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 instr uctions
- state (r egisters, memory)

Thread types

- "T raditional" threads
- Independent (non-speculativ e) program fragments
- Speculativ e threads
  - control-driven
  - helper/scout/data-driven

Issues with Threads

- Gr ain size
- Thr ead management (spawning, execution, ..)
- Synchr onization

Traditional Threads

- Dif ferent programs (processes)
- Multithr eaded single program (lightweight processes, kernel threads)
  - typically programmer specified
- T ypically very coarse grain (1K-10K+ instructions)
- Scheduled b y 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 lar ge grain -- 100-1K+ instructions
- P arallel execution can improve single program execution time
- Har d to extract automatically

Independent Program Fragments -- Observations

- Start with sequence of instructions in a total (control-driven) order
- Cr eate threads which execute in a partial order
- Maintain o verall 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 w hen more traditional (independent) threads can't be extracted from program

Speculative Control-Driven Threads

- Consider o verall (dynamic) program execution order
  - traversal of static program CFG
- Divide into m ultiple pieces
- Consider each piece a "speculativ e 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

Example

```
for (indx = 0; indx < BUFSIZE; indx++) {
    /* get the symbol for which to search */
    symbol = SYMVAL(buffer[indx]);

    /* do a linear search fo rthe 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
- Confir mation of correct speculation allows speculative state to become architectural
- Example: Space-time computing in MAJC

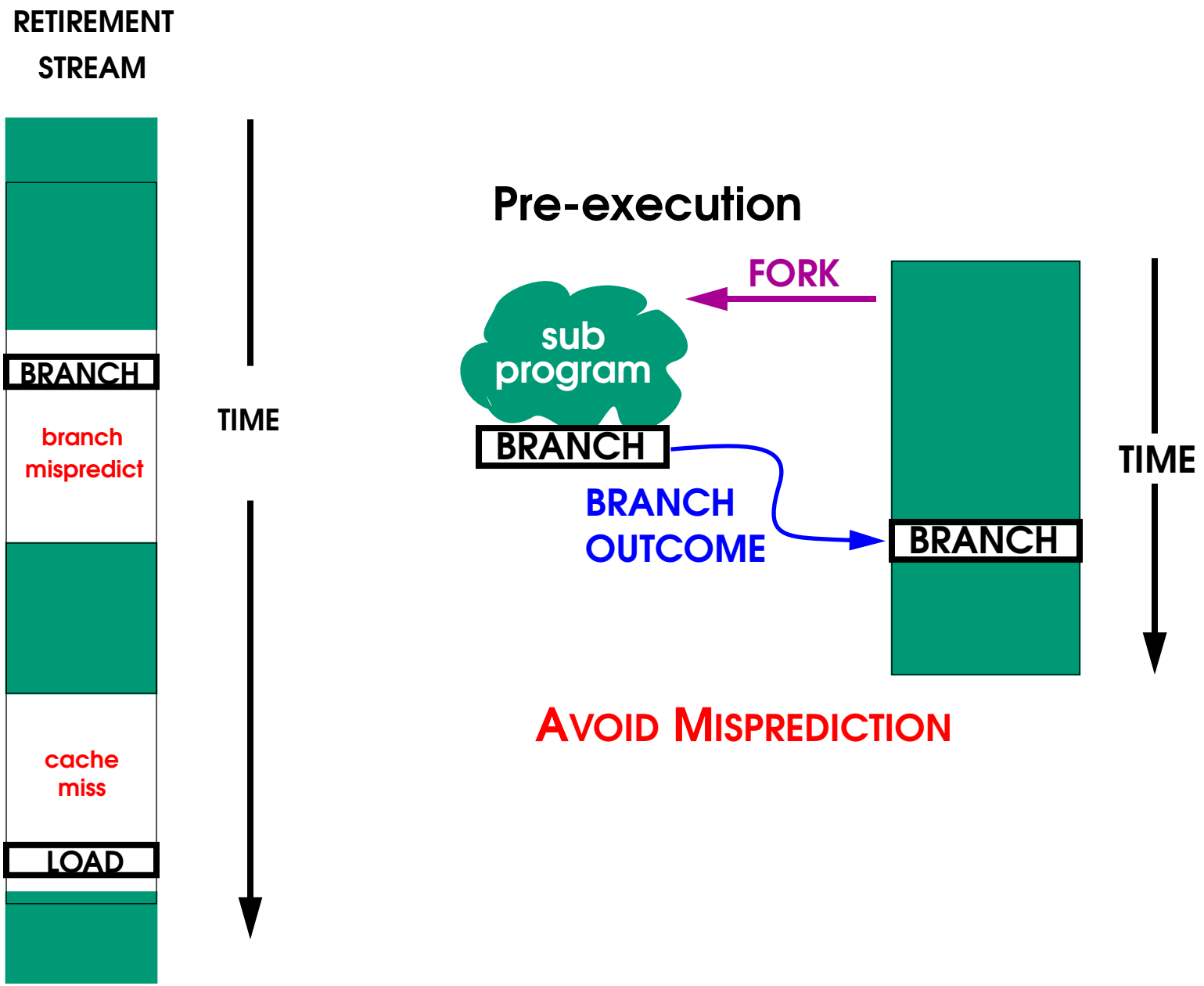
Speculative Data-driven Threads: Motivation

- Pr ogram execution is typically processing of low-latency instructions, with pauses for long-latency events (e.g., cache misses, branch mispredicts)
- What if long-latenc y events did not cause stalls?
  - processing of low-latency instructions
- Ho w to tolerate long latencies?
  - initiate long-latency events earlier
    - traditional solution -- scheduling -- full of problems

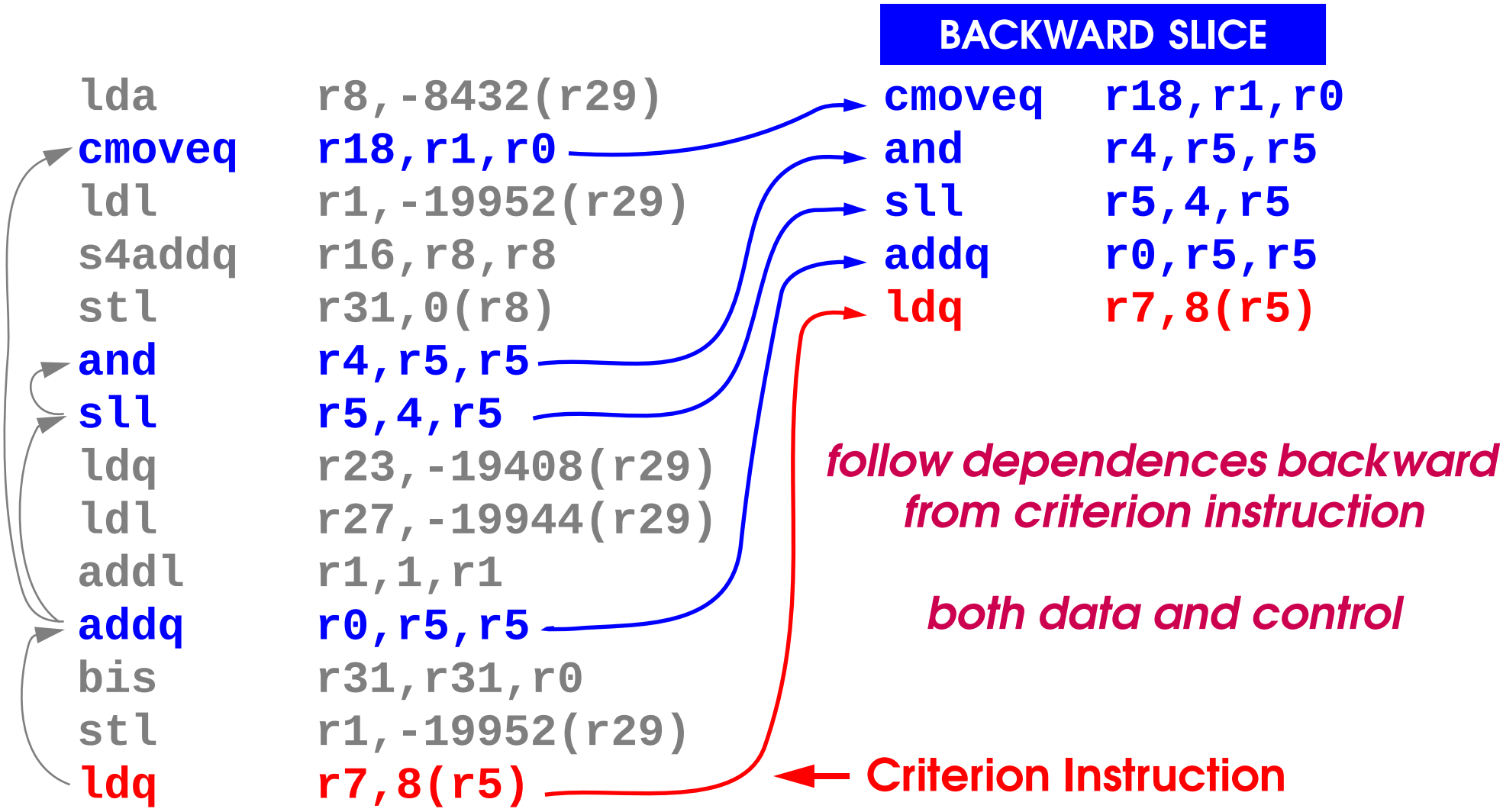
Speculative Data-driven Threads

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

Motivation Example

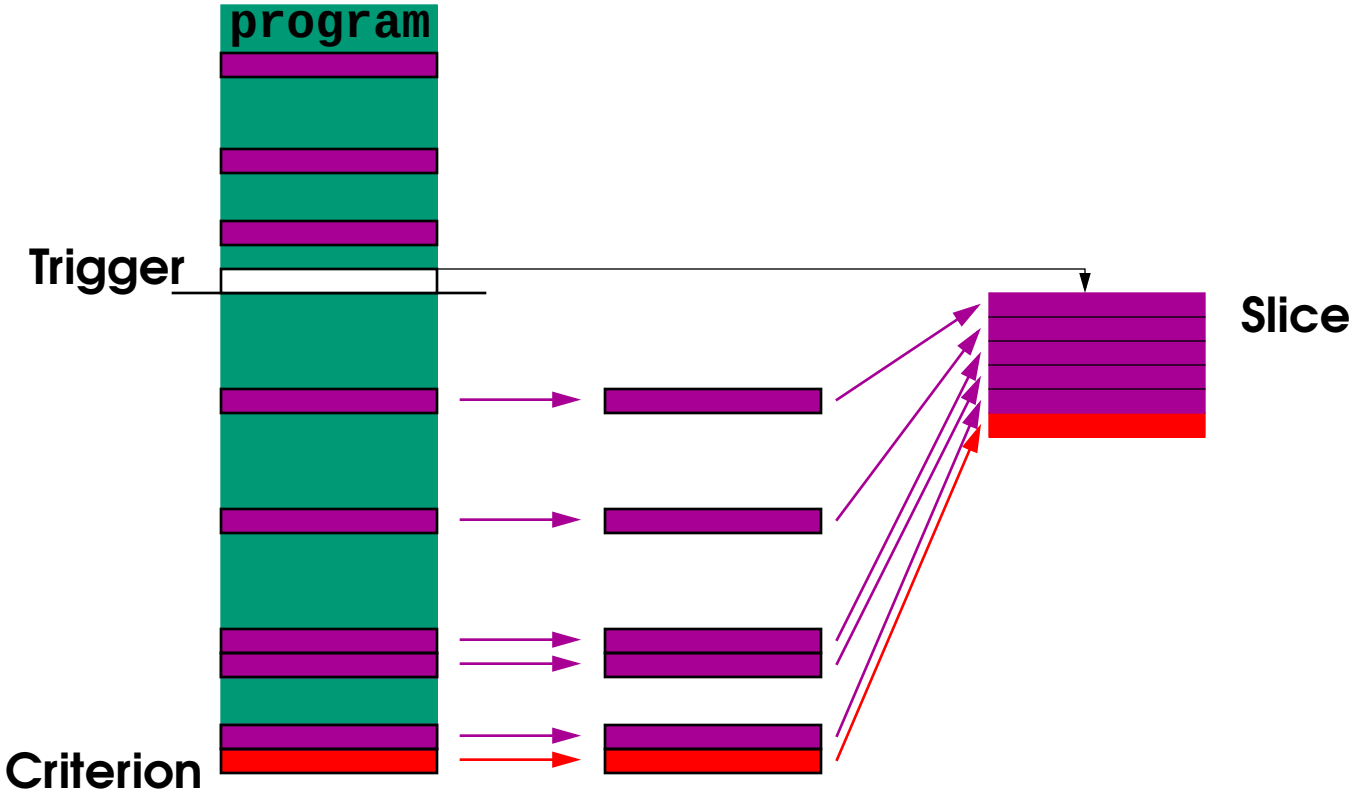


Speculative Data-driven Thread: Example

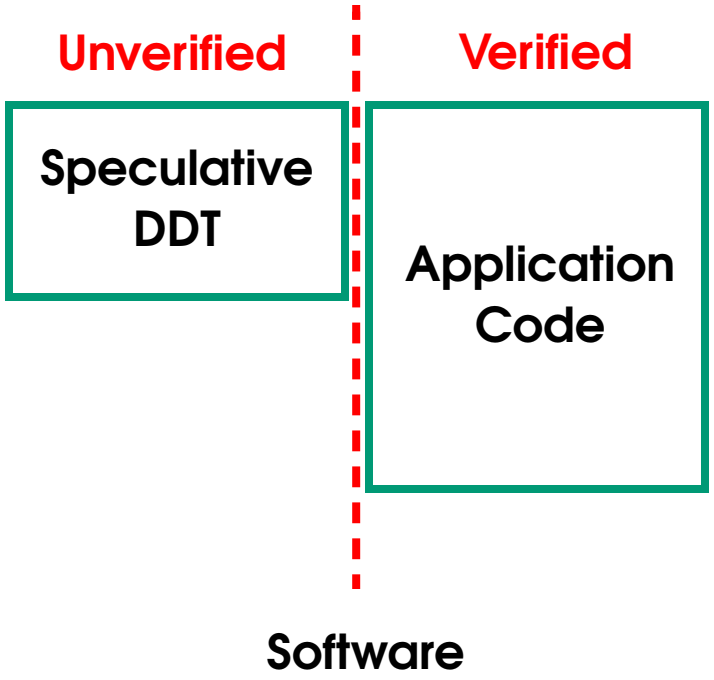


Speculative Data-driven Threads

- 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



Supporting Threads

- Thr ead management: initiating threads and providing thread context
- Synchr onization, if need be
  - Helper threads may not need values to be passed architecturally

Initiating and Managing Threads

- Pr ovide pointer to starting instruction plus sequencing method
- Pr ovide 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

Targ Spec	Branch, Branch
Targ1	OUTER
Targ2	OUTERFALLOUT
Create mask	$4,$8,$17,$20,$23

```
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:
```

Forward Bits	Stop Bits
F	
F	
F	
	Stop Always

Opportunities Provided by Multiple Threads/Sequencers

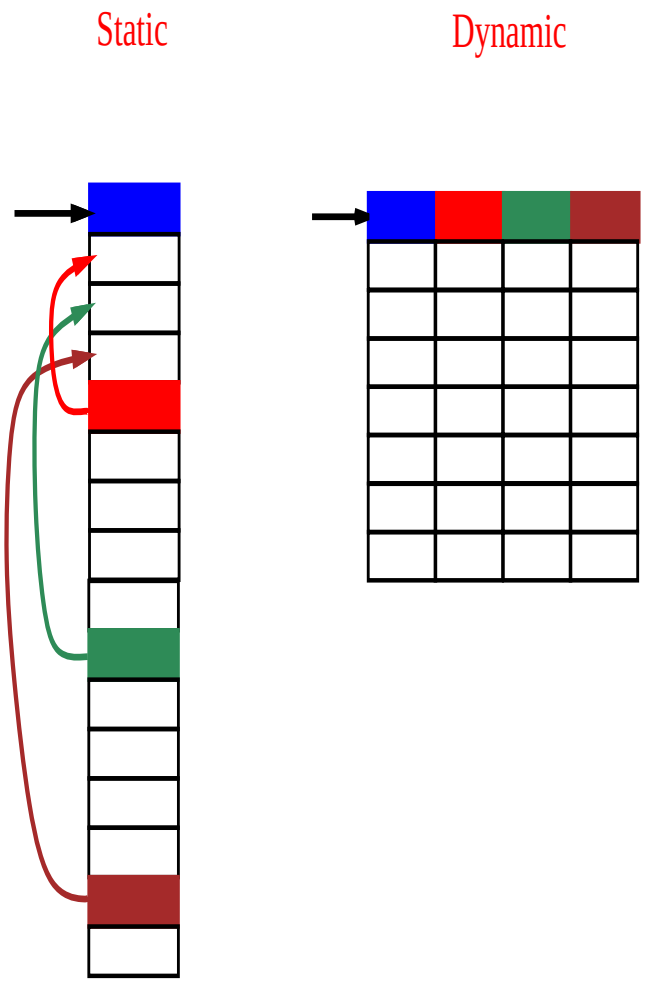
- Under stand performance tricks used in traditional (single-threaded) architectures
  - Instruction scheduling
  - Dealing with branches

Instruction Scheduling

- Oper ation in program can only be initiated when the operation is "seen" by the execution hardware
- T raditionally, operations "seen" by hardware when (single) sequencer reaches operation
  - sequencer traversing static program representation
- Static scheduling used to mo ve operation "earlier"
  - "earlier" positioning in executable allows earlier initiation

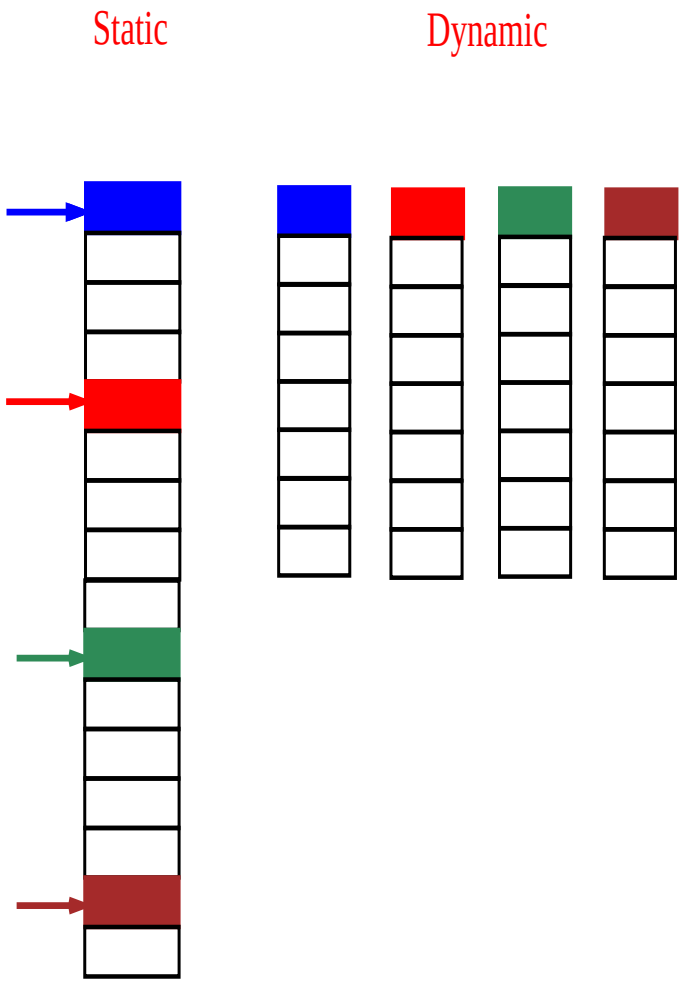
Sequencer/Scheduling Interplay

- 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 a 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

Dynamic Sequencing Model

Static Sequencing Model

	Single, Narrow	Single, Wide	Multiple
Single Narrow	Single register set	Renamed registers	Multiple physical, single logical register set
Single Wide		Various forms	
Multiple			Multiple, small, register sets

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

Questions

- Can w e treat different thread types uniformly?
- Ho w to represent threads in executable?
- Ho w to initiate threads effi ciently?
- Ho w to deal with variable context availability?
- Ho w to synchronize effi ciently?
- Ho w to emulate performance optimizations for single sequencers with multiple sequencers?