Computer Architecture: Speculation (in Parallel Machines)

Prof. Onur Mutlu
Carnegie Mellon University

Readings: Speculation

- Required
  - Sohi et al., "Multiscalar Processors," ISCA 1995.
  - Zhou, "Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window," PACT 2005.
  - Herlihy and Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," ISCA 1993.
  - Rajwar and Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," MICRO 2001.

- Recommended
  - Colohan et al., "A Scalable Approach to Thread-Level Speculation," ISCA 2000.
  - Akkary and Driscoll, "A dynamic multithreading processor," MICRO 1998.
  - And, many others

Speculation in Parallel Machines

Speculation

- Speculation: Doing something before you know it is needed.

- Mainly used to enhance performance

- Single processor context
  - Branch prediction
  - Data value prediction
  - Prefetching

- Multi-processor context
  - Thread-level speculation
  - Transactional memory
  - Helper threads

Speculative Parallelization Concepts

- Idea: Execute threads unsafely in parallel
  - Threads can be from a sequential or parallel application

- Hardware or software monitors for data dependence violations

- If data dependence ordering is violated
  - Offending thread is squashed and restarted
- If data dependences are not violated
  - Thread commits
  - If threads are from a sequential order, the sequential order needs to be preserved → threads commit one by one and in order

Inter-Thread Value Communication

- Can happen via
  - Registers
  - Memory

- Register communication
  - Needs hardware support between processors
  - Dependences between threads known by compiler
  - Can be producer initiated or consumer initiated
  - If consumer executes first:
    - consumer stalls, producer forwards
  - If producer executes first
    - producer writes and continues, consumer reads later
  - Can be implemented with Full/Empty bits in registers

Memory Communication

- Memory dependences not known by the compiler
- True dependencies between predecessor/successor threads need to be preserved

- Threads perform loads speculatively
  - get the data from the closest predecessor
  - keep record that they read the data (in L1 cache or another structure)
- Stores performed speculatively
  - buffer the update while speculative (write buffer or L1)
  - check successors for premature reads
  - if successor did a premature read: squash
  - typically squash the offending thread and all successors

Dependences and Versioning

- Only true data dependence violations should cause a thread squash
- Types of dependence violations:
  - LD A → ST A: name dependence; hardware may handle
  - ST A → ST A: name dependence; hardware may handle
  - ST A → LD A: true dependence; causes a squash

- Name dependences can be resolved using versioning
- Idea: Every store to a memory location creates a new version

- Example: Gopal et al., "Speculative Versioning Cache," HPCA 1998.

Where to Keep Speculative Memory State

- Separate buffers
  - E.g. store queue shared between threads
  - Address resolution buffer in Multiscalar processors
  - Runahead cache in Runahead execution

- L1 cache
  - Speculatively stored blocks marked as speculative
  - Not visible to other threads
  - Need to make them non-speculative when thread commits
  - Need to invalidate them when thread is squashed

Speculation to “Parallelize” Single-Threaded Programs

Referenced Readings

- Sohi et al., "Multiscalar Processors," ISCA 1995.
- Herlihy and Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," ISCA 1993.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
- Gopal et al., "Speculative Versioning Cache," HPCA 1998.
- Steffan et al., "A Scalable Approach to Thread-Level Speculation," ISCA 2000.
- Franklin and Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," IEEE TC 1996.
- Moshovos et al., "Dynamic Speculation and Synchronization of Data Dependences," ISCA 1997.
- Chrysos and Emer, "Memory Dependence Prediction using Store Sets," ISCA 1998.
- Dubois and Song, "Assisted Execution," USC Tech Report 1998.
- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.
- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.
- Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.
- Sundaramoorthy et al., "Slipstream Processors: Improving both Performance and Fault Tolerance," ASPLOS 2000.
- Zhou, "Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window," PACT 2005.

Thread Level Speculation

- Speculative multithreading, dynamic multithreading, etc…

- Idea: Divide a single instruction stream (speculatively) into multiple threads at compile time or run-time
  - Execute speculative threads in multiple hardware contexts
  - Merge results into a single stream

- Hardware/software checks if any true dependencies are violated and ensures sequential semantics
- Threads can be assumed to be independent
- Value/branch prediction can be used to break dependencies between threads
- Need to verify such predictions: can be done by executing a “safe version” or checking invariants

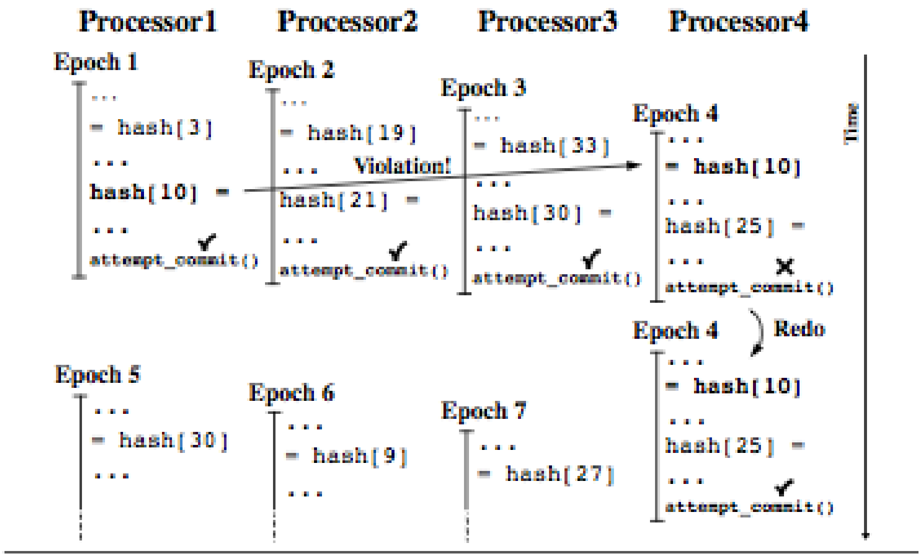
Thread Level Speculation Example

- Steffan et al., "A Scalable Approach to Thread-Level Speculation," ISCA 2000.

(a) *Example psuedo-code*

```
while(continue_condition) {
    ...
    x = hash[index1];
    ...
    hash[index2] = y;
    ...
}
```

(b) *Execution using thread-level speculation*



TLS Conflict Detection Example

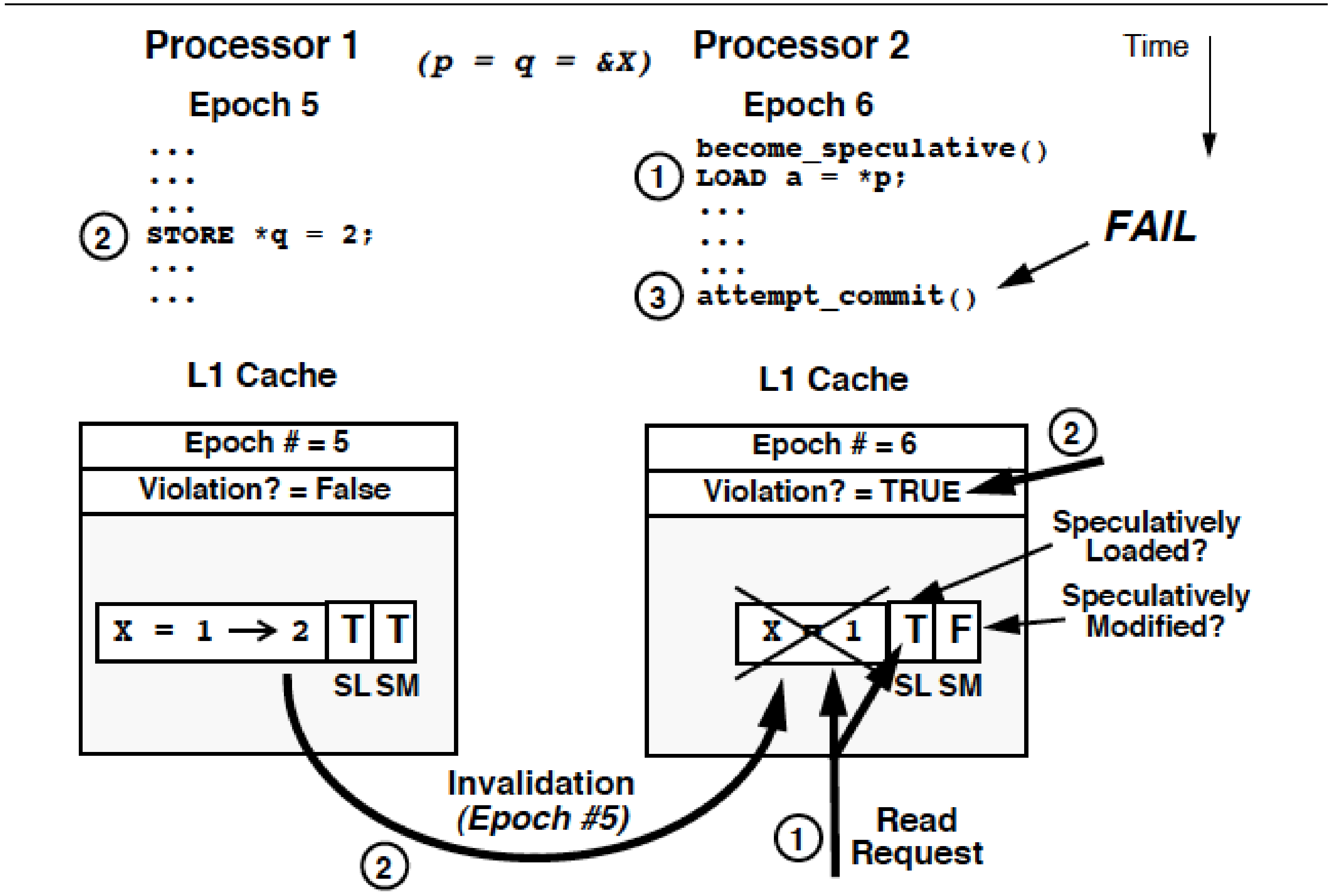


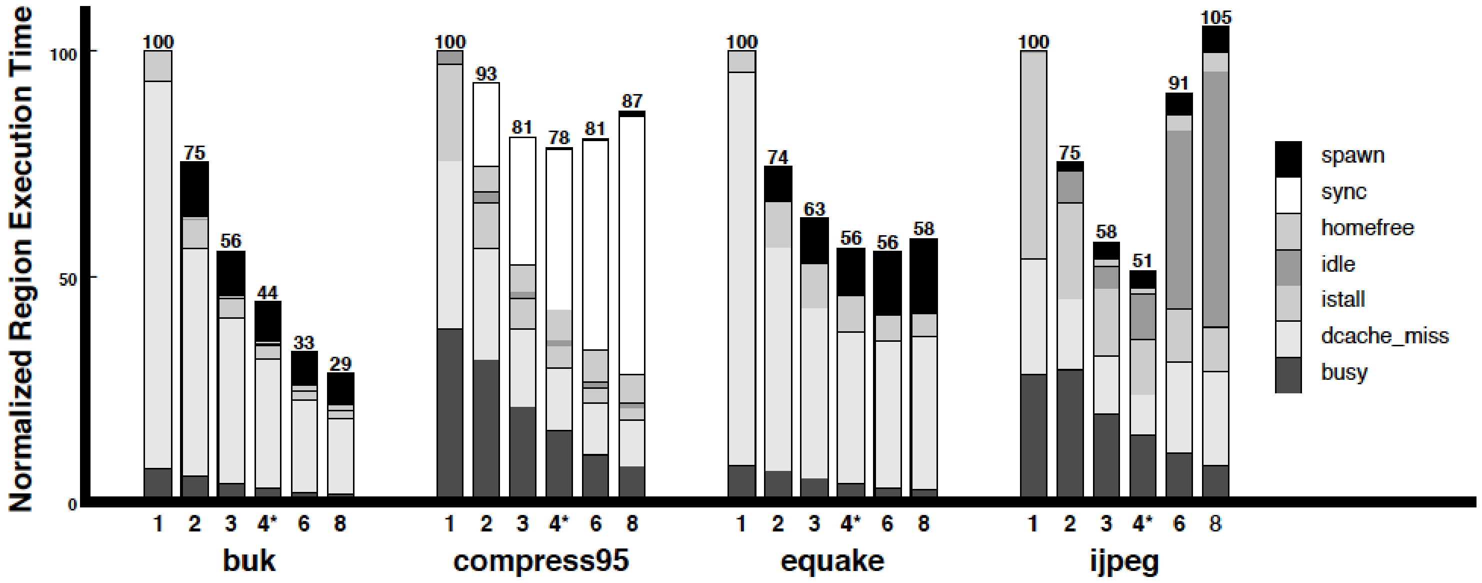
Figure 2. Using cache coherence to detect a RAW dependence violation.

Some Sample Results [Colohan+ ISCA 2000]

Table 3. Performance impact of TLS on our baseline architecture (a four-processor single-chip multiprocessor).

Application	Overall Region Speedup	Parallel Coverage	Program Speedup
buk	2.26	56.6%	1.46
compress95	1.27	47.3%	1.12
equake	1.77	39.3%	1.21
ijpeg	1.94	22.1%	1.08

(a) *Execution Time*



Multiscalar Processors (ISCA 1992, 1995)

- Exploit "implicit" thread-level parallelism within a serial program
- Compiler divides program into tasks
- Tasks scheduled on independent processing resources

- Hardware handles register dependences between tasks
  - Compiler specifies which registers should be communicated between tasks
- Memory speculation for memory dependences
  - Hardware detects and resolves misspeculation

- Franklin and Sohi, "The expandable split window paradigm for exploiting fine-grain parallelism," ISCA 1992.
- Sohi et al., "Multiscalar processors," ISCA 1995.

Multiscalar vs. Large Instruction Windows

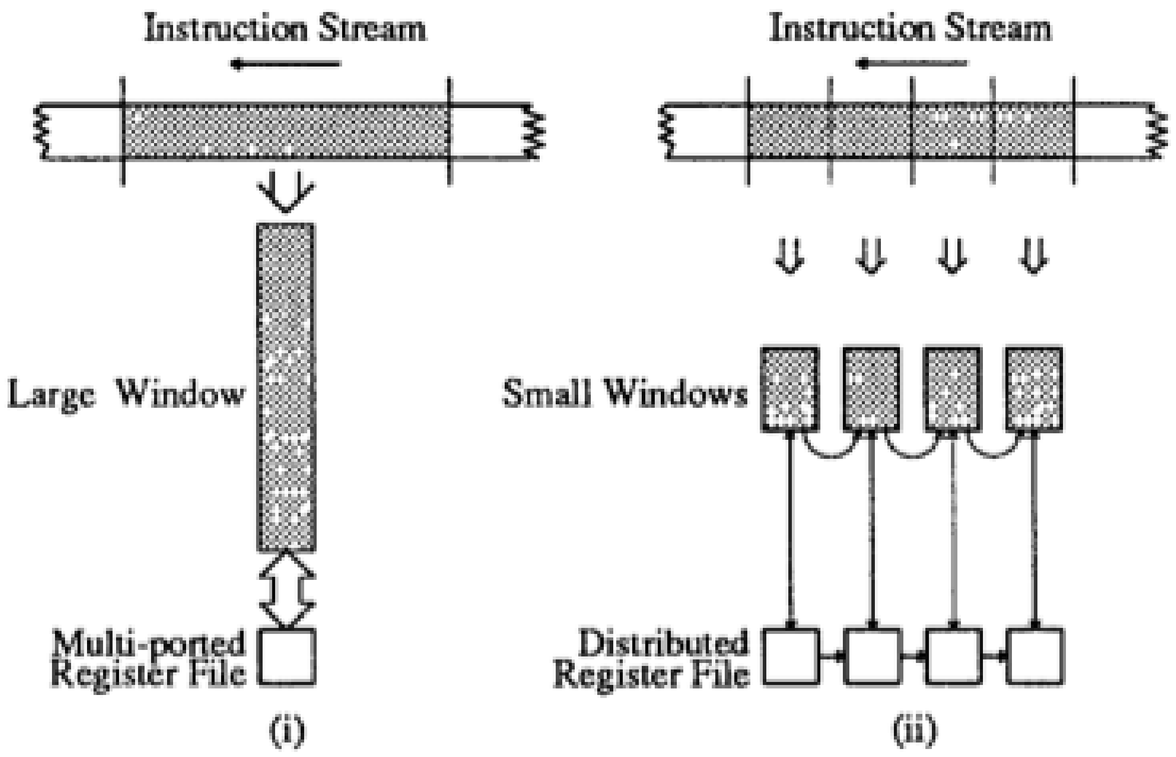
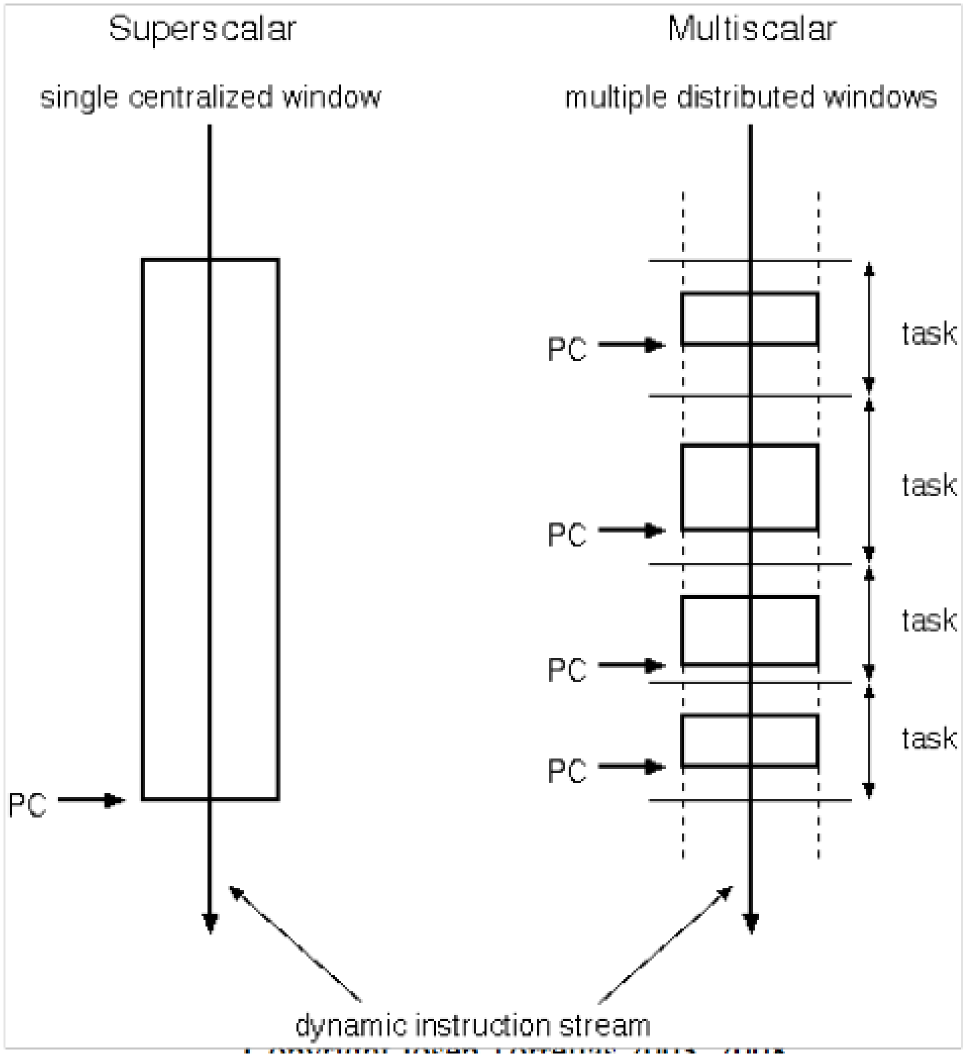


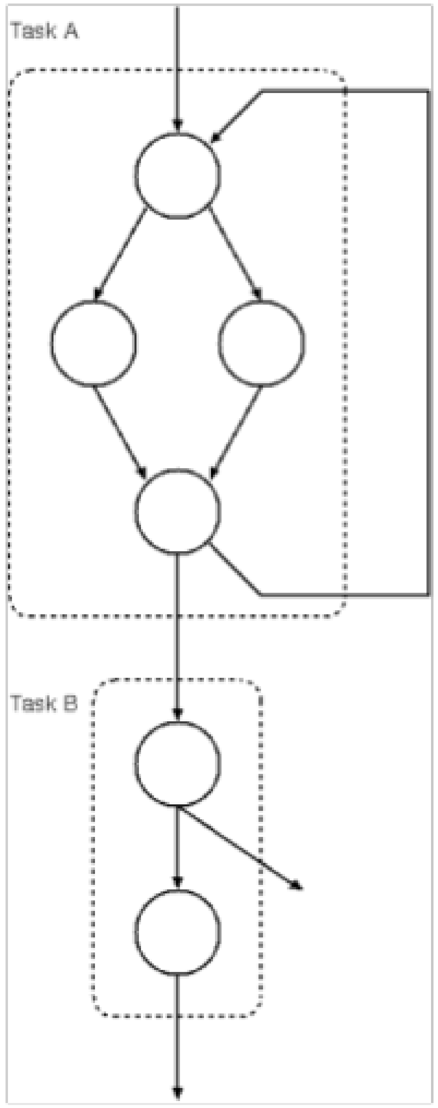
Figure 1: Splitting a large window of instructions into smaller windows
(i) A single large window (ii) A number of small windows

Multiscalar Model of Execution

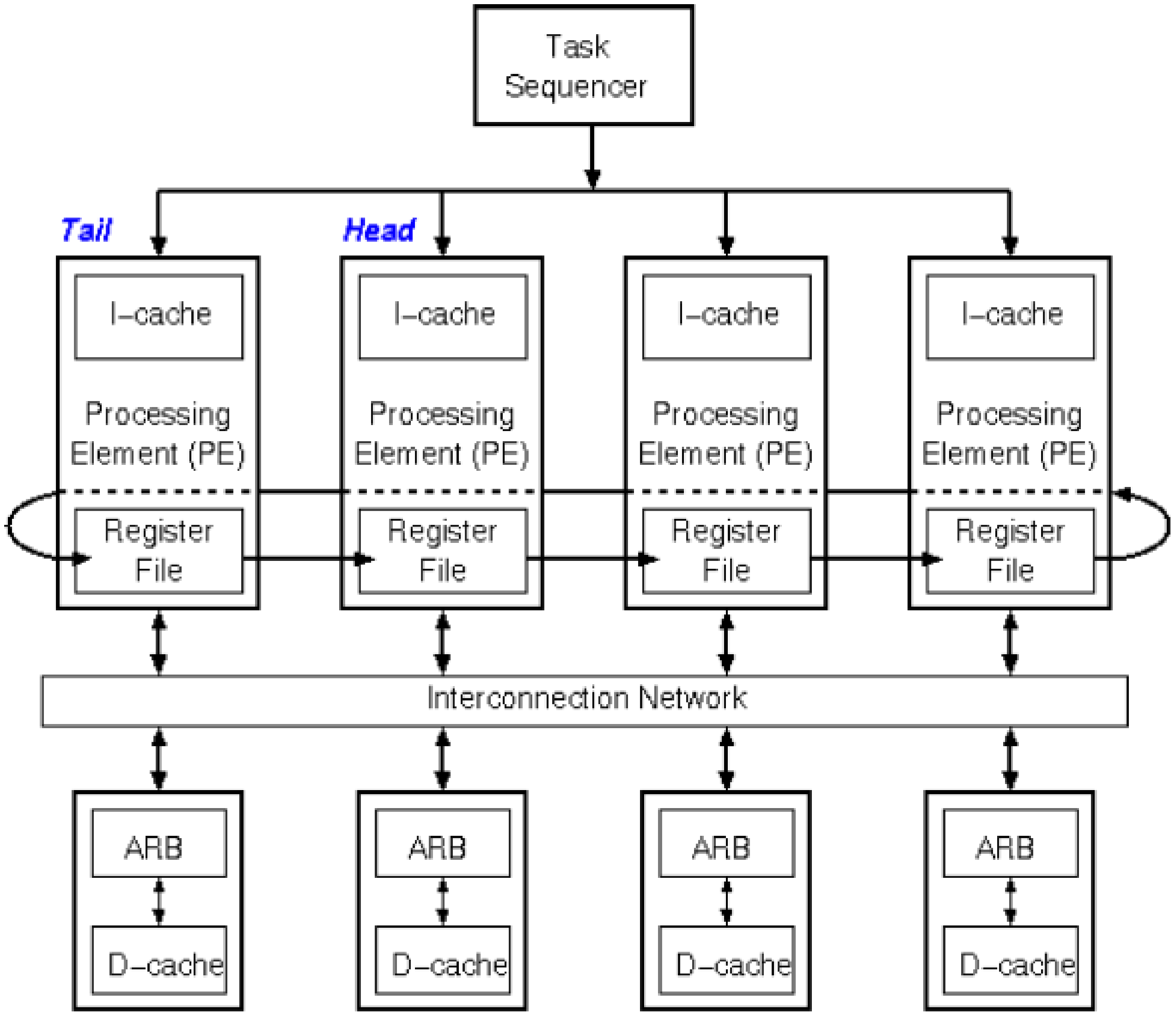

Superscalar
single centralized window
Multiscalar
multiple distributed windows
PC
task
task
task
task
PC
PC
PC
PC
dynamic instruction stream

Multiscalar Tasks

- A task is a subgraph of the control flow graph (CFG)
  - e.g., a basic block, multiple basic blocks, loop body, function
- Tasks are selected by compiler and conveyed to hardware
- Tasks are predicted and scheduled by processor
- Tasks may have data and/or control dependences



Multiscalar Processor



Multiscalar Compiler

- Task selection: partition CFG into tasks
  - Load balance across processors
  - Minimize inter-task data dependences
  - Minimize inter-task control dependences
    - By embedding hard-to-predict branches within tasks

- Convey task and communication information in the executable
  - Task headers
    - create_mask (1 bit per register)
      - Indicates all registers that are *possibly modified or created by the task (better: live-out of the task)*
      - *Don't forward instances received from prior tasks*
    - *PCs of successor tasks*
  - Release instructions: Release a register to be forwarded to a receiving task

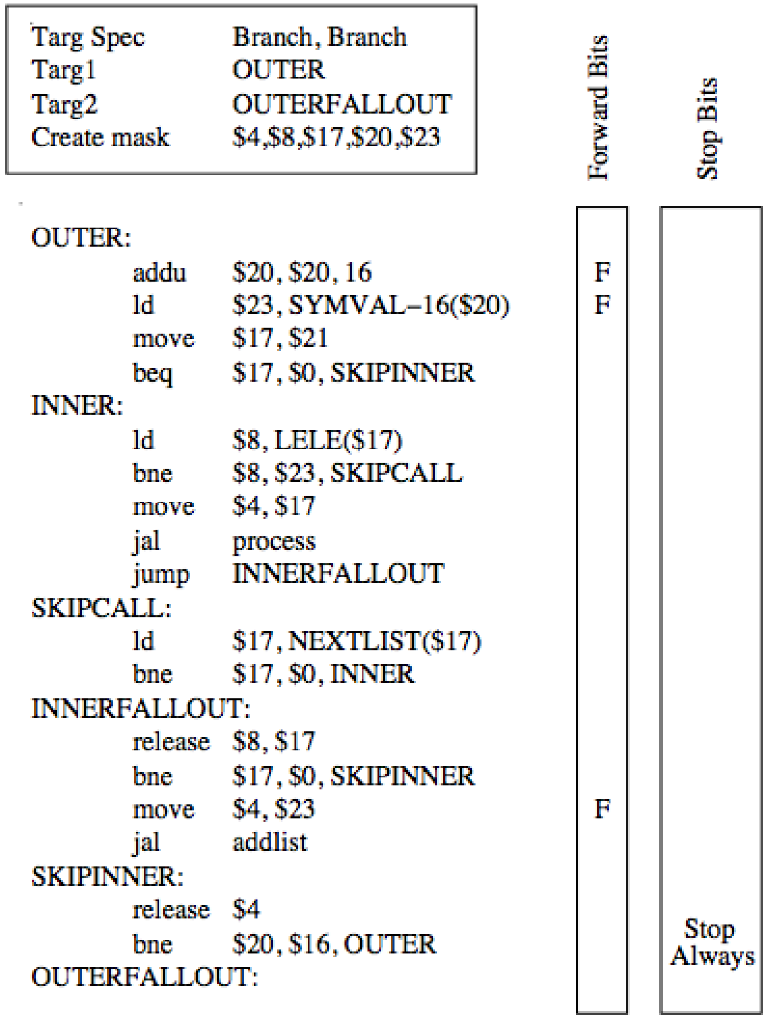
Multiscalar Program Example

```
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 in the list, add to the tail */
    if (!list) {
        addlist(symbol);
    }
}
```

Figure 3: An Example Code Segment.



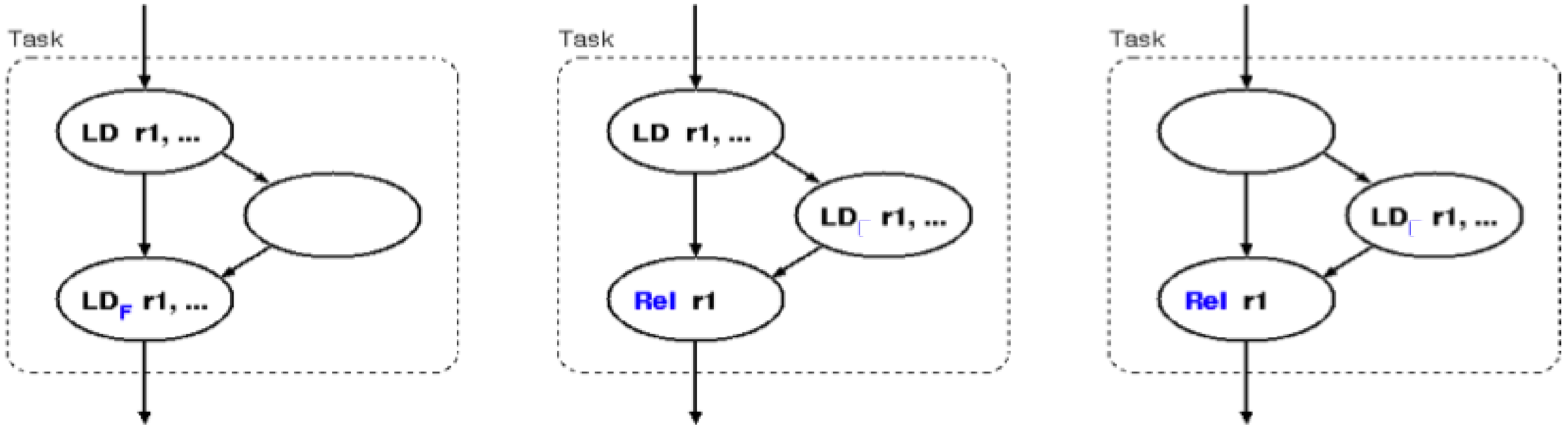
Targ Spec	Branch, Branch
Targ1	OUTER
Targ2	OUTERFALLOUT
Create mask	$4,$8,$17,$20,$23

```
                                              Forward Bits   Stop Bits
OUTER:
        addu     $20, $20, 16                      F
        ld       $23, SYMVAL-16($20)               F
        move     $17, $21
        beq      $17, $0, SKIPINNER
INNER:
        ld       $8, LELE($17)
        bne      $8, $23, SKIPCALL
        move     $4, $17
        jal      process
        jump     INNERFALLOUT
SKIPCALL:
        ld       $17, NEXTLIST($17)
        bne      $17, $0, INNER
INNERFALLOUT:
        release  $8, $17
        bne      $17, $0, SKIPINNER                F
        move     $4, $23
        jal      addlist
SKIPINNER:
        release  $4
        bne      $20, $16, OUTER                                 Stop
OUTERFALLOUT:                                                    Always
```

Figure 4: An Example of a Multiscalar Program.

Forwarding Registers Between Tasks

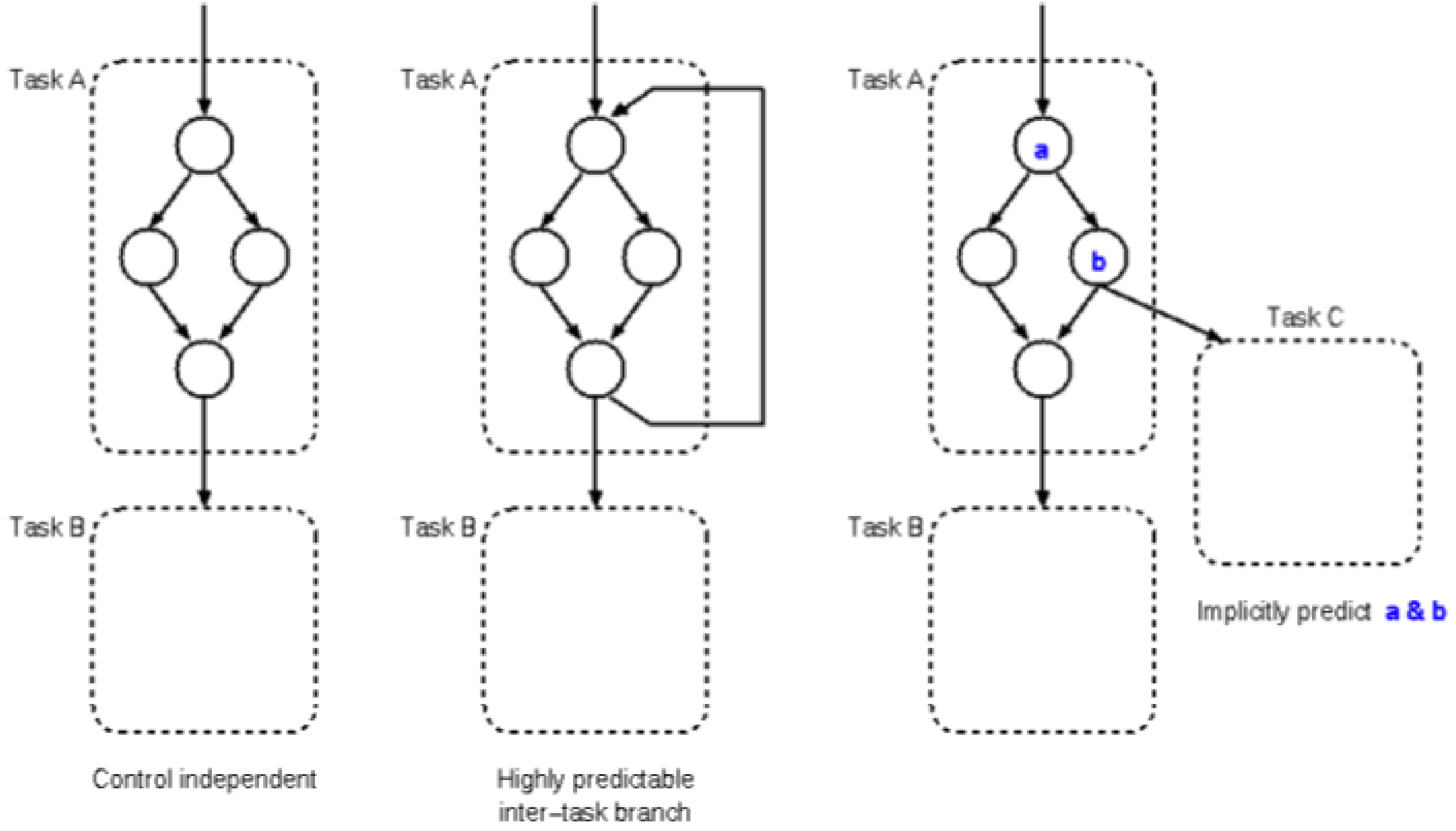
- Compiler must identify the last instance of write to a register within a task
  - Opcodes that write a register have additional forward bit, indicating the instance should be forwarded
  - Stop bits - indicate end of task
  - Release instruction
    - tells PE to forward the register value



F: forward bit
Rel: release

Task Sequencing

- Task prediction analogous to branch prediction
- Predict inter-task control flow



Handling Inter-Task Dependences

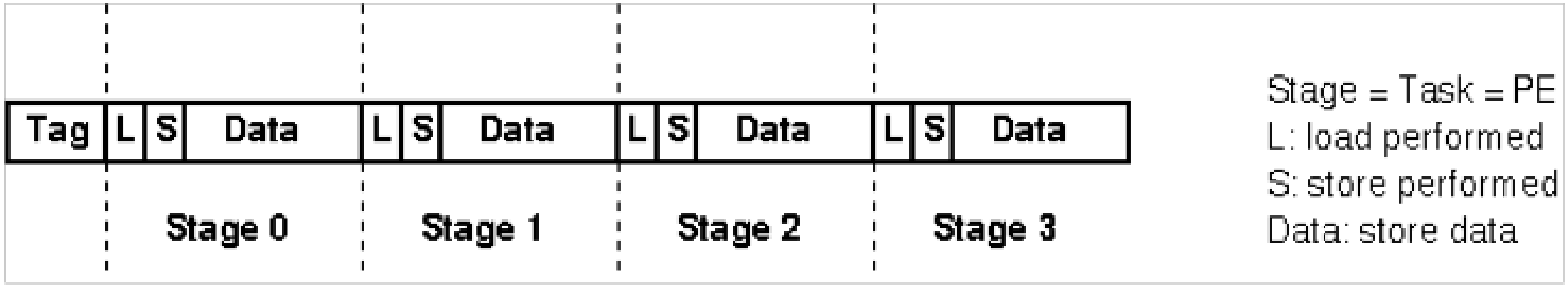
- Control dependences
  - Predict
  - Squash subsequent tasks on inter-task misprediction
    - Intra-task mispredictions do not need to cause flushing of later tasks

- Data dependences
  - Register file: mask bits and forwarding (stall until available)
  - Memory: address resolution buffer (speculative load, squash on violation)

Address Resolution Buffer

- Multiscalar issues loads to ARB/D-cache as soon as address is computed
- ARB is organized like a cache, maintaining state for all outstanding load/store addresses
- Franklin and Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," IEEE TC 1996.

- An ARB entry:



Address Resolution Buffer

- Loads
  - ARB miss: data comes from D-cache (no prior stores yet)
  - ARB hit: get most recent data to the load, which may be from D-cache, or nearest prior task with S=1
- Stores
  - ARB buffers speculative stores
  - If store from an older task finds a load from a younger task to the same address → misspeculation detected
  - When a task commits, *commit all of the task's stores into the D-cache*

Address Resolution Buffer

- Franklin and Sohi, “ARB: A hardware mechanism for dynamic reordering of memory references,” IEEE TC 1996.

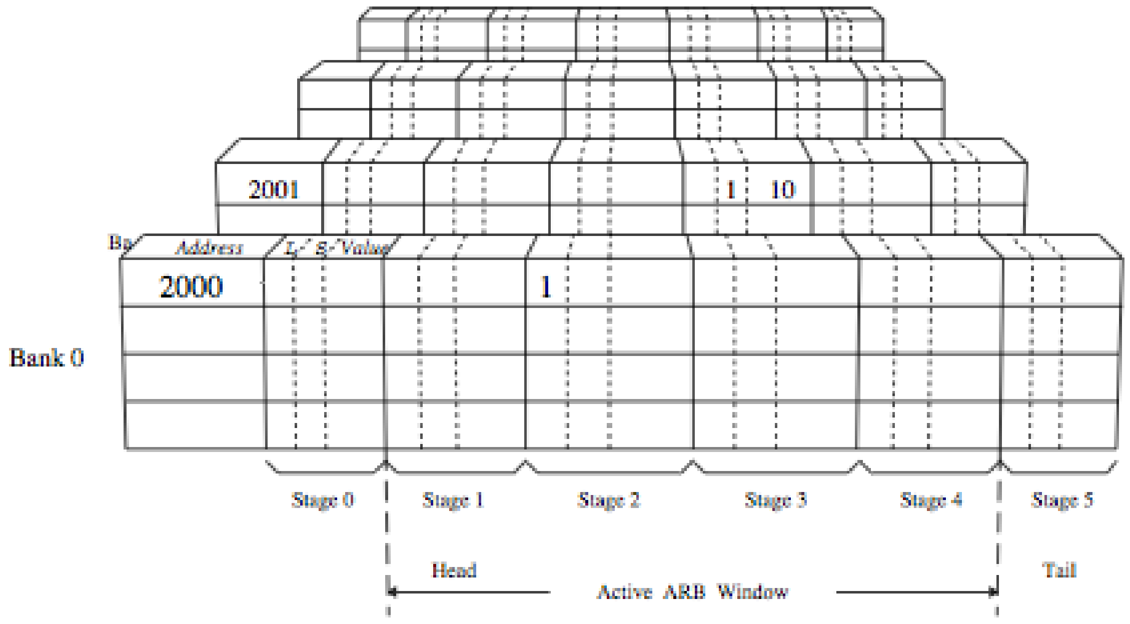


Figure 1: A 4-Way Interleaved, 6-stage ARB

Memory Dependence Prediction

- ARB performs memory renaming
- However, it does not perform dependence prediction
  - Can reduce intra-task dependency flushes by accurate memory dependence prediction

- Idea: Predict whether or not a load instruction will be dependent on a previous store (and predict which store). Delay the execution of the load if it is predicted to be dependent.

- Moshovos et al., "Dynamic Speculation and Synchronization of Data Dependences," ISCA 1997.
- Chrysos and Emer, "Memory Dependence Prediction using Store Sets," ISCA 1998.

Handling of Store-Load Dependencies

- A load's dependence status is not known until all previous store addresses are available.

- How does the processor detect dependence of a load instruction on a previous store?
  - Option 1: Wait until all previous stores committed (no need to check)
  - Option 2: Keep a list of pending stores in a store buffer and check whether load address matches a previous store address

- How does the processor engine treat the scheduling of a load instruction with respect to previous stores?
  - Option 1: Assume load independent of all previous stores
  - Option 2: Assume load dependent on all previous stores
  - Option 3: Predict the dependence of a load on an outstanding store

Memory Disambiguation

- Option 1: Assume load independent of all previous stores

  + Simple and can be common case: no delay for independent loads

  -- Requires recovery and re-execution of load and dependents on misprediction

- Option 2: Assume load dependent on all previous stores

  + No need for recovery

  -- Too conservative: delays independent loads unnecessarily

- Option 3: Predict the dependence of a load on an outstanding store

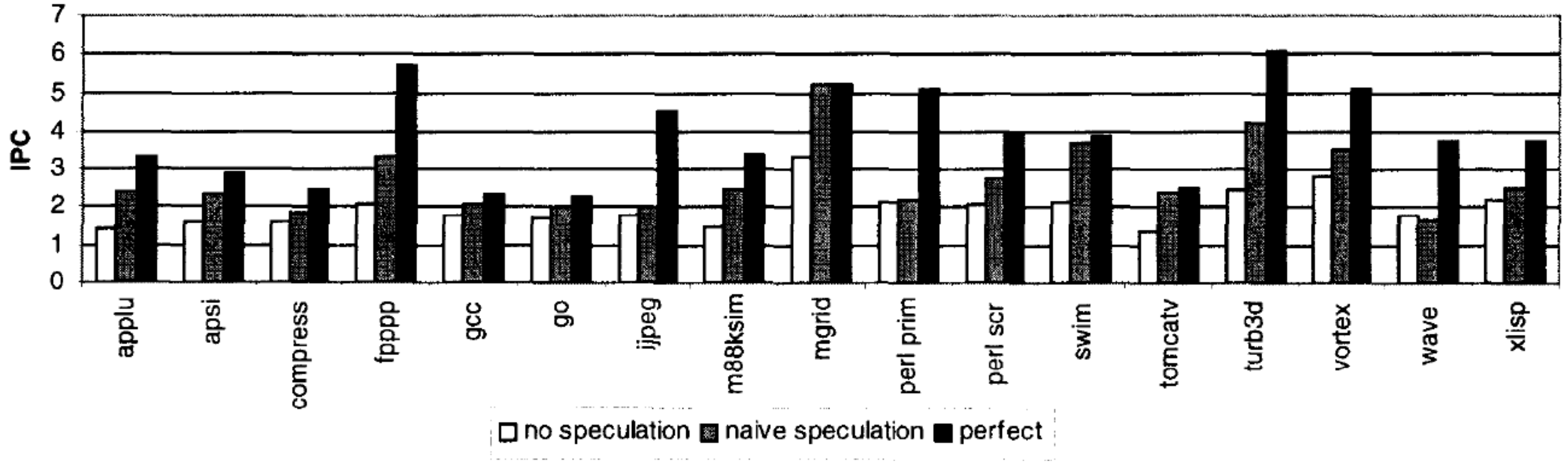
  + More accurate. Load store dependencies persist over time

  -- Still requires recovery/re-execution on misprediction

  - Alpha 21264 : Initially assume load independent, delay loads found to be dependent
  - Moshovos et al., "Dynamic speculation and synchronization of data dependences," ISCA 1997.
  - Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.

Memory Disambiguation

- Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.



- Predicting store-load dependencies important for performance
- Simple predictors (based on past history) can achieve most of the potential performance

Multiscalar Comparisons and Questions

- vs. superscalar, out-of-order?
- vs. multi-core?
- vs. CMP and SMT-based thread-level speculation mechanisms
  - What is different in multiscalar hardware?

- Scalability of fine-grained register communication
- Scalability of memory renaming and dependence speculation

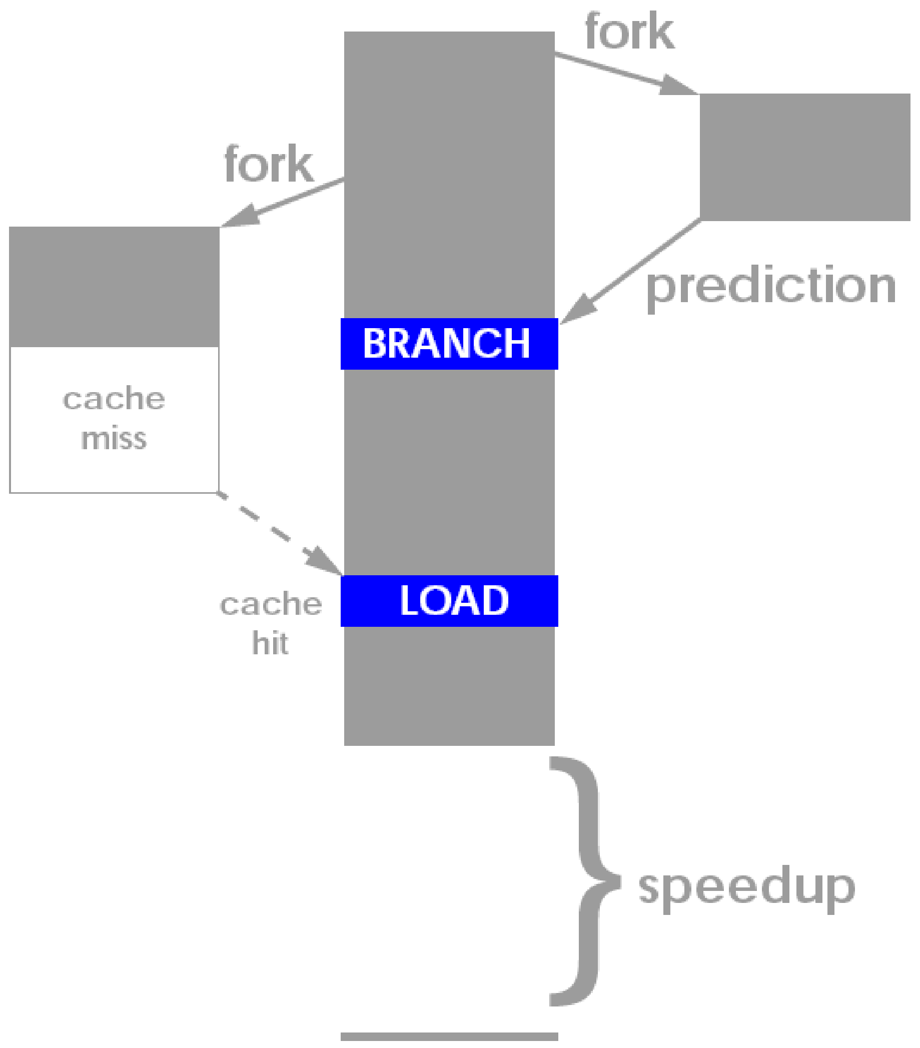
Helper Threading for Prefetching

- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses

- Speculative thread: Pre-executed program piece can be considered a “thread”

- Speculative thread can be executed
  - On a separate processor/core
  - On a separate hardware thread context
  - On the same thread context in idle cycles (during cache misses)

Generalized Thread-Based Pre-Execution



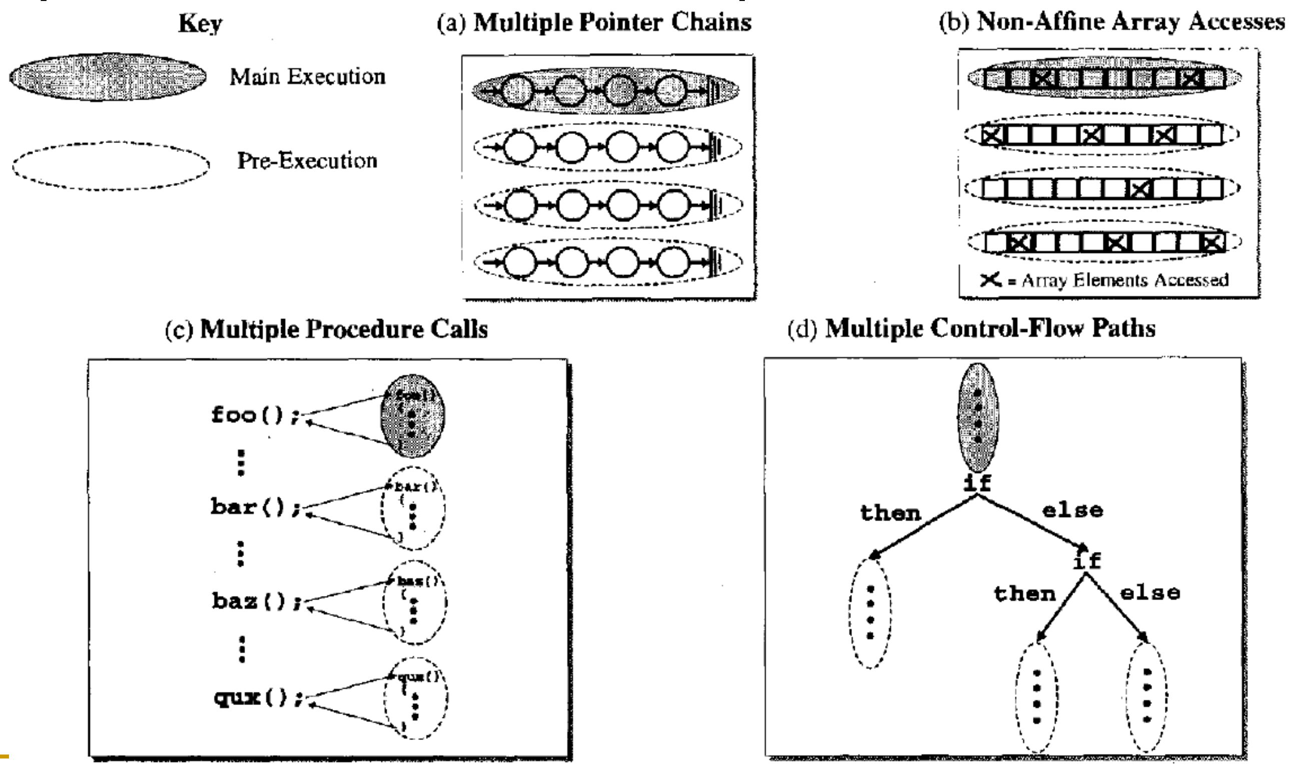
- Dubois and Song, "Assisted Execution," USC Tech Report 1998.
- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.
- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.

Thread-Based Pre-Execution Issues

- Where to execute the precomputation thread?
  1. Separate core (least contention with main thread)
  2. Separate thread context on the same core (more contention)
  3. Same core, same context
     - When the main thread is stalled
- When to spawn the precomputation thread?
  1. Insert spawn instructions well before the "problem" load
     - How far ahead?
       - Too early: prefetch might not be needed
       - Too late: prefetch might not be timely
  2. When the main thread is stalled
- When to terminate the precomputation thread?
  1. With pre-inserted CANCEL instructions
  2. Based on effectiveness/contention feedback

Thread-Based Pre-Execution Issues

- Read
  - Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," ISCA 2001.
  - Many issues in software-based pre-execution discussed



Key
Main Execution
Pre-Execution
(a) Multiple Pointer Chains
(b) Non-Affine Array Accesses
= Array Elements Accessed
(c) Multiple Procedure Calls
foo();
bar();
baz();
qux();
(d) Multiple Control-Flow Paths
if
then
else
if
then
else

An Example

(a) Original Code

```
register int i;
register arc_t *arcout;
for(; i<trips; ){
    // loop over 'trips' lists
    if (arcout[1].ident != FIXED) {
        ...
        first_of_sparse_list = arcout + 1;
    }
    ...
    arcin = (arc_t *)first_of_sparse_list
                    →tail→mark;
    // traverse the list starting with
    // the first node just assigned
    while (arcin) {
        tail = arcin→tail;
        ...
        arcin = (arc_t *)tail→mark;
    }
    i++, arcout+=3;
}
```

(b) Code with Pre-Execution

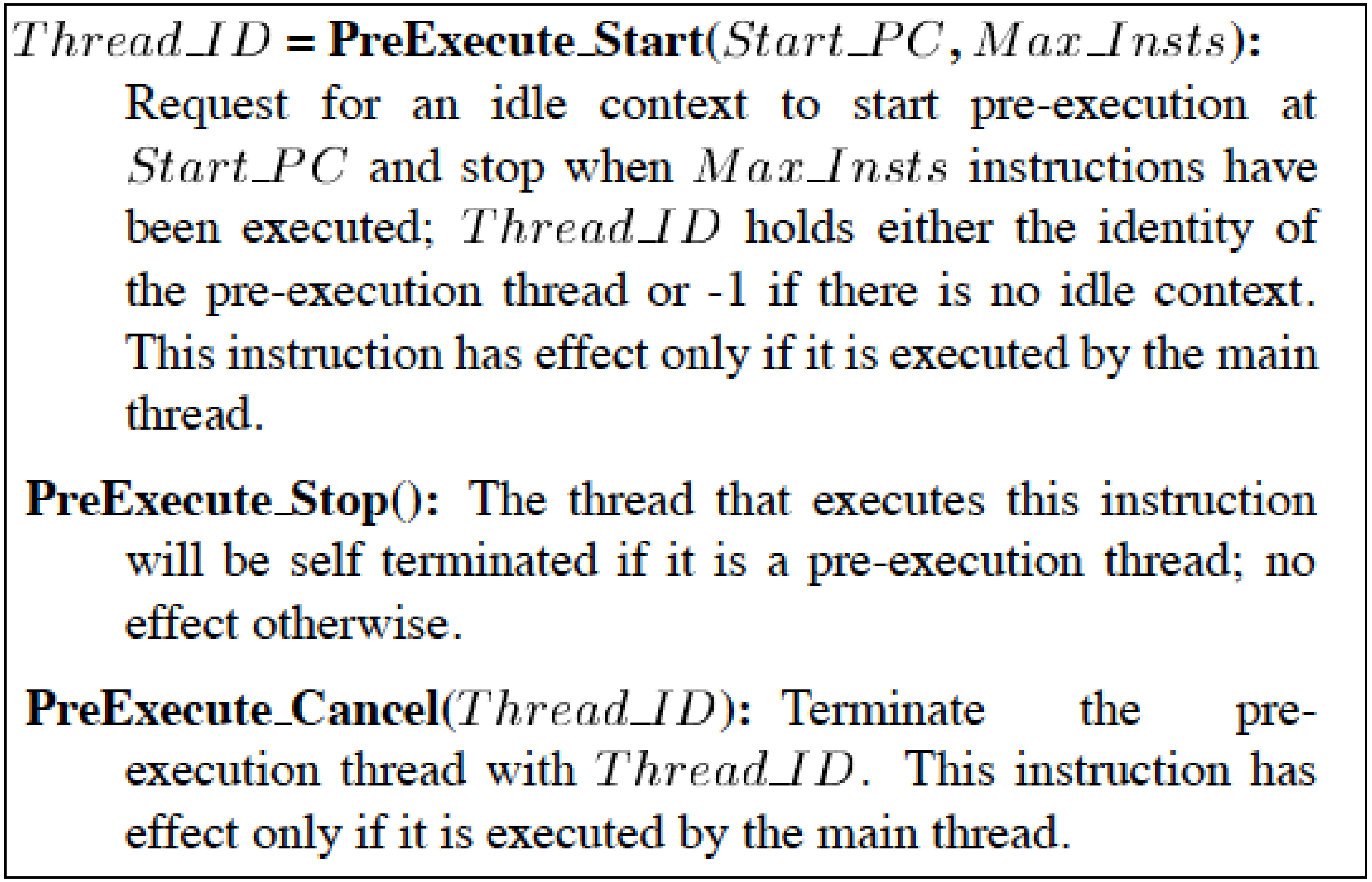
```
register int i;
register arc_t *arcout;
for(; i<trips; ){
    // loop over 'trips' lists
    if (arcout[1].ident != FIXED) {
        ...
        first_of_sparse_list = arcout + 1;
    }
    ...
    // invoke a pre-execution starting
    // at END_FOR
    PreExecute_Start(END_FOR);
    arcin = (arc_t *)first_of_sparse_list
            →tail→mark;
    // traverse the list starting with
    // the first node just assigned
    while (arcin) {
        tail = arcin→tail;
        ...
        arcin = (arc_t *)tail→mark;
    }
    // terminate this pre-execution after
    // prefetching the entire list
    PreExecute_Stop();
END_FOR:
    // the target address of the pre-
    // execution
    i++, arcout+=3;
}
// terminate this pre-execution if we
// have passed the end of the for-loop
PreExecute_Stop();
```

Figure 2. Abstract versions of an important loop nest in the Spec2000 benchmark `mcf`. Loads that incur many cache misses are underlined.

The Spec2000 benchmark `mcf` spends roughly half of its execution time in a nested loop which traverses a set of linked lists. An abstract version of this loop is shown in Figure 2(a), in which the for-loop iterates over the lists and the while-loop visits the elements of each list. As we observe from the figure, the first node of each list is assigned by dereferencing the pointer `first_of_sparse_list`, whose value is in fact determined by `arcout`, an induction variable of the for-loop. Therefore, even when we are still working on the current list, the first and the remaining nodes on the next list can be loaded speculatively by pre-executing the next iteration of the for-loop.

Figure 2(b) shows a version of the program with pre-execution code inserted (shown in boldface). **END_FOR** is simply a label to denote the place where `arcout` gets updated. The new instruction **PreExecute_Start(END_FOR)** initiates a pre-execution thread, say T, starting at the PC represented by **END_FOR**. Right after the pre-execution begins, T's registers that hold the values of `i` and `arcout` will be updated. Then `i`'s value is compared against `trips` to see if we have reached the end of the for-loop. If so, thread T will exit the for-loop and encounters a **PreExecute_Stop**(), which will terminate the pre-execution and free up T for future use. Otherwise, T will continue pre-executing the body of the for-loop, and hence compute the first node of the next list automatically. Finally, after traversing the entire list through the while-loop, the pre-execution will be terminated by another **PreExecute_Stop**(). Notice that any **PreExecute_Start**() instructions encountered during pre-execution are simply ignored as we do not allow nested pre-execution in order to keep our design simple. Similarly, **PreExecute_Stop**() instructions cannot terminate the main thread either.

Example ISA Extensions



$Thread\_ID$ = **PreExecute_Start**($Start\_PC$, Max_Insts)**:** Request for an idle context to start pre-execution at $Start\_PC$ and stop when Max_Insts instructions have been executed; $Thread\_ID$ holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction has effect only if it is executed by the main thread.

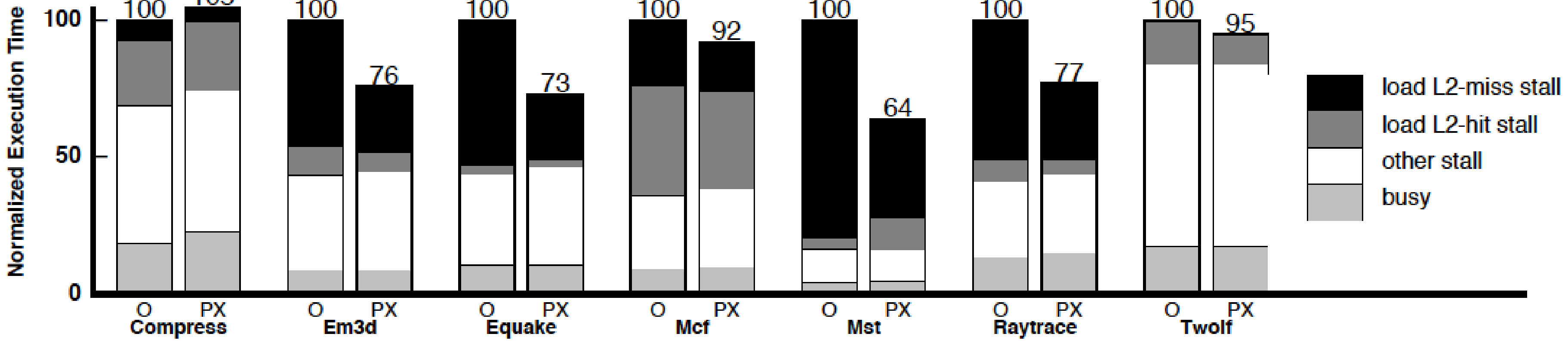
PreExecute_Stop(): The thread that executes this instruction will be self terminated if it is a pre-execution thread; no effect otherwise.

PreExecute_Cancel($Thread\_ID$)**:** Terminate the pre-execution thread with $Thread\_ID$. This instruction has effect only if it is executed by the main thread.

Figure 4. Proposed instruction set extensions to support pre-execution. (C syntax is used to improve readability.)

Results on an SMT Processor

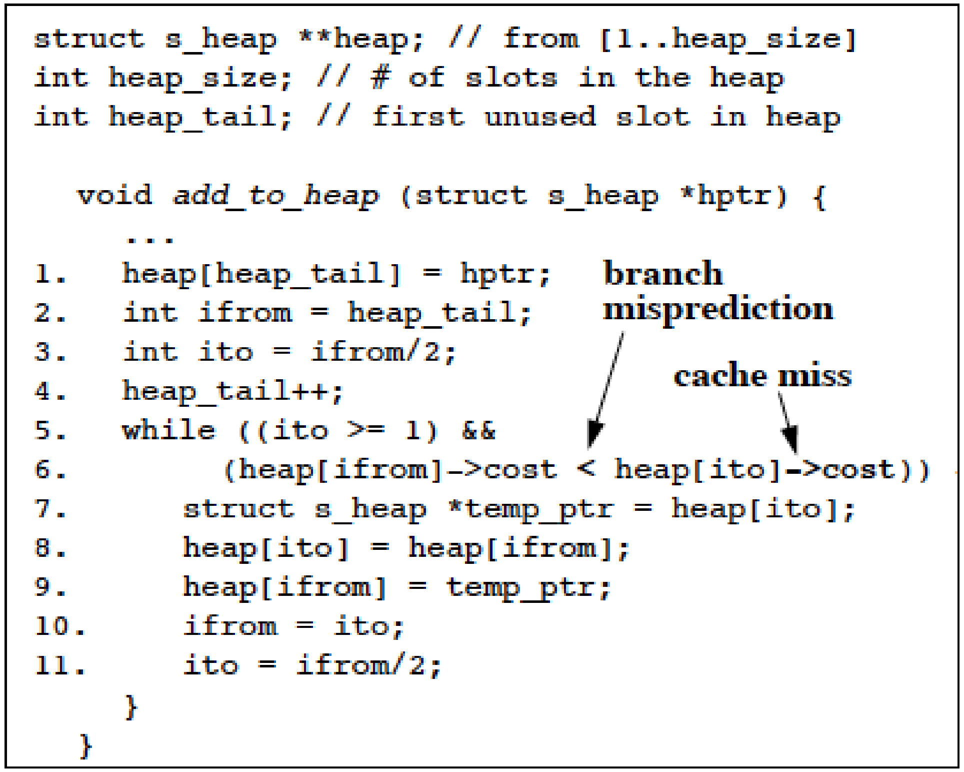
(a) Execution Time Normalized to the Original Case



Problem Instructions

- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.
- Zilles and Sohi, "Understanding the backward slices of performance degrading instructions," ISCA 2000.

Figure 2. ***Example problem instructions from heap insertion routine in vpr.***



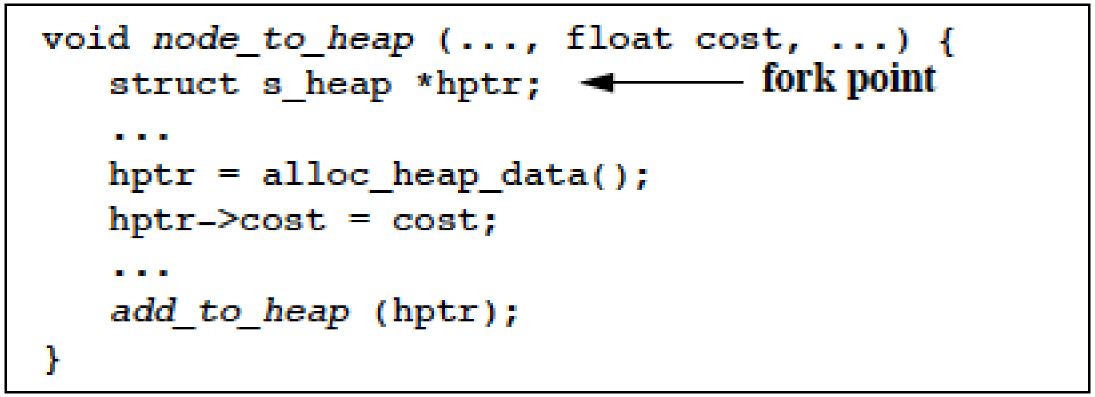
```
struct s_heap **heap; // from [1..heap_size]
int heap_size; // # of slots in the heap
int heap_tail; // first unused slot in heap

  void add_to_heap (struct s_heap *hptr) {
    ...
    heap[heap_tail] = hptr;
    int ifrom = heap_tail;
    int ito = ifrom/2;
    heap_tail++;
    while ((ito >= 1) &&
           (heap[ifrom]->cost < heap[ito]->cost))
       struct s_heap *temp_ptr = heap[ito];
       heap[ito] = heap[ifrom];
       heap[ifrom] = temp_ptr;
       ifrom = ito;
       ito = ifrom/2;
    }
  }
```

branch misprediction
cache miss

Fork Point for Prefetching Thread

Figure 3. *The* **`node_to_heap`** *function, which serves as the fork point for the slice that covers* **`add_to_heap`**.



```
void node_to_heap (..., float cost, ...) {
   struct s_heap *hptr;
   ...
   hptr = alloc_heap_data();
   hptr->cost = cost;
   ...
   add_to_heap (hptr);
}
```

Pre-execution Slice Construction

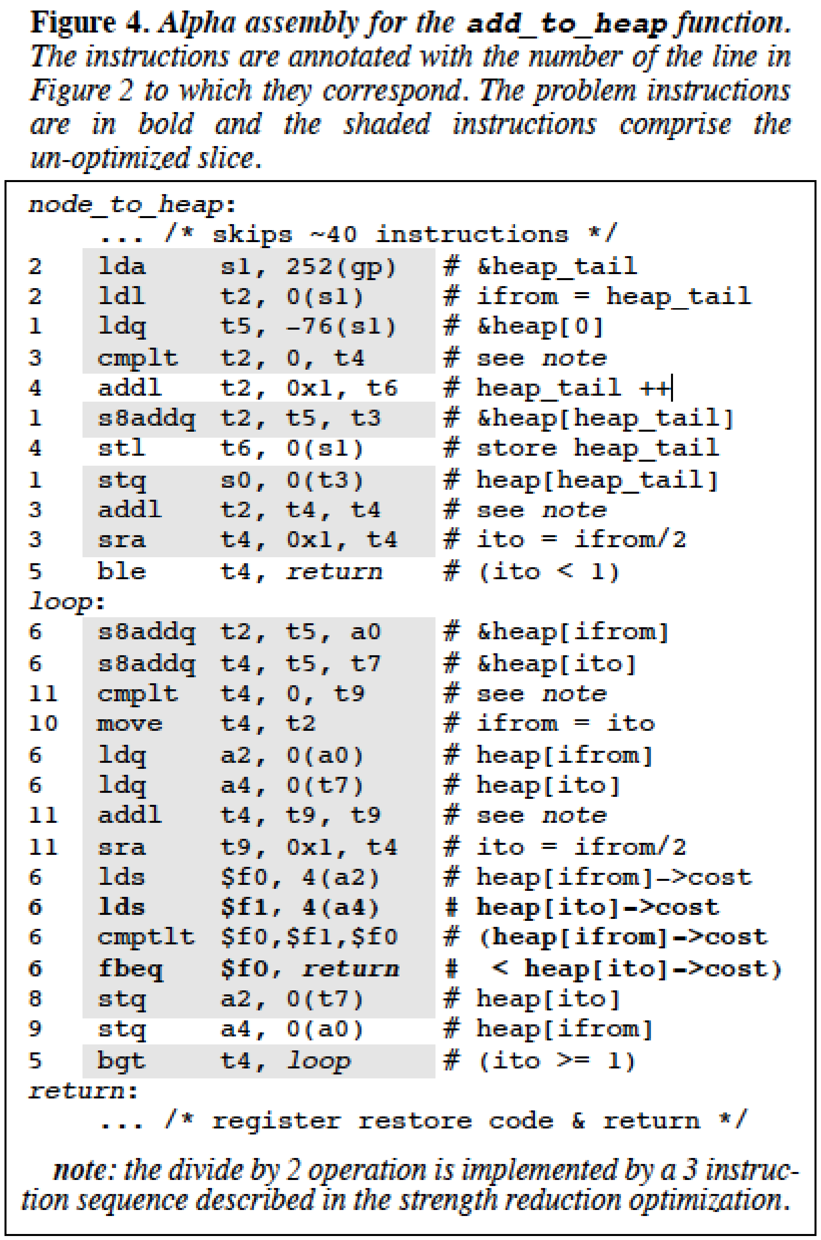


Figure 4. *Alpha assembly for the* **`add_to_heap`** *function. The instructions are annotated with the number of the line in Figure 2 to which they correspond. The problem instructions are in bold and the shaded instructions comprise the un-optimized slice.*

```
node_to_heap:
     ... /* skips ~40 instructions */
2    lda     s1, 252(gp)    # &heap_tail
2    ldl     t2, 0(s1)      # ifrom = heap_tail
1    ldq     t5, -76(s1)    # &heap[0]
3    cmplt   t2, 0, t4      # see note
4    addl    t2, 0x1, t6    # heap_tail ++
1    s8addq  t2, t5, t3     # &heap[heap_tail]
4    stl     t6, 0(s1)      # store heap_tail
1    stq     s0, 0(t3)      # heap[heap_tail]
3    addl    t2, t4, t4     # see note
3    sra     t4, 0x1, t4    # ito = ifrom/2
5    ble     t4, return     # (ito < 1)
loop:
6    s8addq  t2, t5, a0     # &heap[ifrom]
6    s8addq  t4, t5, t7     # &heap[ito]
11   cmplt   t4, 0, t9      # see note
10   move    t4, t2         # ifrom = ito
6    ldq     a2, 0(a0)      # heap[ifrom]
6    ldq     a4, 0(t7)      # heap[ito]
11   addl    t4, t9, t9     # see note
11   sra     t9, 0x1, t4    # ito = ifrom/2
6    lds     $f0, 4(a2)     # heap[ifrom]->cost
6    lds     $f1, 4(a4)     # heap[ito]->cost
6    cmptlt  $f0,$f1,$f0    # (heap[ifrom]->cost
6    fbeq    $f0, return    #  < heap[ito]->cost)
8    stq     a2, 0(t7)      # heap[ito]
9    stq     a4, 0(a0)      # heap[ifrom]
5    bgt     t4, loop       # (ito >= 1)
return:
     ... /* register restore code & return */
```

note: the divide by 2 operation is implemented by a 3 instruction sequence described in the strength reduction optimization.

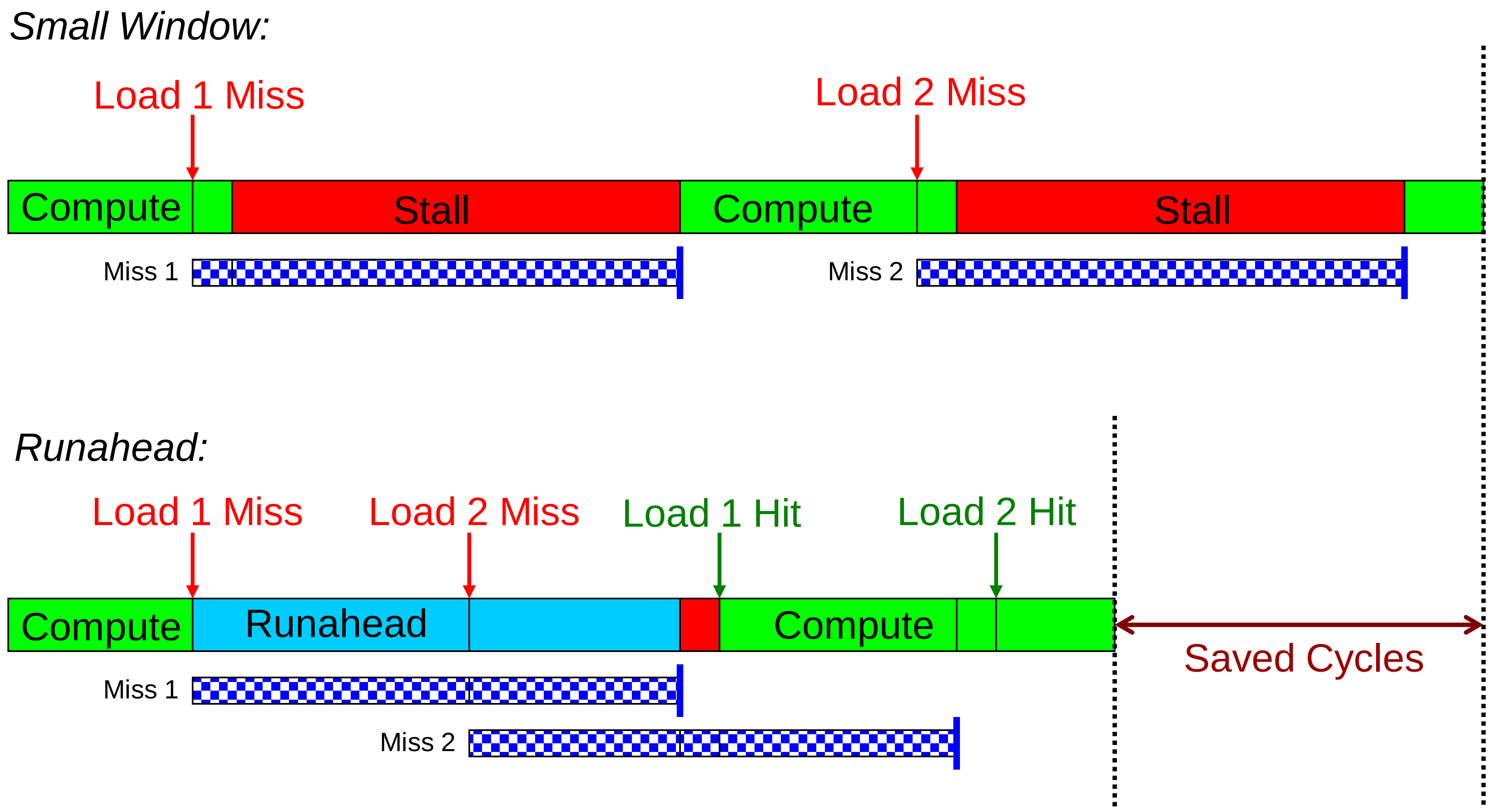
Review: Runahead Execution

- A simple pre-execution method for prefetching purposes

- When the oldest instruction is a long-latency cache miss:
  - Checkpoint architectural state and enter runahead mode
- In runahead mode:
  - Speculatively pre-execute instructions
  - The purpose of pre-execution is to generate prefetches
  - L2-miss dependent instructions are marked INV and dropped
- Runahead mode ends when the original miss returns
  - Checkpoint is restored and normal execution resumes

- Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.

Review: Runahead Execution (Mutlu et al., HPCA 2003)



Slipstream Processors

- Goal: use multiple hardware contexts to speed up single thread execution (implicitly parallelize the program)
- Idea: Divide program execution into two threads:
  - Advanced thread executes a reduced instruction stream, speculatively
  - Redundant thread uses results, prefetches, predictions generated by advanced thread and ensures correctness

- Benefit: Execution time of the overall program reduces
- Core idea is similar to many thread-level speculation approaches, except with a reduced instruction stream

- Sundaramoorthy et al., "Slipstream Processors: Improving both Performance and Fault Tolerance," ASPLOS 2000.

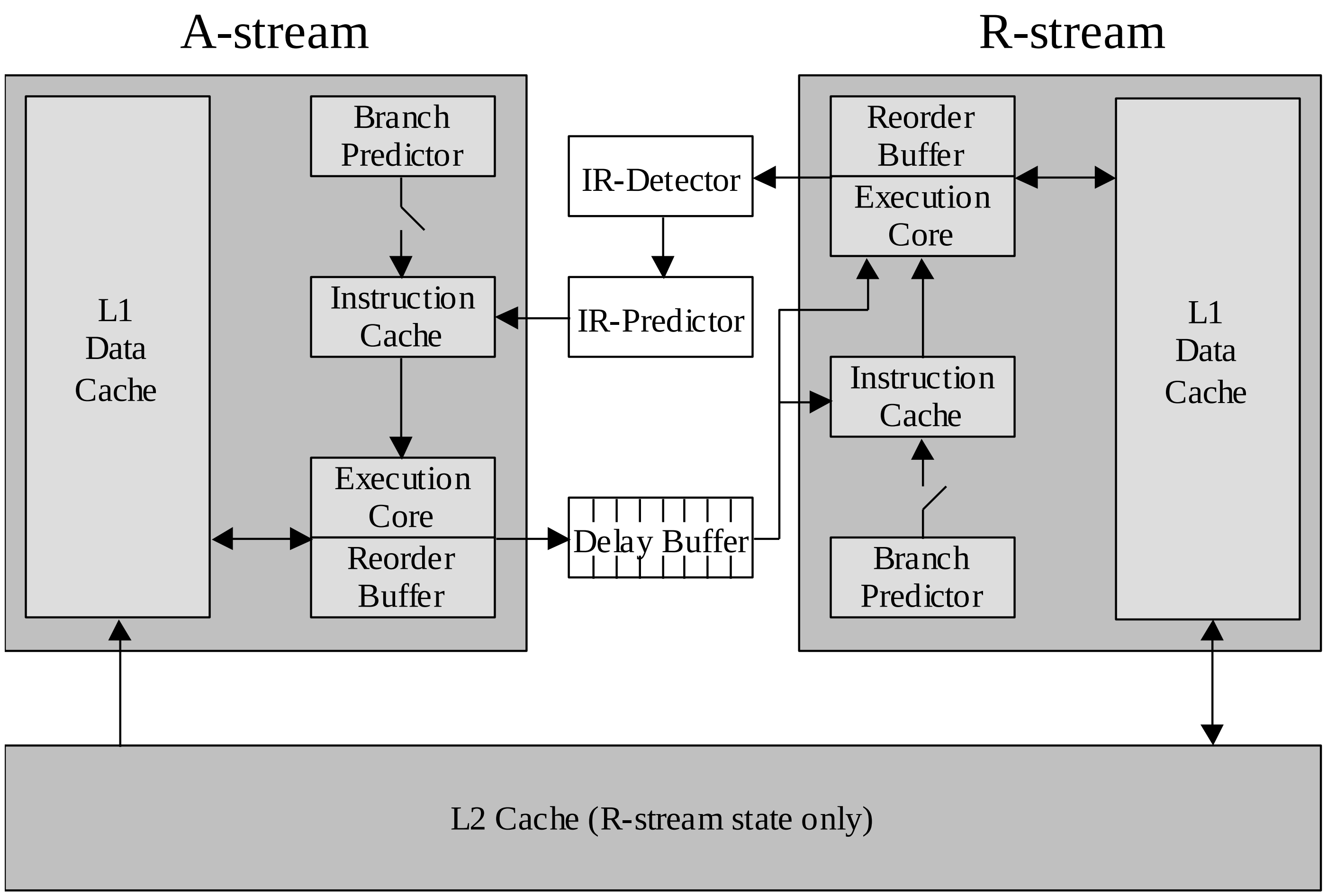
Slipstreaming

- "At speeds in excess of 190 m.p.h., high air pressure forms at the front of a race car and a partial vacuum forms behind it. This creates drag and limits the car's top speed.
- A second car can position itself close behind the first (a process called *slipstreaming* or *drafting*). This fills the vacuum behind the lead car, reducing its drag. And the trailing car now has less wind resistance in front (and by some accounts, the vacuum behind the lead car actually helps pull the trailing car).
- As a result, both cars speed up by several m.p.h.: the two combined go faster than either can alone."

Slipstream Processors

- Detect and remove ineffectual instructions; run a shortened "effectual" version of the program (Advanced or A-stream) in one thread context
- Ensure correctness by running a complete version of the program (Redundant or R-stream) in another thread context
- Shortened A-stream runs fast; R-stream consumes near-perfect control and data flow outcomes from A-stream and finishes close behind
- Two streams together lead to faster execution (by helping each other) than a single one alone

Slipstream Idea and Possible Hardware



Instruction Removal in Slipstream

- IR detector
  - Monitors retired R-stream instructions
  - Detects ineffectual instructions and conveys them to the IR predictor
  - Ineffectual instruction examples:
    - dynamic instructions that repeatedly and predictably have no observable effect (e.g., unreferenced writes, non-modifying writes)
    - dynamic branches whose outcomes are consistently predicted correctly.
- IR predictor
  - Removes an instruction from A-stream after repeated indications from the IR detector
- A stream skips ineffectual instructions, executes everything else and inserts their results into delay buffer
- R stream executes all instructions but uses results from the delay buffer as predictions

What if A-stream Deviates from Correct Execution?

- Why
  - A-stream deviates due to incorrect removal or stale data access in L1 data cache
- How to detect it?
  - Branch or value misprediction happens in R-stream (known as an IR misprediction)
- How to recover?
  - Restore A-stream register state: copy values from R-stream registers using delay buffer or shared-memory exception handler
  - Restore A-stream memory state: invalidate A-stream L1 data cache (or speculatively written blocks by A-stream)

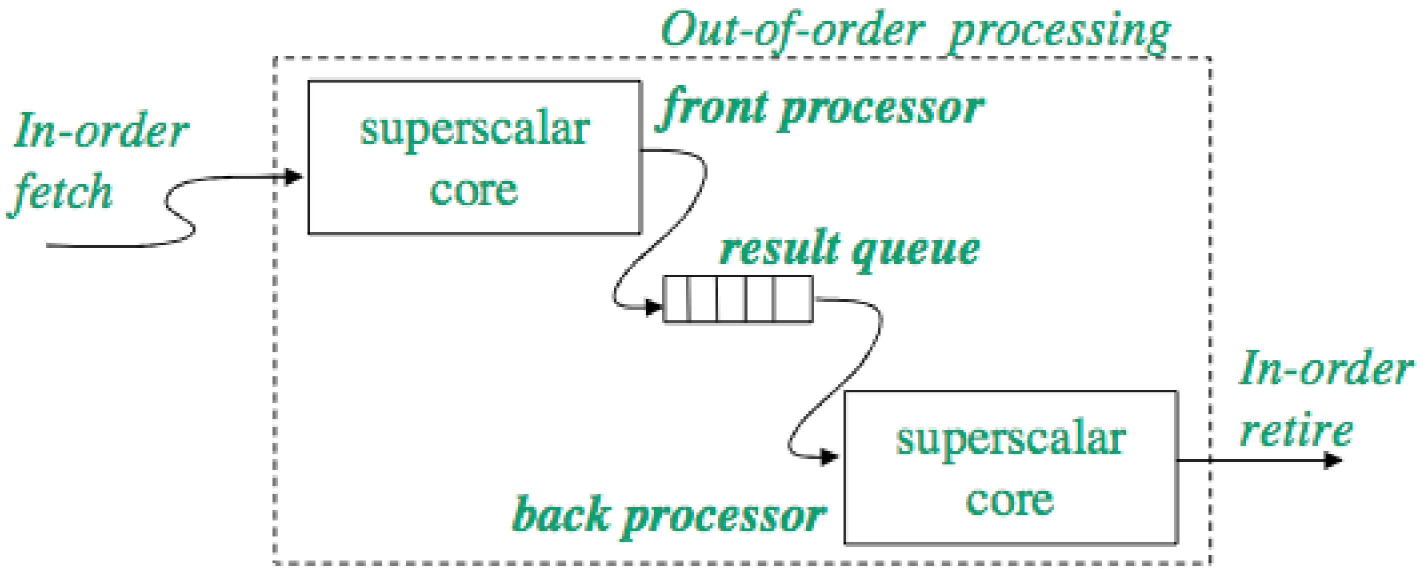
Slipstream Questions

- How to construct the advanced thread
  - Original proposal:
    - Dynamically eliminate redundant instructions (silent stores, dynamically dead instructions)
    - Dynamically eliminate easy-to-predict branches
  - Other ways:
    - Dynamically ignore long-latency stalls
    - Static based on profiling

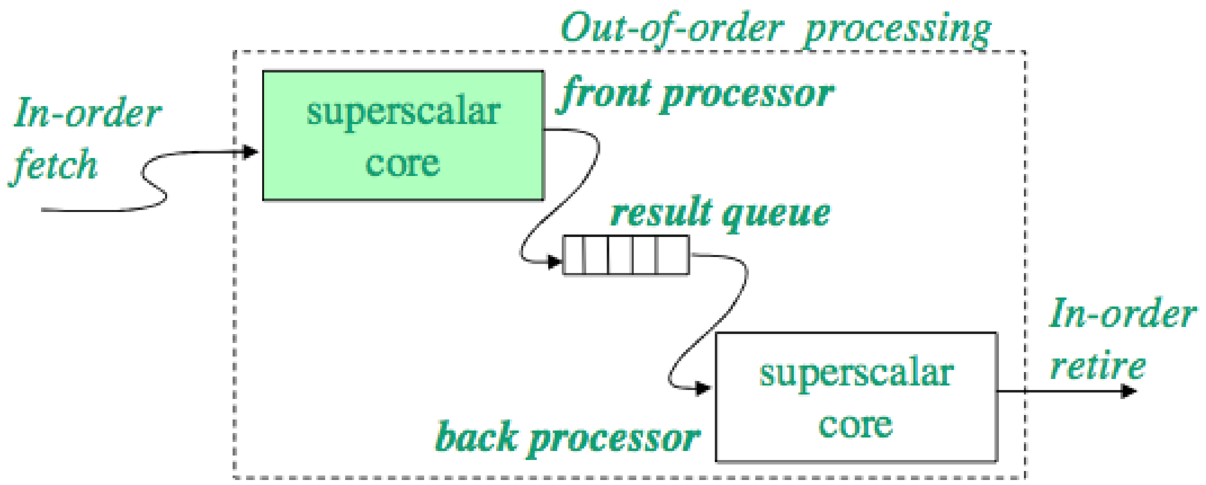
- How to speed up the redundant thread
  - Original proposal: Reuse instruction results (control and data flow outcomes from the A-stream)
  - Other ways: Only use branch results and prefetched data as predictions

Dual Core Execution

- Idea: One thread context speculatively runs ahead on load misses and prefetches data for another thread context
- Zhou, "Dual-Core Execution: Building a Highly Scalable Single- Thread Instruction Window," PACT 2005.



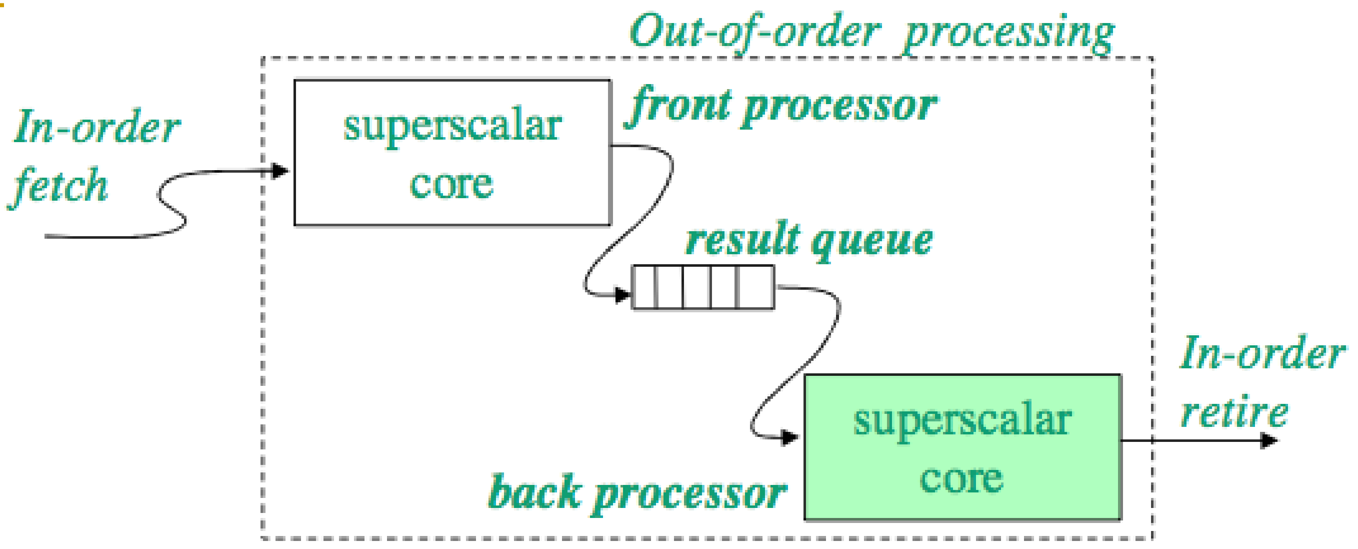
Dual Core Execution: Front Processor



- The front processor runs faster by invalidating long-latency cache-missing loads, same as runahead execution
  - Load misses and their dependents are invalidated
  - Branch mispredictions dependent on cache misses cannot be resolved

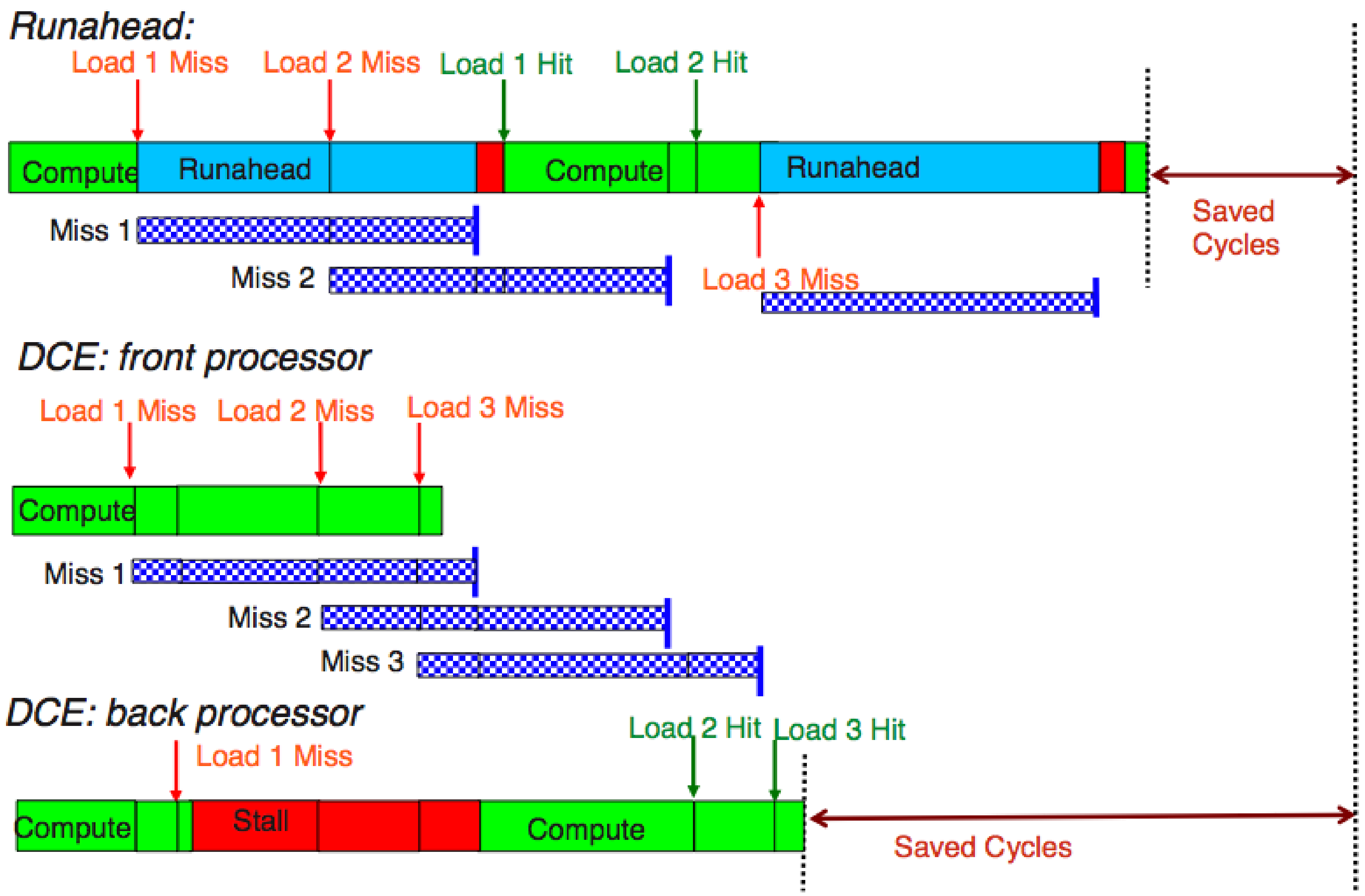
- Highly accurate execution as independent operations are not affected
  - Accurate prefetches to warm up caches
  - Correctly resolved miss-independent branch mispredictions

Dual Core Execution: Back Processor



- Re-execution ensures correctness and provides precise program state
    - Resolve branch mispredictions dependent on long-latency cache misses
- Back processor makes faster progress with help from the front processor
    - Highly accurate instruction stream
    - Warmed up data caches

Dual Core Execution



DCE Microarchitecture

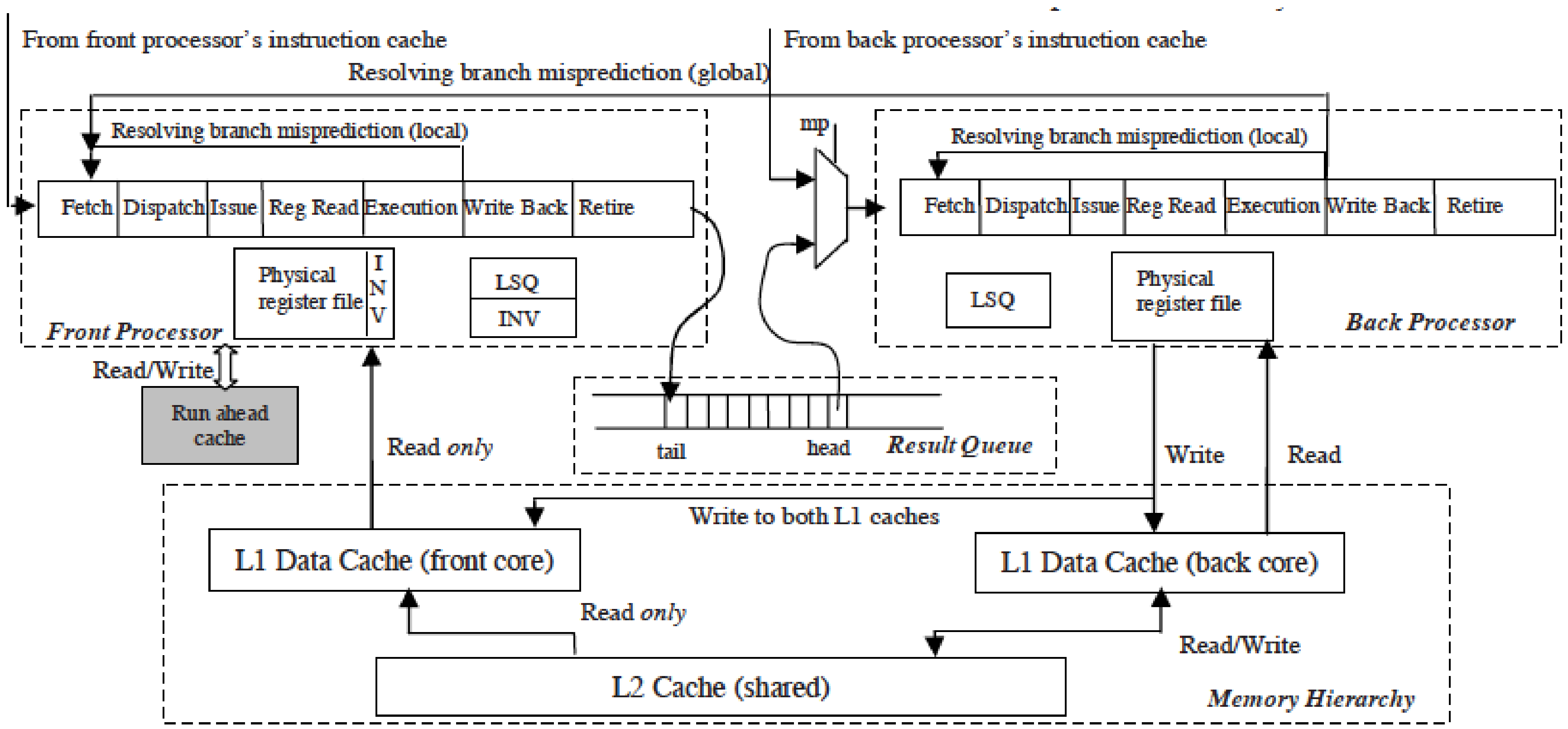


Figure 3. The design of DCE architecture.

Dual Core Execution vs. Slipstream

- Dual-core execution does not
  - remove dead instructions
  - reuse instruction register results
  - uses the "leading" hardware context solely for prefetching and branch prediction

+ Easier to implement, smaller hardware cost and complexity

+ Tolerates memory latencies better with the use of runahead execution in the front processor

- "Leading thread" cannot run ahead as much as in slipstream when there are no cache misses

- Not reusing results in the "trailing thread" can reduce overall performance benefit

Some Results

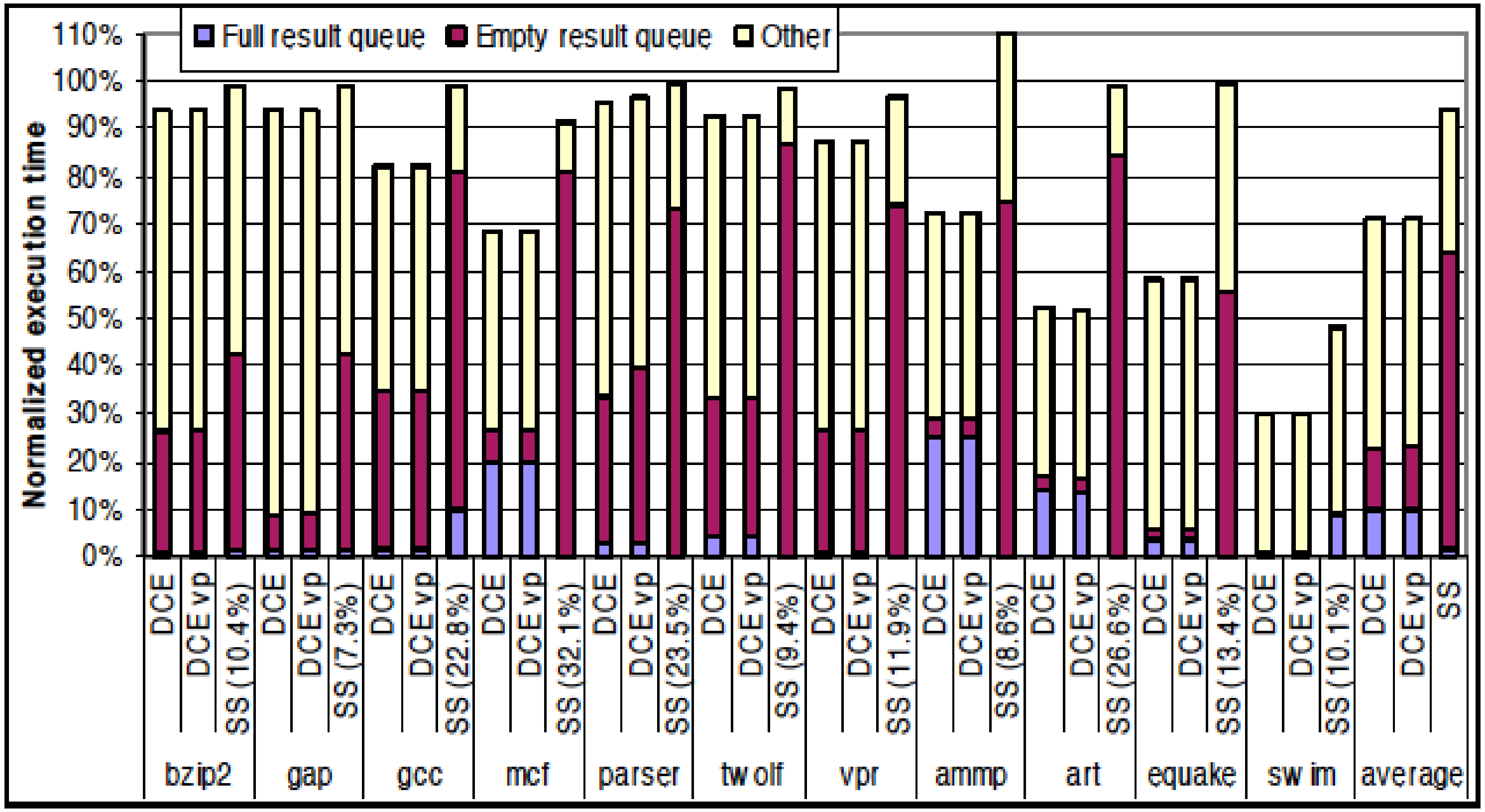


Figure 6. Normalized execution time of DCE, DCE with value prediction (DCE vp), and slipstreaming processors (SS).

Speculation to Improve Parallel Programs

Computer Architecture: Speculation (in Parallel Machines)

Prof. Onur Mutlu
Carnegie Mellon University

Speculation to Improve Parallel Programs

Referenced Readings

- Herlihy and Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," ISCA 1993.
- Rajwar and Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," MICRO 2001.
- Martinez and Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," ASPLOS 2002.
- Rajwar and Goodman, "Transactional lock-free execution of lock-based programs," ASPLOS 2002.
- Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009, IEEE Micro Top Picks 2010.
- Jensen et al., "A New Approach to Exclusive Data Access in Shared Memory Multiprocessors," LLNL Tech Report 1987.
- Dice et al., "Early Experience with a Commercial Hardware Transactional Memory Implementation," ASPLOS 2009.
- Wang et al., "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories," PACT 2012.
- Jacobi et al., "Transactional Memory Architecture and Implementation for IBM System Z," MICRO 2012.
- Chung et al., "ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory," MICRO 2010.

Speculation to Improve Parallel Programs

- Goal: reduce the impact of serializing bottlenecks
  - Improve performance
  - Improve programming ease

- Examples
  - Herlihy and Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," ISCA 1993.
  - Rajwar and Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," MICRO 2001.
  - Martinez and Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," ASPLOS 2002.
  - Rajwar and Goodman, "Transactional lock-free execution of lock-based programs," ASPLOS 2002.

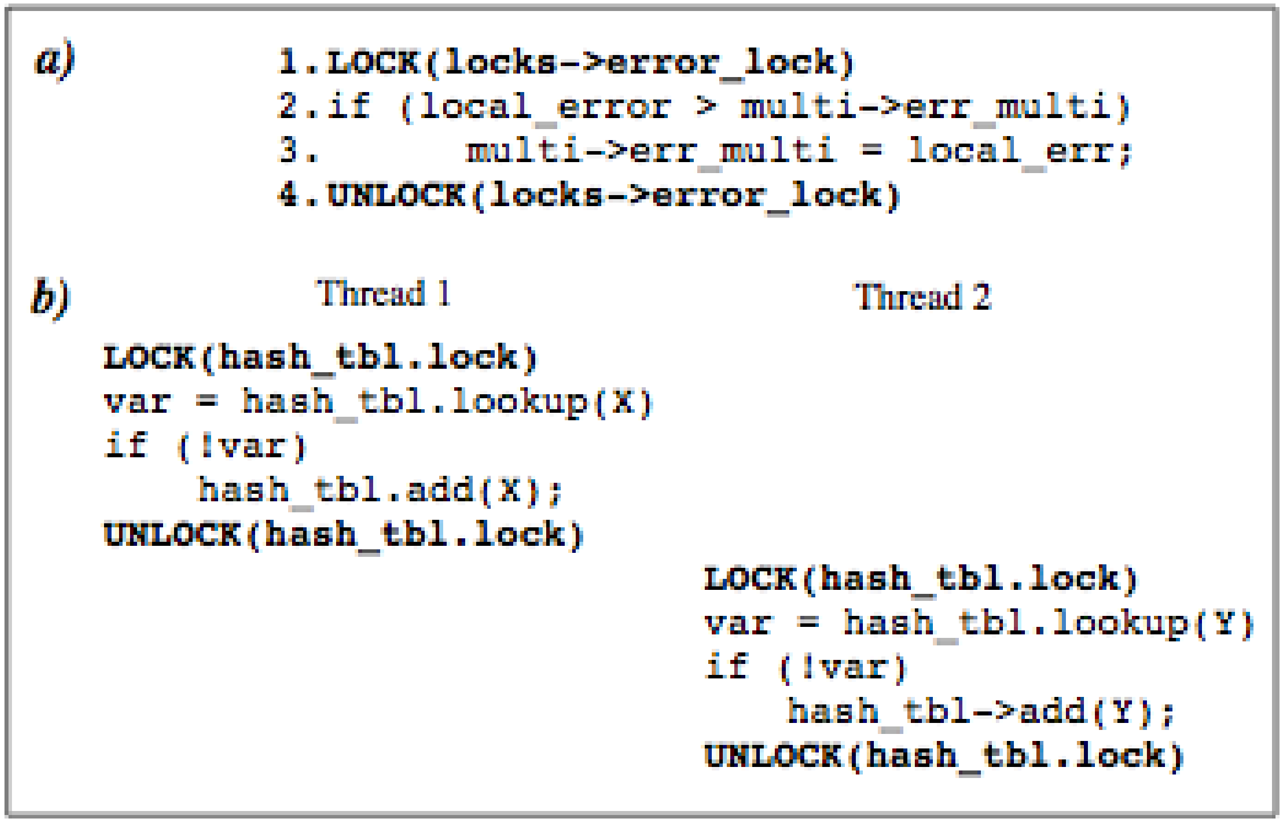
Speculative Lock Elision

- Many programs use locks for synchronization
- Many locks are not necessary
  - Stores occur infrequently during execution
  - Updates can occur to disjoint parts of the data structure

- Idea:
  - Speculatively assume lock is not necessary and execute critical section without acquiring the lock
  - Check for conflicts within the critical section
  - Roll back if assumption is incorrect

- Rajwar and Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," MICRO 2001.

Dynamically Unnecessary Synchronization



a)

```
1.LOCK(locks->error_lock)
2.if (local_error > multi->err_multi)
3.        multi->err_multi = local_err;
4.UNLOCK(locks->error_lock)
```

b) Thread 1 / Thread 2

```
                Thread 1                              Thread 2
LOCK(hash_tbl.lock)
var = hash_tbl.lookup(X)
if (!var)
    hash_tbl.add(X);
UNLOCK(hash_tbl.lock)
                                          LOCK(hash_tbl.lock)
                                          var = hash_tbl.lookup(Y)
                                          if (!var)
                                              hash_tbl->add(Y);
                                          UNLOCK(hash_tbl.lock)
```

Figure 1. ***Two examples of potential parallelism masked by dynamically unnecessary synchronization.***

Speculative Lock Elision: Issues

- Either the entire critical section is committed or none of it

- How to detect the lock
- How to keep track of dependencies and conflicts in a critical section
  - Read set and write set
- How to buffer speculative state
- How to check if “atomicity” is violated
  - Dependence violations with another thread
- How to support commit and rollback

Maintaining Atomicity

- If atomicity is maintained, all locks can be removed
- Conditions for atomicity:
  - Data read is not modified by another thread until critical section is complete
  - Data written is not accessed by another thread until critical section is complete

- If we know the beginning and end of a critical section, we can monitor the memory addresses read or written to by the critical section and check for conflicts
  - Using the underlying coherence mechanism

SLE Implementation

- Checkpoint register state before entering SLE mode

- In SLE mode:
  - Store: Buffer the update in the write buffer (do not make visible to other processors), request exclusive access
  - Store/Load: Set "access" bit for block in the cache
  - Trigger misspeculation on some coherence actions
    - If external invalidation to a block with "access" bit set
    - If exclusive access to request to a block with "access" bit set
  - If not enough buffering space, trigger misspeculation

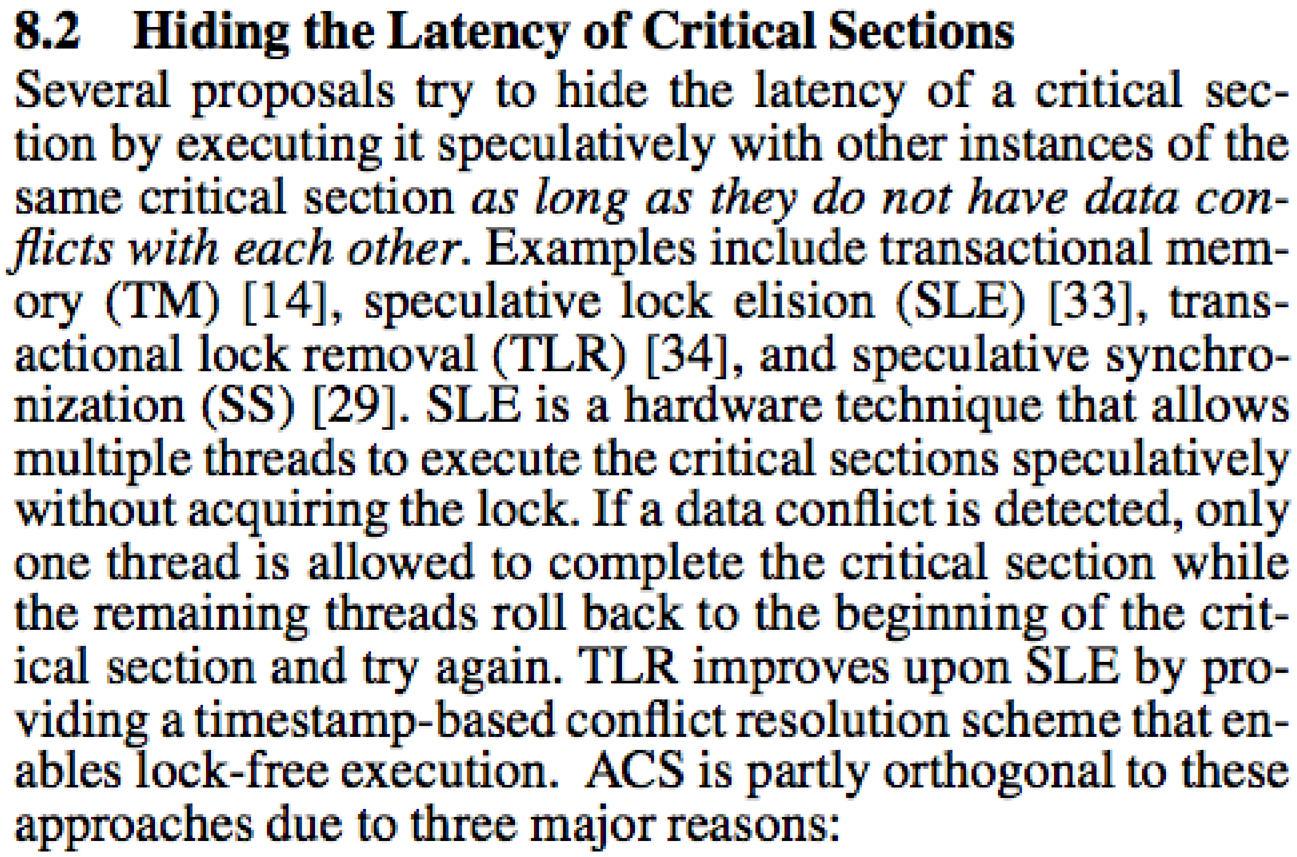
- If end of critical section reached without misspeculation, commit all writes (needs to appear instantaneous)

Accelerated Critical Sections (ACS) vs. SLE

- ACS Advantages over SLE
  + Speeds up each individual critical section
  + Keeps shared data and locks in a single cache (improves shared data and lock locality)
  + Does not incur re-execution overhead since it does not speculatively execute critical sections in parallel

- ACS Disadvantages over SLE
  - Needs transfer of private data and control to a large core (reduces private data locality and incurs overhead)
  - Executes non-conflicting critical sections serially
  - Large core can reduce parallel throughput (assuming no SMT)

ACS vs. SLE (I)

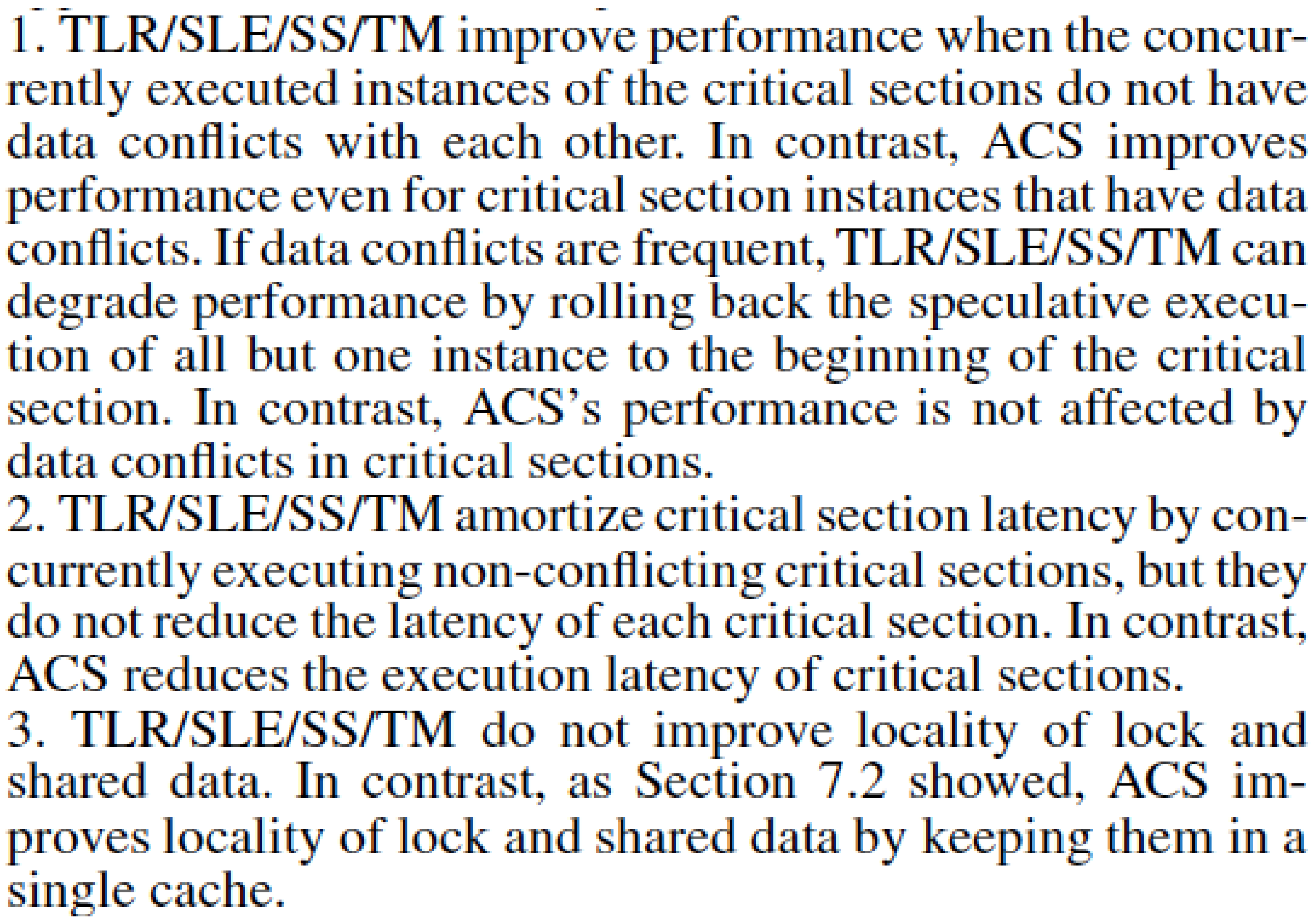


8.2 Hiding the Latency of Critical Sections

Several proposals try to hide the latency of a critical section by executing it speculatively with other instances of the same critical section *as long as they do not have data conflicts with each other*. Examples include transactional memory (TM) [14], speculative lock elision (SLE) [33], transactional lock removal (TLR) [34], and speculative synchronization (SS) [29]. SLE is a hardware technique that allows multiple threads to execute the critical sections speculatively without acquiring the lock. If a data conflict is detected, only one thread is allowed to complete the critical section while the remaining threads roll back to the beginning of the critical section and try again. TLR improves upon SLE by providing a timestamp-based conflict resolution scheme that enables lock-free execution. ACS is partly orthogonal to these approaches due to three major reasons:

Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009, IEEE Micro Top Picks 2010.

ACS vs. SLE (II)



1. TLR/SLE/SS/TM improve performance when the concurrently executed instances of the critical sections do not have data conflicts with each other. In contrast, ACS improves performance even for critical section instances that have data conflicts. If data conflicts are frequent, TLR/SLE/SS/TM can degrade performance by rolling back the speculative execution of all but one instance to the beginning of the critical section. In contrast, ACS's performance is not affected by data conflicts in critical sections.
2. TLR/SLE/SS/TM amortize critical section latency by concurrently executing non-conflicting critical sections, but they do not reduce the latency of each critical section. In contrast, ACS reduces the execution latency of critical sections.
3. TLR/SLE/SS/TM do not improve locality of lock and shared data. In contrast, as Section 7.2 showed, ACS improves locality of lock and shared data by keeping them in a single cache.

ACS vs. Transactional Lock Removal

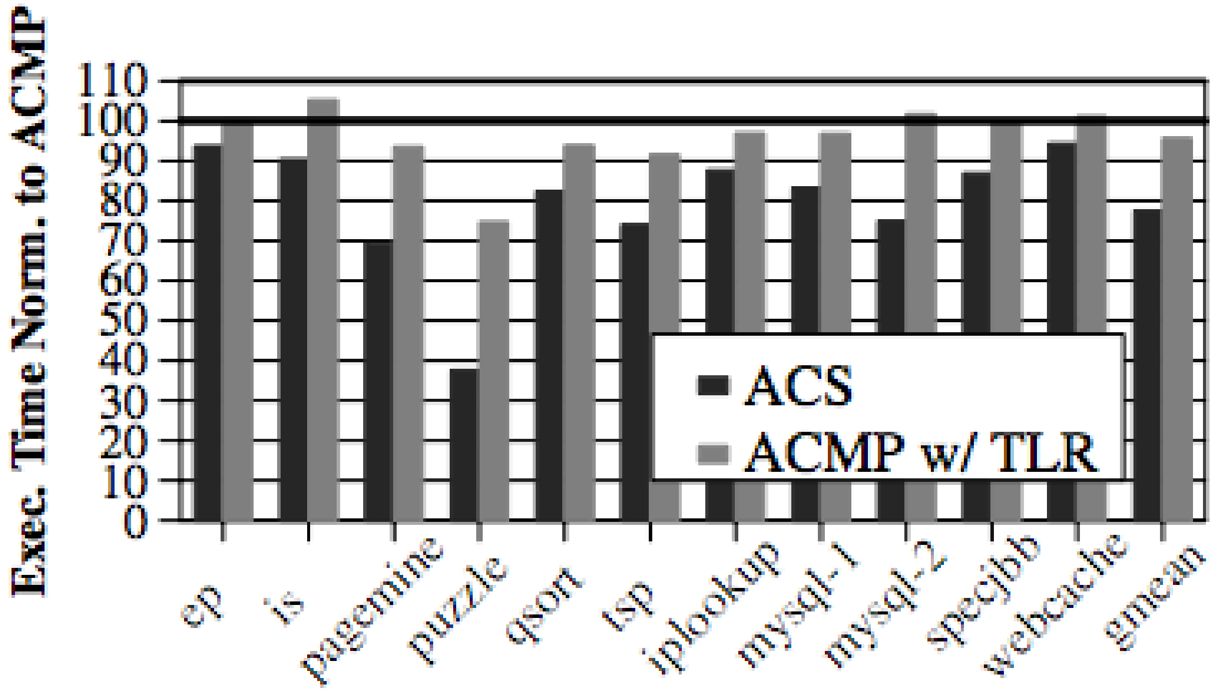


Figure 13. ACS vs. TLR performance.

ACS vs. SLE (Accelerating vs. Eliding Locks)

- Can you combine both?
- How would you combine both?
- Can you do better than both?

Four Issues in Speculative Parallelization

- How to deal with unavailable values: predict vs. wait
- How to deal with speculative updates: Logging/buffering
- How to detect conflicts
- How and when to abort/rollback or commit

- Eager vs. lazy approaches are possible for each issue

Transactional Memory

Transactional Memory

- Idea: Programmer specifies code to be executed atomically as transactions. Hardware/software guarantees atomicity for transactions.

- Motivated by difficulty of lock-based programming
- Motivated by lack of concurrency (performance issues) in blocking synchronization (or "pessimistic concurrency")

Locking Issues

- Locks: objects only one thread can hold at a time
  - Organization: lock for each shared structure
  - Usage: (block) → acquire → access → release

- Correctness issues
  - Under-locking → data races
  - Acquires in different orders → deadlock

- Performance issues
  - Conservative serialization
  - Overhead of acquiring
  - Difficult to find right granularity
  - Blocking

Locks vs. Transactions

Lock issues:	How transactions help:
– Under-locking → data races	+ Simpler interface/reasoning
– Deadlock due to lock ordering	+ No ordering
– Blocking synchronization	+ Nonblocking (Abort on conflict)
– Conservative serialization	+ Serialization only on conflicts

- Locks → pessimistic concurrency
- Transactions → optimistic concurrency

Transactional Memory

- Transactional Memory (TM) allows arbitrary multiple memory locations to be updated atomically (all or none)

- Basic Mechanisms:
  - Isolation and conflict management: Track read/writes per transaction, detect when a conflict occurs between transactions
  - Version management: Record new/old values (where?)
  - Atomicity: Commit new values or abort back to old values → all or none semantics of a transaction

- Issues the same as other speculative parallelization schemes
  - Logging/buffering
  - Conflict detection
  - Abort/rollback or commit

Four Issues in Transactional Memory

- How to deal with unavailable values: predict vs. wait
- How to deal with speculative updates: logging/buffering
- How to detect conflicts: lazy vs. eager
- How and when to abort/rollback or commit

Many Variations of TM

- Software (STM)

  -- High performance overhead (e.g., tracking read and write sets)

  ++ No virtualization issues

- Hardware (HTM)

  -- Need support for virtualization
  - What if buffering is not enough?
  - Context switches, I/O within transactions?

  ++ Low performance overhead

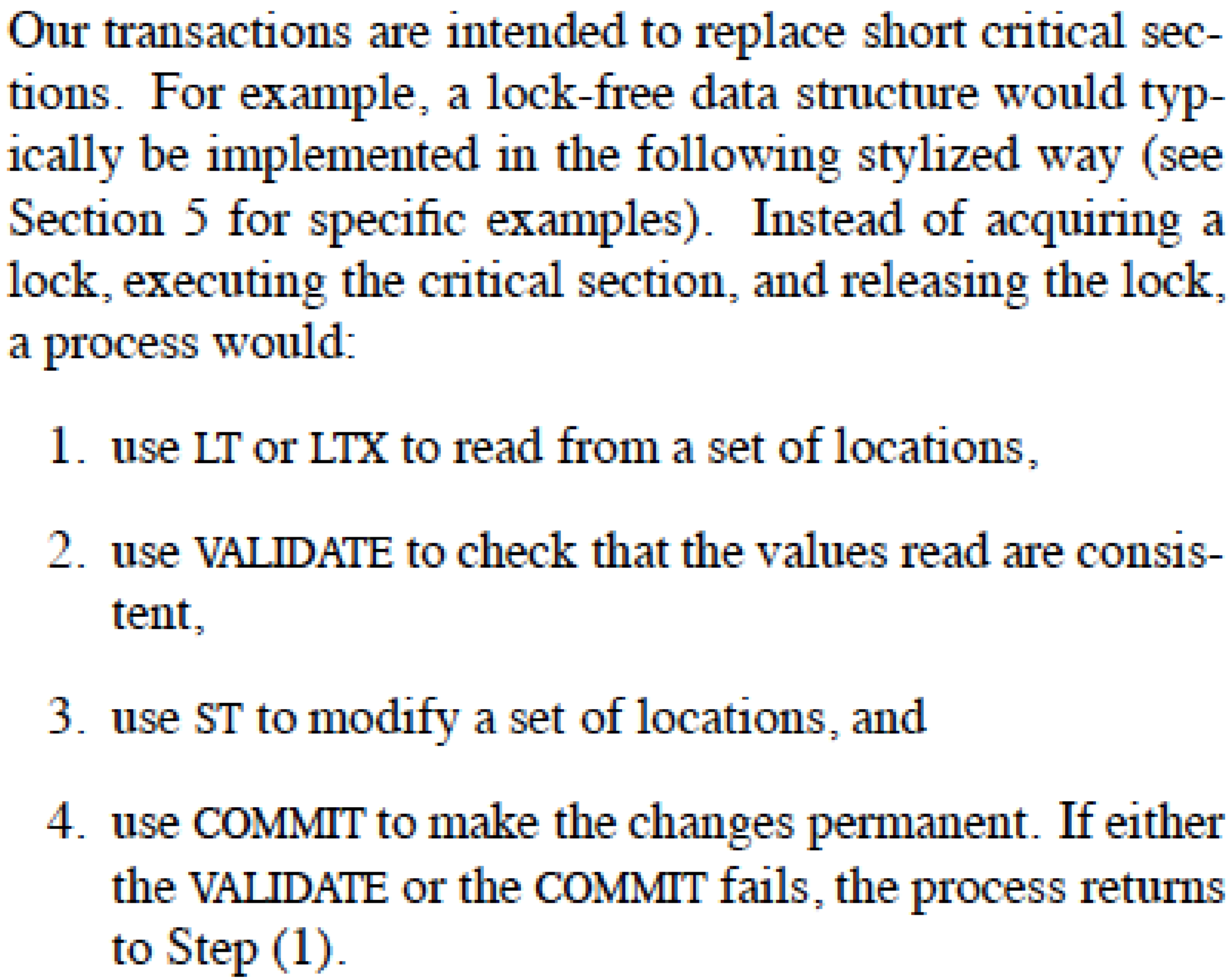
- Hybrid HW/SW
  - HTM when buffering is enough and no I/O or context switch
  - Switch to STM to handle long transactions and buffer overflows

Initial TM Ideas

- Load Linked Store Conditional Operations **[Jensen+, LLNL 1987]**
  - Lock-free atomic update of a single cache line
  - Used to implement non-blocking synchronization
    - Alpha, MIPS, ARM, PowerPC
  - Load-linked returns current value of a location
  - A subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to the location

- Herlihy and Moss **[Herlihy and Moss, ISCA 1993]**
  - Instructions explicitly identify transactional loads and stores
  - Used dedicated transaction cache for buffering
  - Size of transactions limited to transaction cache

Herlihy and Moss, ISCA 1993



Our transactions are intended to replace short critical sections. For example, a lock-free data structure would typically be implemented in the following stylized way (see Section 5 for specific examples). Instead of acquiring a lock, executing the critical section, and releasing the lock, a process would:

1. use LT or LTX to read from a set of locations,
2. use VALIDATE to check that the values read are consistent,
3. use ST to modify a set of locations, and
4. use COMMIT to make the changes permanent. If either the VALIDATE or the COMMIT fails, the process returns to Step (1).

Current Implementations of TM/SLE

- Sun ROCK
  - Dice et al., "Early Experience with a Commercial Hardware Transactional Memory Implementation," ASPLOS 2009.
- IBM Blue Gene
  - Wang et al., "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories," PACT 2012.
- IBM System z: Two types of transactions
  - Best effort transactions: Programmer responsible for aborts
  - Guaranteed transactions are subject to many limitations
  - Jacobi et al., "Transactional Memory Architecture and Implementation for IBM System Z," MICRO 2012.
- Intel Haswell
- AMD
  - Chung et al., "ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory," MICRO 2010.

Some TM Research Issues

- Scalability
  - Conflict and version management
- Complexity
  - How to virtualize transactions (without much complexity)
    - Ensure long transactions execute correctly
    - In the presence of context switches, paging
  - Handling I/O within transactions
  - Hardware complexity
- Semantics of nested transactions (more of a language/programming research topic)
- Does TM increase programmer productivity?
  - Does the programmer need to optimize transactions?

Computer Architecture: Speculation (in Parallel Machines)

Prof. Onur Mutlu
Carnegie Mellon University

Backup slides