Multiple Threads and Future-Generation Architectures

Guri Sohi
Overview

- Thread types
- Supporting threads
- Opportunities provided by threads
- Will not talk about why technology trends favor underlying hardware capable of supporting multiple threads
  - wire delays
  - ease of design
  - ease of verification
  - ease of “adaptability”
What is a thread?

- A sequence of instructions
- state (registers, memory)
Thread types

• "Traditional" threads
• Independent (non-speculative) program fragments
• Speculative threads
  ◦ control-driven
  ◦ helper/scout/data-driven
Issues with Threads

- Grain size
- Thread management (spawning, execution, ..)
- Synchronization
Traditional Threads

- Different programs (processes)
- Multithreaded single program (lightweight processes, kernel threads)
  - Typically programmer specified
- Typically very coarse grain (1K-10K+ instructions)
- Scheduled by OS (dispatcher) on underlying hardware platform
  - Heavy weight
- Parallel execution increases system throughput but does not decrease single program execution time
Independent Program Fragments

- Carved from single program either by programmer or (parallelizing) compiler
  - E.g., iterations of a DOALL loop
- Medium to large grain -- 100-1K+ instructions
- Parallel execution can improve single program execution time
- Hard to extract automatically
Independent Program Fragments -- Observations

- Start with sequence of instructions in a total (control-driven) order
- Create threads which execute in a partial order
- Maintain overall total order
  - respect original ordering of observable events (e.g., dependences) in thread creation
Speculative Threads

Rationale: use speculation to overcome barriers to extraction of traditional (independent) threads

• Threads whose effect is not architectural without other events
• Useful when more traditional (independent) threads can’t be extracted from program
Speculative Control-Driven Threads

• Consider overall (dynamic) program execution order
  ◦ traversal of static program CFG
• Divide into multiple pieces
• Consider each piece a “speculative thread”
  ◦ Ordered speculative threads re-create total program order
  ◦ threads could execute in parallel using control and data-dependence speculation
    - no guarantees of control and data independence
for (indx = 0; indx < BUFSIZE; indx++) {
    /* get the symbol for which to search */
    symbol = SYMVAL(buffer[indx]);

    /* do a linear search for the symbol in the list */
    for (list = listhd; list; list = LNEXT(list)) {
        /* if symbol already present, process entry */
        if (symbol == LELE(list)) {
            process(list);
            break;
        }
    }

    /* if symbol not found, add it to the tail */
    if (! list) {
        addlist(symbol);
    }
}
Speculative Control-Driven Threads

- Speculative threads execute, buffering speculative state
- Confirmation of correct speculation allows speculative state to become architectural
- Example: Space-time computing in MAJC
Speculative Data-driven Threads: Motivation

- Program execution is typically processing of low-latency instructions, with pauses for long-latency events (e.g., cache misses, branch mispredicts)
- What if long-latency events did not cause stalls?
  - processing of low-latency instructions
- How to tolerate long latencies?
  - initiate long-latency events earlier
    - traditional solution -- scheduling -- full of problems
Speculative Data-driven Threads

- Isolate computation leading to long-latency event
  - use speculation to facilitate isolation
  - use speculation to trade off size vs. accuracy
- Isolated computation is speculative data-driven thread
  - data-driven because instructions of thread not contiguous in original program order
- Execute speculative thread in parallel with normal control-driven thread (original program)
Motivation Example

RETIREMENT STREAM

TIME

BRANCH
branch mispredict

LOAD

cache miss

Pre-execution

FORK

sub program

BRANCH OUTCOME

AVOID MIS_PREDICTION

Avoid Misprediction
Speculative Data-driven Thread: Example

**BACKWARD SLICE**

- **lda**: r8, -8432(r29)
- **cmoveq**: r18, r1, r0
- **ldl**: r1, -19952(r29)
- **s4addq**: r16, r8, r8
- **stl**: r31, 0(r8)
- **and**: r4, r5, r5
- **sll**: r5, 4, r5
- **ldq**: r23, -19408(r29)
- **ldl**: r27, -19944(r29)
- **addl**: r1, 1, r1
- **addq**: r0, r5, r5
- **bis**: r31, r31, r0
- **stl**: r1, -19952(r29)
- **ldq**: r7, 8(r5)

*follow dependences backward from criterion instruction*

*both data and control*

**Criterion Instruction**
Speculative Data-driven Threads

- **Trigger**: point in the program from which slice will be forked
- Trigger selection is a research topic
  - trade-off between latency tolerance and slice size
Positioning of Data-driven Threads

- Unverified
  - Speculative DDT

- Verified
  - Application Code

Software
Supporting Threads

- Thread management: initiating threads and providing thread context
- Synchronization, if need be
  - Helper threads may not need values to be passed architecturally
Initiating and Managing Threads

• Provide pointer to starting instruction plus sequencing method
• Provide working thread context
  ◦ Is a “spare” register file available?
  ◦ Can the appearance of a spare register file be provided?
  ◦ How are initial values of registers provided?

• Managing threads and thread initiation influenced by how working thread context can be provided
Synchronization

• How are values passed between threads, and how are they synchronized?
• Values passed through memory namespace; synchronized through synchronization namespace (in memory)
  ◦ high overhead
  ◦ better solution in single-chip environment, with different types of threads?
• Values passed through register namespace; synchronized through reservations on registers
  ◦ E.g., speculative threads in multiscalar
• What other options?
  ◦ efficient synchronization key to fine-grain threads
Example: Threads in Multiscalar

<table>
<thead>
<tr>
<th>Targ Spec</th>
<th>Branch, Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ1</td>
<td>OUTER</td>
</tr>
<tr>
<td>Targ2</td>
<td>OUTERFALLOUT</td>
</tr>
<tr>
<td>Create mask</td>
<td>$4,$8,$17,$20,$23</td>
</tr>
</tbody>
</table>

```
OUTER:
    addu $20, $20, 16
    ld    $23,  SYMVAL−16($20)
    move $17, $21
    beq  $17, $0, SKIPINNER
INNER:
    ld    $8,  LELE($17)
    bne  $8,  $23, SKIPCALL
    move $4, $17
    jal  process
    j    INNERFALLOUT
SKIPCALL:
    ld    $17,  NEXTLIST($17)
    bne  $17, $0, INNER
INNERFALLOUT:
    release $8, $17
    bne $17, $0, SKIPINNER
    move $4, $23
    jal addlist
SKIPINNER:
    release $4
    bne $20, $16, OUTER
OUTERFALLOUT:
```

Stop Bits
```
F
```

Forward Bits
```
F
```
Opportunities Provided by Multiple Threads/Sequencers

- Understand performance tricks used in traditional (single-threaded) architectures
  - Instruction scheduling
  - Dealing with branches
 Instruction Scheduling

- Operation in program can only be initiated when the operation is “seen” by the execution hardware.
- Traditionally, operations “seen” by hardware when (single) sequencer reaches operation:
  - sequencer traversing static program representation.
- Static scheduling used to move operation “earlier”:
  - “earlier” positioning in executable allows earlier initiation.
Sequencer/Scheduling Interplay

• Single sequencer
  • Schedule is facilitated by placing instructions in static representation
    • move instructions “up”
  • statically orchestrating schedule very important
Sequencer/Scheduling Interplay

• Multiple sequencers
  • Schedule is an ensemble of pipelined schedules
  • moving instructions “up” less important
  • statically orchestrating schedule less important
**Artifacts of Single Sequencers**

- **Wide instructions**: to sequence through (and schedule) more than one operation at a time
- **Predicated instructions**: work around branches in scheduling
  - poor man’s way of getting “multiple” flows of control
- **Non-trapping instructions**: to allow “early” placement of high-latency instructions
- **Software Pipelining**: to allow overlapped execution of multiple loop iterations

*Less important with multiple sequencers*
**Sequencers and Register Sets**

<table>
<thead>
<tr>
<th>Static Sequencing Model</th>
<th>Dynamic Sequencing Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Narrow</td>
<td>Single, Narrow</td>
</tr>
<tr>
<td>Single Narrow</td>
<td>Single register set</td>
</tr>
<tr>
<td>Single Wide</td>
<td>Various forms</td>
</tr>
<tr>
<td>Multiple</td>
<td></td>
</tr>
</tbody>
</table>

With multiple register sets, need to increase the size of each set becomes less important
Questions

• Can we treat different thread types uniformly?
• How to represent threads in executable?
• How to initiate threads efficiently?
• How to deal with variable context availability?
• How to synchronize efficiently?
• How to emulate performance optimizations for single sequencers with multiple sequencers?