Uncovering Hidden Loop Level Parallelism in Sequential Applications

Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, Scott Mahlke

Advanced Computer Architecture Lab.
University of Michigan
CMP Architectures

• **Multiple cores on a chip**
  – Higher throughput
  – Reduced complexity (per core)
  – More power/heat friendly

• **Multithreaded applications**
How About Single Thread?

[Source: Bridges et al, MICRO '07]
Loop Level Parallelization

DOALL loop

i = 0-39
Loop Level Parallelization

DOALL loop

i = 0-39

Core 0

i = 0-19

Core 1

i = 20-39
Loop Level Parallelization

Speculative DOALL loop

\[ i = 0-39 \]
Loop Level Parallelization

Speculative DOALL loop

Core 0

Core 1

Loop Chunk

i = 0-9

i = 10-19

i = 20-29

i = 30-39

i = 0-39
Loop Level Parallelization

Speculative DOALL loop

**Bad news:** limited number of parallel loops in general purpose applications

–1.3x speedup for SpecINT2000 on 4 cores

Core 0

Core 1
Contributions

• **Code generation framework**
  – Speculative parallelization of uncounted loops

• **Compiler transformations**
  – Speculative loop fission
  – Isolation of infrequent dependences
  – Speculative prematerialization
Target Architecture

![Diagram of a processor architecture with four cores and two L2 caches.](image-url)
Target Architecture

Scalar operand network

Core 0
Core 1
Core 2
Core 3

L2 cache

L2 cache
Target Architecture

Scalar operand network

Hardware transactional memory

L2 cache

Core 0

Core 1

Core 2

Core 3
Code Generation Framework

```c
for (i=0; i<n; i++)
  // original loop code
```
Code Generation Framework

```
while (...) 
IS+=...; IE+=...; 
XBEGIN 
for (i=IS;i<IE;i++) 
// original loop code 
XCOMMIT 
```
Code Generation Framework

while (...)
IS+=...; IE+=...;
XBEGIN

for (i=IS;i<IE;i++)
// original loop code

RECV(THREAD_{j-1})
XCOMMIT
SEND(THREAD_{j+1})
Code Generation Framework

while (...) IS+=...; IE+=...; XBEGIN

for (i=IS;i<IE;i++) // original loop code

RECV(THREAD_{j-1})
XCOMMIT
SEND(THREAD_{j+1})
Code Generation Framework

while (...) IS+=...; IE+=...; XBEGIN
for (i=IS;i<IE;i++)
// original loop code
if (brkCond) break;
RECV(THREAD_{j-1})
XCOMMIT
SEND(THREAD_{j+1})
Code Generation Framework

while (...)  
IS+=...; IE+=...;  
XBEGIN  
if (globalBrk) break;

for (i=IS;i<IE;i++)  
// original loop code  
if (brkCond)  
  localBrk=1; break;

RECV(THREAD_{j-1})  
XCOMMIT  
if (localBrk)  
  globalBrk=1; abortOtherTXs;  
SEND(THREAD_{j+1})
Code Generation Framework

```c
for (i=IS;i<IE;i++)
    // original loop code
    if (brkCond)
        localBrk=1; break;

while (...) IS+=...; IE+=...;
XBEGIN
if (globalBrk) break;

RECV(THREAD_{j-1})
XCOMMIT
if (localBrk)
    globalBrk=1; abortOtherTXs;
SEND(THREAD_{j+1})

Consolidation

Spawn
```
Code Generation Framework

- Supports **counted** and **uncounted** loops
  - Software managed control speculation
- Iteration chunking
- Enforce transaction ordering
- Handles livein, liveout & accumulator registers
DOALL Coverage – Provable and Profiled
DOALL Coverage – Provable and Profiled
DOALL Coverage – Provable and Profiled

Still not good enough!
Few dependences hinder parallelization in many loops
DOALL Coverage – Provable and Profiled

Still not good enough!
Few dependences hinder parallelization in many loops

Compiler can help:
• Speculative fission
• Isolation of infrequent paths
• Speculative prematerialization
Speculative Loop Fission

```c
1: while (node) {
2:   work(node);
3:   node = node->next;
}
```
Speculative Loop Fission

1: while (node) {
2:   work(node);
3:   node = node->next;
4:   node_array[count++] = node;
}
Speculative Loop Fission

1: while (node) {
2:   work(node);
3:   node = node->next;
}

1: while (node) {
4:   node_array[count++] = node;
3:   node = node->next;
}

XBEGIN
5: node = node_array[IS];
   i = 0;
1':while (node && i++ < CS) {
2:    work(node);
3':   node = node->next;
}
RECV(THREAD\textsubscript{j-1})
XCOMMIT
SEND(THREAD\textsubscript{j+1})
}
Speculative Loop Fission

```c
while (node) {
    work(node);
    node = node->next;
}
```

```c
while (node) {
    node_array[count++] = node;
    node = node->next;
}
```

```c
node = node_array[IS];
i = 0;
while (node && i++ < CS) {
    work(node);
    node = node->next;
}
```

```c
if (node!= node_array[IS+CS]){
    update_node_array;
    kill_other_threads();
}
```

```
XBEGIN
5: node = node_array[IS];
i = 0;
1':while (node && i++ < CS) {
2: work(node);
3': node = node->next;
}
RECV(THREADj-1)
XCOMMIT
if (node!= node_array[IS+CS]){
    update_node_array;
    kill_other_threads();
}
SEND(THREADj+1)
```
Speculative Loop Fission

1: while (node) {
  2:   work(node);
  3:   node = node->next;
}

1: while (node) {
  4:   node_array[count++] = node;
  3:   node = node->next;
}

XBEGIN
5: node = node_array[IS];
   i = 0;
1':while (node && i++ < CS) {
  2:   work(node);
  3':  node = node->next;
}
RECV(THREADj-1)
XCOMMIT
if (node!= node_array[IS+CS]){
   update_node_array;
   kill_other_threads();
   SEND(THREADj+1)
}

compilers creating custom processors
Speculative Loop Fission

1: while (node) {
2:   work(node);
3:   node = node->next;
}

1: while (node) {
4:   node_array[count++] = node;
3:   node = node->next;
}

XBEGIN
5: node = node_array[IS];
   i = 0;
1': while (node && i++ < CS) {
2:    work(node);
3':   node = node->next;
}

RECV(THREAD_{j-1})
XCOMMIT
if (node != node_array[IS+CS]){
    update_node_array;
    kill_other_threads();
}
SEND(THREAD_{j+1})

Speculative Loop Fission

```
1: while (node) {
2:   work(node);
3:   node = node->next;
}
```

```
1: while (node) {
4:   node_array[count++] = node;
3:   node = node->next;
}
```

```
XBEGIN
5: node = node_array[IS];
i = 0;
1':while (node && i++ < CS) {
2:    work(node);
3':   node = node->next;
}
RECV(THREADj-1)
XCOMMIT
if (node!= node_array[IS+CS]){
    update_node_array;
    kill_other_threads();
}
SEND(THREADj+1)
```
Speculative Loop Fission

1: while (node) {
2:   work(node);
3:   node = node->next;
}

XBEGIN
5: node = node_array[IS];
   i = 0;
1':while (node && i++ < CS) {
2:    work(node);
3':   node = node->next;
}
RECV(THREADj-1)
XCOMMIT
if (node!= node_array[IS+CS]){
   update_node_array;
   kill_other_threads();
} SEND(THREADj+1)

1: while (node) {
4:   node_array[count++] = node;
3:   node = node->next;
}
Infrequent Dependence Isolation

Diagram:

- Node 1: 99%
- Node 2: 1%
- Node C

Connections:
- A to 1: 99%
- B to 2: 1%
- C to 1 and 2

University of Michigan
Electrical Engineering and Computer Science
Infrequent Dependence Isolation

Diagram:

- Node A with an arrow to Node C (99%)
- Node B with an arrow to Node A (1%)
- Node C with an arrow to Node B

After isolation:

- Node A remains unchanged
- Node B remains unchanged
- Node C is isolated
- The arrow from Node B to Node C is removed
Infrequent Dependence Isolation
Infrequent Dependence Isolation
Infrequent Dependence Isolation

Sample loop from yacc benchmark

```c
for( j=0; j<=nstate; ++j ){
    if( tystate[j] == 0 ) continue;
    if( tystate[j] == best ) continue;
    count = 0;
    cbest = tystate[j];
    for (k=j; k<=nstate; ++k)
        if (tystate[k]==cbest) ++count;
    if ( count > times) {
        best = cbest;
        times = count;
    }
}
```
for( j=0; j<=nstate; ++j ){
    if( tystate[j] == 0 ) continue;
    if( tystate[j] == best ) continue;
    count = 0;
    cbest = tystate[j];
    for (k=j; k<=nstate; ++k)
        if (tystate[k]==cbest) ++count;
    if ( count > times) {
        best = cbest;
        times = count;
    }
}
for( j=0; j<=nstate; ++j ){
    if( tystate[j] == 0 ) continue;
    if( tystate[j] == best ) continue;
    count = 0;
    cbest = tystate[j];
    for (k=j; k<=nstate; ++k)
        if (tystate[k]==cbest) ++count;
    if ( count > times) {
        best = cbest;
        times = count;
    }
}
for( j=0; j<=nstate; ++j ){
  if( tystate[j] == 0 ) continue;
  if( tystate[j] == best ) continue;
  count = 0;
  cbest = tystate[j];
  for (k=j; k<=nstate; ++k)
    if (tystate[k]==cbest) ++count;
  if ( count > times) {
    best = cbest;
    times = count;
  }
}

Infrequent Dependence Isolation

if ( count > times) {
  best = cbest;
  times = count;
}

Sample loop from yacc benchmark
for( j=0; j<=nstate; ++j ){
    if( tystate[j] == 0 ) continue;
    if( tystate[j] == best ) continue;
    count = 0;
    cbest = tystate[j];
    for (k=j; k<=nstate; ++k)
        if (tystate[k]==cbest) ++count;
    if (count > times) {
        best = cbest;
        times = count;
    }
}

Infrequent Dependence Isolation

1 %

Sample loop from yacc benchmark

j=0;
while (j<=nstate){
    for( ; j<=nstate; ++j ){
        if( tystate[j] == 0 ) continue;
        if( tystate[j] == best ) continue;
        count = 0;
        cbest = tystate[j];
        for (k=j; k<=nstate; ++k)
            if (tystate[k]==cbest) ++count;
        if (count > times) {
            best = cbest;
            times = count;
            j++;
        }
    }
}

1 %
Infrequent Dependence Isolation

for( j=0; j<=nstate; ++j ){
    if( tystate[j] == 0 ) continue;
    if( tystate[j] == best ) continue;
    count = 0;
    cbest = tystate[j];
    for (k=j; k<=nstate; ++k)
        if (tystate[k]==cbest) ++count;
    if ( count > times) {
        best = cbest;
        times = count;
    }
}

if (count > times) {
    best = cbest;
    times = count; j++;
}
Infrequent Dependence Isolation

Sample loop from yacc benchmark

```c
for( j=0; j<=nstate; ++j )
    if( tystate[j] == 0 ) continue;
    if( tystate[j] == best ) continue;
    count = 0;
    cbest = tystate[j];
    for( k=j; k<=nstate; ++k )
        if( tystate[k] == cbest ) ++count;
    if( count > times )
        best = cbest;
        times = count;
```

```c
while (j<=nstate){
    for( ; j<=nstate; ++j )
        if( tystate[j] == 0 ) continue;
        if( tystate[j] == best ) continue;
        count = 0;
        cbest = tystate[j];
        for( k=j; k<=nstate; ++k )
            if( tystate[k] == cbest ) ++count;
        if( count > times )
            break;
    if( count > times )
        best = cbest;
        times = count; j++;
}
```
DOALL Coverage – Profiled and Transformed

Fraction of sequential execution

<table>
<thead>
<tr>
<th>SPEC FP</th>
<th>SPEC INT</th>
<th>Mediabench</th>
<th>Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>052.alvinn</td>
<td>056.ear</td>
<td>171.swim</td>
<td>177.mesa</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>173.equake</td>
<td>188.ammp</td>
<td>188.ear</td>
</tr>
<tr>
<td>183.equake</td>
<td>008.espresso</td>
<td>023.eqntott</td>
<td>026.compress</td>
</tr>
<tr>
<td>072.sc</td>
<td>099.go</td>
<td>105.122</td>
<td>124.m88km3</td>
</tr>
<tr>
<td>129.compress</td>
<td>130.li</td>
<td>132.ijpeg</td>
<td>164.gzip</td>
</tr>
<tr>
<td>175.vpr</td>
<td>181.mcf</td>
<td>197.parser</td>
<td>256.bzip2</td>
</tr>
<tr>
<td>300.twolf</td>
<td>300.twolf</td>
<td>300.twolf</td>
<td>300.twolf</td>
</tr>
</tbody>
</table>

profiles + provable
DOALL Coverage – Profiled and Transformed

SPEC FP
SPEC INT
Mediabench
Utilities

052.alvinn
056.ear
171.swim
177.mgrid
179.art
183.equake
188.ammp
008.espresso
023.eqntott
026.compress
072.sc
099.go
124.m88ksim
129.compress
130.ii
132.ljpeg
164.gzip
172.mgrid
175.vpr
177.mesa
181.mcf
197.parser
256.bzip2
300.twolf
Cjpeg
djpeg
epic
g721decode
g721encode
gsmdec
gsmenc
mpeg2dec
mpeg2enc
mpeg2vid
gpeg
rawaudio
rawajpeg
unepic
grep
lex
yacc
average

Fraction of sequential execution
profiled + provable
transformations
Coverage Breakdown

- DOALL loops
- Speculative fission
- Speculative prematerialization
- Infrequent dependence isolation
- Control speculation for uncounted loops
- DOALL loops after transformations

The graph shows the fraction of sequential execution for different benchmarks:

- **SpecINT**
  - DOALL loops: 10%
  - Speculative fission: 20%
  - Speculative prematerialization: 5%
  - Infrequent dependence isolation: 1%
  - Control speculation for uncounted loops: 5%
  - DOALL loops after transformations: 2%

- **MediaBench**
  - DOALL loops: 20%
  - Speculative fission: 30%
  - Speculative prematerialization: 15%
  - Infrequent dependence isolation: 5%
  - Control speculation for uncounted loops: 10%
  - DOALL loops after transformations: 5%

- **Utilities**
  - DOALL loops: 15%
  - Speculative fission: 25%
  - Speculative prematerialization: 20%
  - Infrequent dependence isolation: 5%
  - Control speculation for uncounted loops: 15%
  - DOALL loops after transformations: 10%
Coverage Breakdown

- **DOALL loops**
- **Speculative fission**
- **Speculative prematerialization**
- **Infrequent dependence isolation**
- **Control speculation for uncounted loops**
- **DOALL loops after transformations**

Fraction of sequential execution

<table>
<thead>
<tr>
<th></th>
<th>SpecINT</th>
<th>MediaBench</th>
<th>Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOALL loops</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speculative fission</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speculative prematerialization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Infrequent dependence isolation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control speculation for uncounted loops</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOALL loops after transformations</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experimental Setup

- **OpenIMPACT compiler**
- **Multicore simulator**
  - Simulates up to 8 ARM9-like processors
  - Models scalar operand network
  - Assumes perfect memory system
  - Uses STM library to emulate HTM functionality
Speedup

1.36x, 1.84x and 2.34x speedup on 2-, 4-, and 8-cores
Conclusion

• Figure out ways to use available resources for legacy applications
  – Codes like error handlers, linked list & tree traversal limit parallelism
• Compiler analysis and optimization looks promising
• 1.84x speedup on 4 cores after transformations compared to 1.41x
Questions?

Thank you!
SpecDSWP vs. Speculative Fission
SpecDSWP vs. Speculative Fission
Speculative Prematerialization

```c
for (...) {
    current = ...;
    work(last);
    last = current;
}
```
Speculative Prematerialization

```plaintext
for (...) {
  1:  current = ...;
  2:  work(last);
  3:  last = current;
}
```

```plaintext
XBEGIN
1’: current =
3’: last =
   for (...) {
  1:  current = ...;
  2:  work(last);
  3:  last = current;
}
XCOMMIT
```