 1 ;
   }
}
```

Deep Jam By Example

- Want to group independent pairs of instructions

```
for ( i = 0 ; i < n ; i+=2 ) {
   p = ...;
   q = ...;
   if ( p ) {
      a[i] = ...;
      t[i] = ...;
   }
   while ( q != 0 ) {
      t[i] += a[i] ...;
      q = q << 1 ;
   }
   p = ...;
   q = ...;
   if ( p ) {
      a[i+1] = ...;
      t[i+1] = ...;
   }
   while ( q != 0 ) {
      t[i+1] += a[i+1] ...;
      q = q << 1 ;
   }
}
```

Deep Jam By Example

- Renaming to remove (scalar) memory based dependences

```
for ( i = 0 ; i < n ; i+=2 ) {
   p1 = ...;
   q1 = ...;
   if ( p1 ) {
      a[i] = ...;
      t[i] = ...;
   }
   while ( q1 != 0 ) {
      t[i] += a[i] ...;
      q1 = q1 << 1 ;
   }
   p2 = ...;
   q2 = ...;
   if ( p2 ) {
      a[i+1] = ...;
      t[i+1] = ...;
   }
   while ( q2 != 0 ) {
      t[i+1] += a[i+1] ...;
      q2 = q2 << 1 ;
   }
}
```

Deep Jam By Example

- Enables reordering of matching pairs of control strutures

```
for ( i = 0 ; i < n ; i+=2 ) {
   p1 = ...;
   q1 = ...;
   p2 = ...;
   q2 = ...;
   if ( p1 ) { ...}
   if ( p2 ) { ...}
```

```
   while ( q1 != 0 ) {
      t[i] += a[i] ...;
      q1 = q1 << 1 ;
   }
   while ( q2 != 0 ) {
      t[i+1] += a[i+1] ...;
      q2 = q2 << 1 ;
   }
}
```

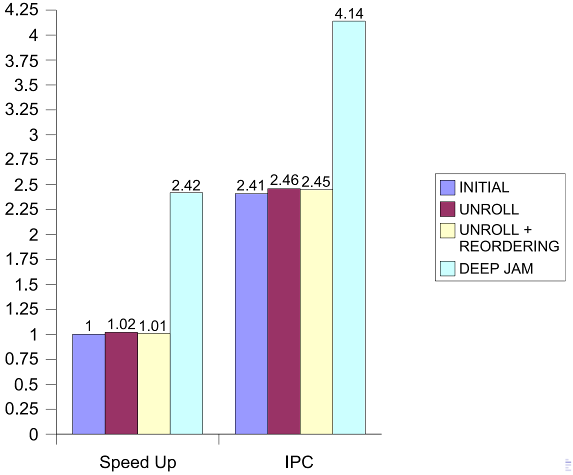
Deep Jam By Example

- Jamming `while` structures

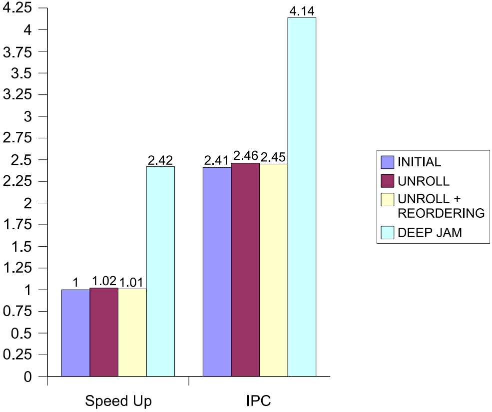
```
for ( i = 0 ; i < n ; i+=2 ) {
  ...
  while ( q1 != 0 && q2 != 0) {
    t[i] += a[i] ...;
    t[i+1] += a[i+1] ...;
    q1 = q1 << 1 ;
    q2 = q2 << 1 ;
  }
```

```
  while ( q1 != 0 ) {
    t[i] += a[i] ...;
    q1 = q1 << 1 ;
  }
  while ( q2 != 0 ) {
    t[i+1] += a[i+1] ...;
    q2 = q2 << 1 ;
  }
}
```

Experimental Results - Itanium 2 1.5Ghz - ICC v8.1



Experimental Results - Itanium 2 1.5Ghz - ICC v8.1



Questions:

- Generalization?
- When does it work?
- Best way to jam?

Outline

Software Thread Integration - STI

Procedure Jamming with intraprocedural code motion (A. Dean & W. So):

- Embedded processors and RTOS
- No dependence between threads
- Manual selection of threads

- STI statically merges logical threads
- Deep Jam iteratively extracts threadlets: candidate for merging (at compile time)

Dependence Removal

Improvement from STI: dealing with dependences.

- Keep intra-threadlet dependences
- Scalar renaming
  - SSA-like to remove inter-threadlet dependences
- Array renaming
  - e.g. ArraySSA and DeArraySSA
- Speculation
  - Break dependences speculatively (control and data)

Jamming Variants

- Static cost models + Dynamic feedback
- Depends on architecture features (ex: predication)
- Impact on compiler backend (ex: software pipelining)

Example: jamming 2 `while` structures

```
// Initial
while ( p1 )
   I1 ;
while ( p2 )
   I2 ;
```

```
// Stable trip
// counts
while ( p1 && p2 )
   I1 ; I2 ;
while ( p1 )
   I1 ;
while ( p2 )
   I2 ;
```

```
// High trip
// counts
while ( p1 || p2 )
   (p1) I1 ;
   (p2) I2 ;
```

```
// Low trip
// counts
if ( p1 ) {
   while(p1) I1 ;
   while(p2) I2 ;
} else {
   while(p2) I2 ;
}
```

Profile Based Deep Jam Algorithm

Assuming we have:

- Static or dynamic IPC evaluator
- Path Profiling (hot/cold paths)
- Value profiling (loop trip counts)

Algorithm based on feedback:

- Variant Generation
  - Test all pairs of matching threadlets with all possible jamming variants
    - Loops: unroll twice (or more?) as a jamming variant
  - In practice: limit the depth of exhaustive search
- Profitability Evaluation

Benchmarks

2 benchmarks:

- Cryptanalysis: SHA-0 Attack
- Computational Biology: ABNDM/BPM

Architecture:

- Intel Itanium 2 Madison Processor
- CPU@1.5 GHz

SHA-0 Attack - Presentation

- Context:
  - SHA-0 cryptanalysis
  - Lead to a full collision in August 2004 [EUROCRYPT'05]
- Source code:
  - Irregular
  - Very complex control flow (many early exits)
  - Main loop to test potential colliding messages
- Deep Jam:
  - 2 iterations of the main loop

SHA-0 Attack - Experimental Results

- Deep Jam on initial version: 12,8% speedup
  - Speedup due to path profiling
  - Reduce number of dynamic instructions
  - Increase IPC
  - Number of bubbles: 45% less
- Deep Jam on vectorized version: 49% speedup

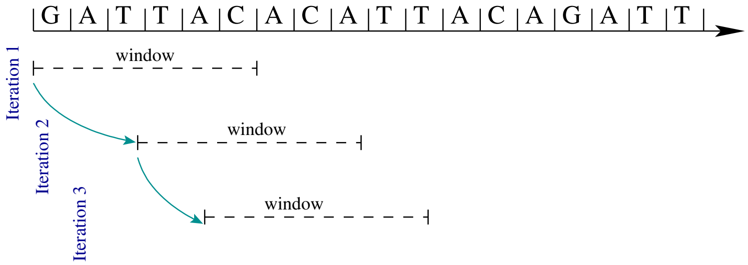
ABNDM/BPM - Presentation

Approximate pattern matching algorithm allowing a fixed number of errors.

- Dynamic programming
- Filtering
- Bit parallelism

ABNDM/BPM - Presentation

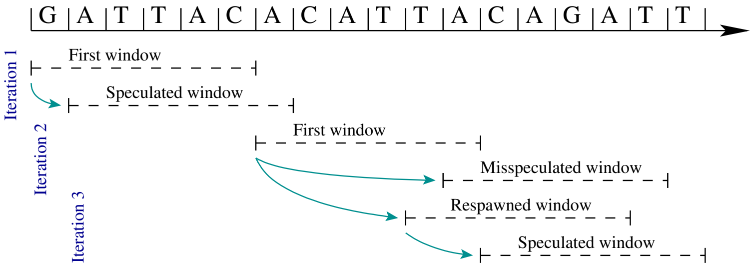
- Main loop iterating on text trough windows



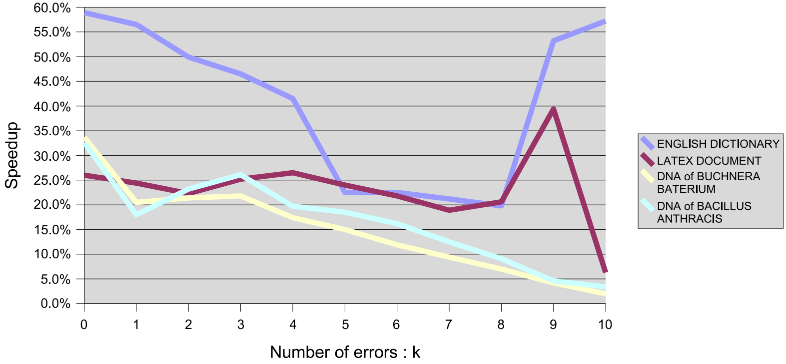
- Window: 2 nested while loops with conditionals
- Deep Jam two consecutive windows

ABNDM/BPM - Presentation

- Need to speculate on skip between windows: adaptive sheme to dynamically set skip distance



ABNDM/BPM - Experimental Results



Conclusion & Future Works

Deep Jam:

- New Transformation
  - Inspired from Unroll&Jam and STI
  - Agressive dependence removal
- Optimizing irregular code
  - Speedup on 2 benchmarks
- Profile based algorithm

Future Work:

- Narrowing variant space automatically
- Improving profitability evaluation