Study on Multi-threading Partitioning Policies

Jinfeng Liu
ECE259       June, 2000
Objective

- **Thread partitioning policy**
  - Break sequential program into parallel threads
  - Critical to the performance of multi-threading processor in extracting ILP -- TLP

- **Current topic of study**
  - Evaluate Different thread partitioning schemes in multi-threading processor
    - Loop
    - Procedure
    - Fixed size instruction chunk
    - MEM-slicing
  - Exploit “adaptive partitioning algorithm” that can apply the optimal partitioning scheme according to the program behavior adaptively
# Existing Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Features</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
</table>
| Trace    | • Fixed size instruction chuck                                | • Good load balance                     | • No knowledge of dependencies  
• Can select bad partitions  
• Need to keep history |
| MEM-Slice| • Partition at memory references after skipping minimum thread size | • Good load balance  
• May eliminate one level dependencies | • Create threads randomly  
• No knowledge of dependencies  
• Can select bad partitions  
• Difficult to keep history |
| Loop     | • Natural loop boundary as thread partition  
• Innermost loops only                                      | • Low creation overhead  
• Speedup do-all loops  
• Simple fetch engine to all TU | • High inter-thread dependencies  
• Parallel only for innermost loops  
• TU idle at sequential exec  
• Squash at outer loops  
• Squash at control misspeculation |
| Procedure| • Leaf procedure threads  
• Non-leaf procedure threads  
• After procedure threads | • Low creation overhead  
• Good control speculation | • Bad load balance |
Experiment Model

- Trace-driven model
- *Trace producer* generates traces by applying different partitioning schemes while parsing the program
- Run-time traces are buffered by *trace buffer*
- *Trace consumer* simulates a multi-threading processor that executes multiple traces in parallel
Trace Producer

Instruction

- SimpleScalar Simulator
- Partition Unit
- Trace Table

- Data Access Table

- Trace ID: M – N + 1
  - Register Access Table
  - Memory Access Table

- Trace ID: M – N + 2
  - Register Access Table
  - Memory Access Table

- Trace ID: M
  - Register Access Table
  - Memory Access Table

- Dependence Checking

- Instruction Buffer
- Trace Buffer
### Trace Producer

<table>
<thead>
<tr>
<th>Entry PC</th>
<th>Exit PC</th>
<th># Branch</th>
<th>Br. Hist.</th>
<th>Size</th>
<th># Exec.</th>
<th>Live-in, Live-out …</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry PC</td>
<td>Exit PC</td>
<td># Branch</td>
<td>Br. Hist.</td>
<td>Size</td>
<td># Exec.</td>
<td>Live-in, Live-out …</td>
</tr>
<tr>
<td>Entry PC</td>
<td>Exit PC</td>
<td># Branch</td>
<td>Br. Hist.</td>
<td>Size</td>
<td># Exec.</td>
<td>Live-in, Live-out …</td>
</tr>
</tbody>
</table>

- **Trace table**
  - Keep information of each dynamic trace
  - Similar to trace cache except for the whole instruction stream body
  - Indexed by trace *Entry PC*
  - Tagged by *Entry PC, Exit PC* and *Branch History*
  - 1K lines with unlimited set-associativity
## Data Access Table

- Record each reg/mem access of most recent N trace executions (N = # of thread unit)
- Determine inter-thread data dependence
Trace Producer

 Partition Unit

- Decision is made based on different partition policies
- Easy to apply different policies without affecting other structures
- Separated mechanism and policy – various policies can be applied on the same mechanism
- Allow to alternate policies at run-time
Each trace buffer entry locates an instructions stream in instruction buffer.

- Instruction buffer holds instruction stream and dependency information.
- Both buffers are recycles in circular queue structure.
Trace Consumer

- Thread units
  - Organized in circular queue
  - Uni-direction data forward
  - In-order startup, in-order completion
  - Local register file and load/store queue
  - Out-of-order, multiple issue
  - Synchronize for true dependencies – stall the pipeline
  - No speculative execution

- Thread unit arbiter
  - Manage the circular ring structure
  - Fetch trace from trace buffer and dispatch to the tail TU
  - Send command to each TU (start, complete, and etc.)
Each thread unit is defined as an finite-state machine.
Simulation

- Thread unit simulation model
  - Total 8 thread units
  - Cycle-by-cycle simulation
  - Each TU runs its own FSM
  - Each state has a dedicated cycle counter
  - Thread unit arbiter only communicates with head and tail TU
  - Thread creation, commitment and pipeline startup overhead – P, Q, R cycles – can be customized by software
  - Instruction execution time – K cycles – depending on instruction type
  - Total execution slots can be broken down by cycles spent on each state
  - Procedure partition is not simulated, since it requires out-of-order thread execution
Simulation

- Limited speed up
- Execution time is sensitive to thread creation and completion overhead
Most cycles are spent on synchronization due to tight dependency chain

Load imbalance also seriously degrades IPC (idle in loop, retirement in basic block)
Conclusion and Future Work

- Reasons for low IPC
  - Synchronization on tight dependency chain
  - Load imbalance

- Solutions
  - Out-of-order execution, multiple-issue on TU
  - Data speculation – no synchronization
  - Compiler support – forward live-out value early

- Future work
  - Out-of-order execution and multiple issue TU model
  - Control speculation – trace prediction
  - Data speculation – value prediction, speculative load
  - Adaptive partitioning – optimize partition policies adaptively at run-time