



Architectural Support for Speculative Precomputation

Dean Tullsen
UCSD

on sabbatical at UPC



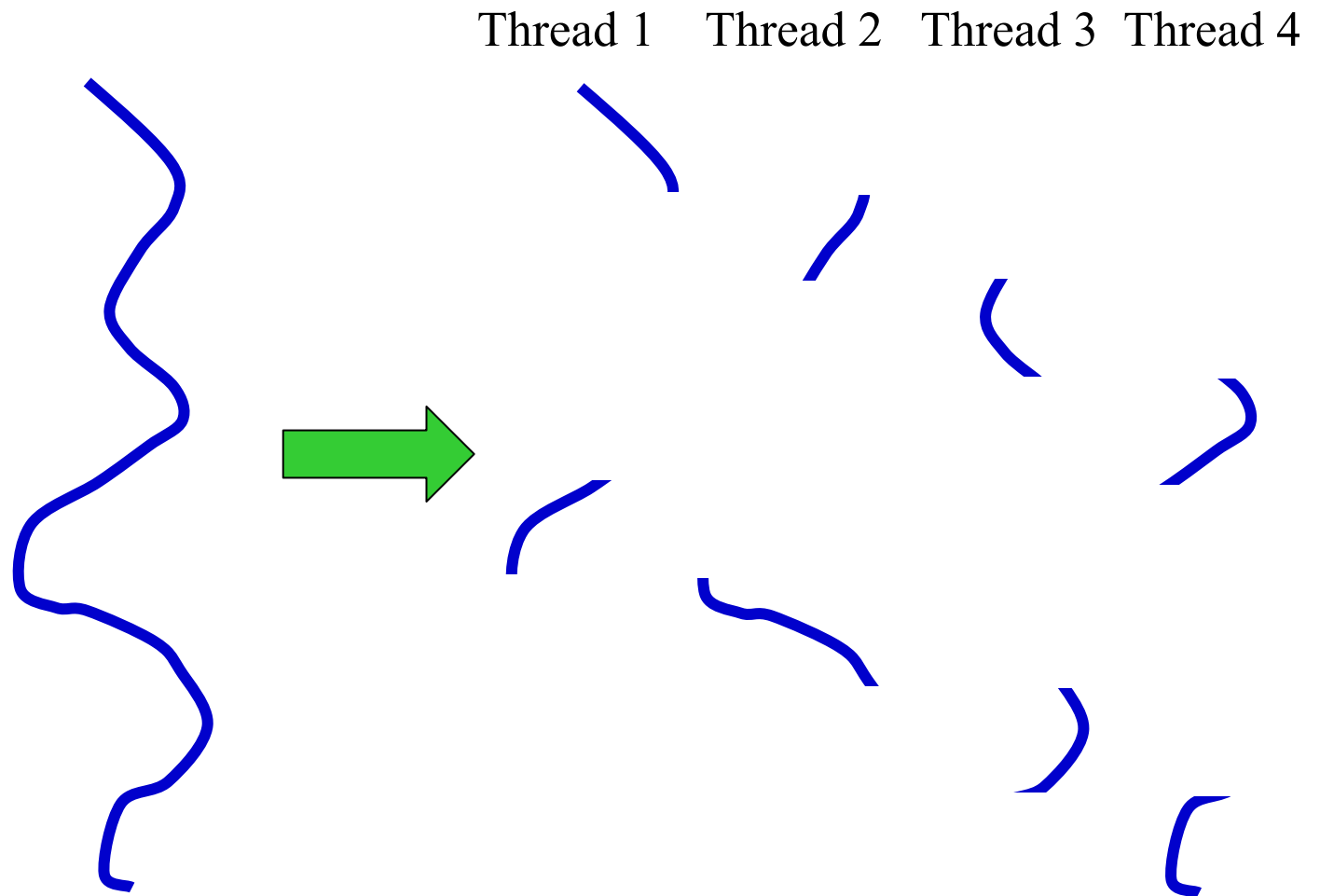
Background -- Three Types of Helper Threads

- Cache Prefetching
 - Branch Precomputation
 - Other
- What architectural support you need/want depends on what your helper thread is doing

Background – Helper Threads as Non-traditional Parallelism

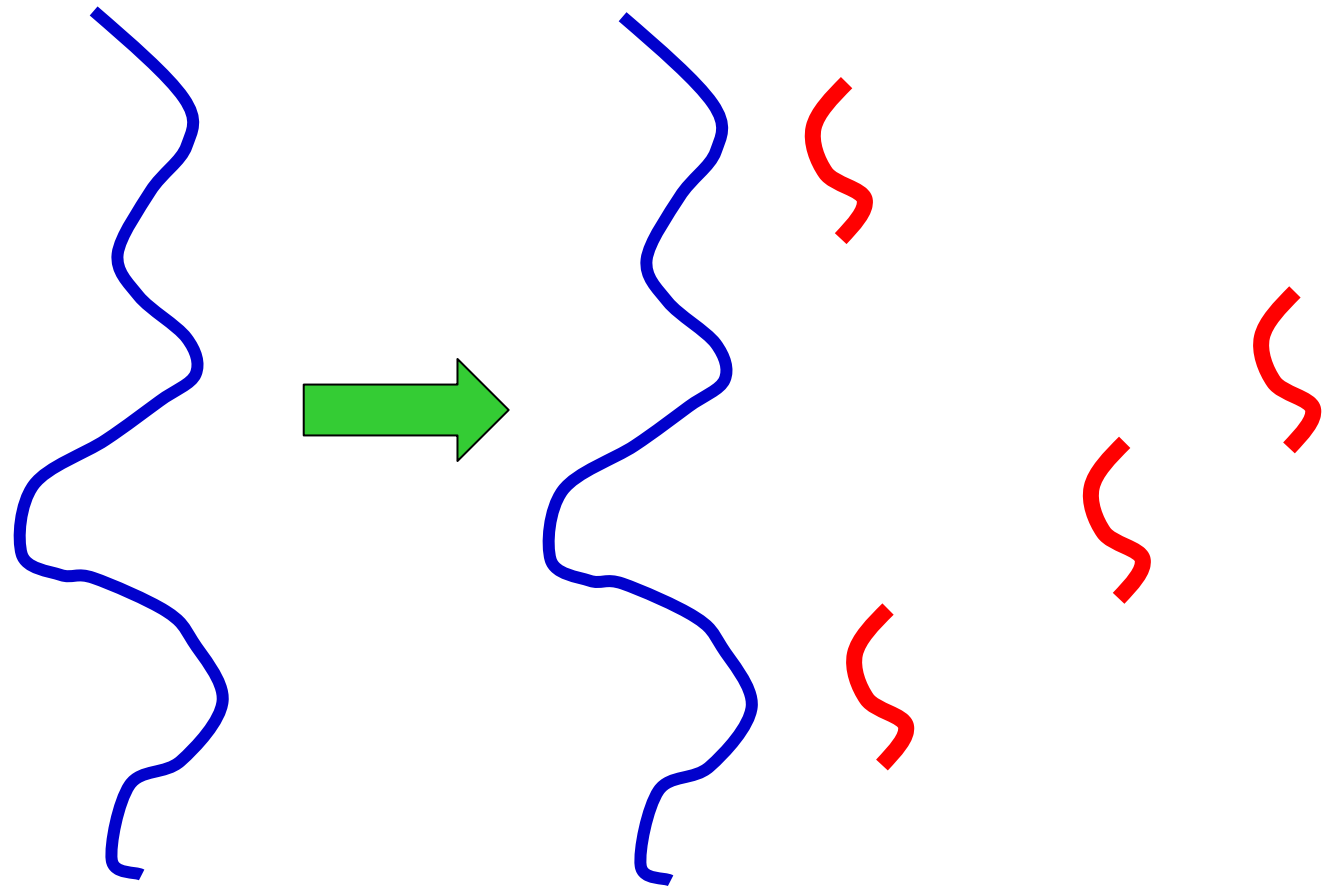
- Traditional Parallelism – We use extra threads/processors to *offload computation*. Threads divide up the execution stream.
- Helper threads – Extra threads are used to speed up computation *without necessarily off-loading any of the original computation*
 - Primary advantage → nearly any code, no matter how inherently serial, can benefit from parallelization.

Traditional Parallelism

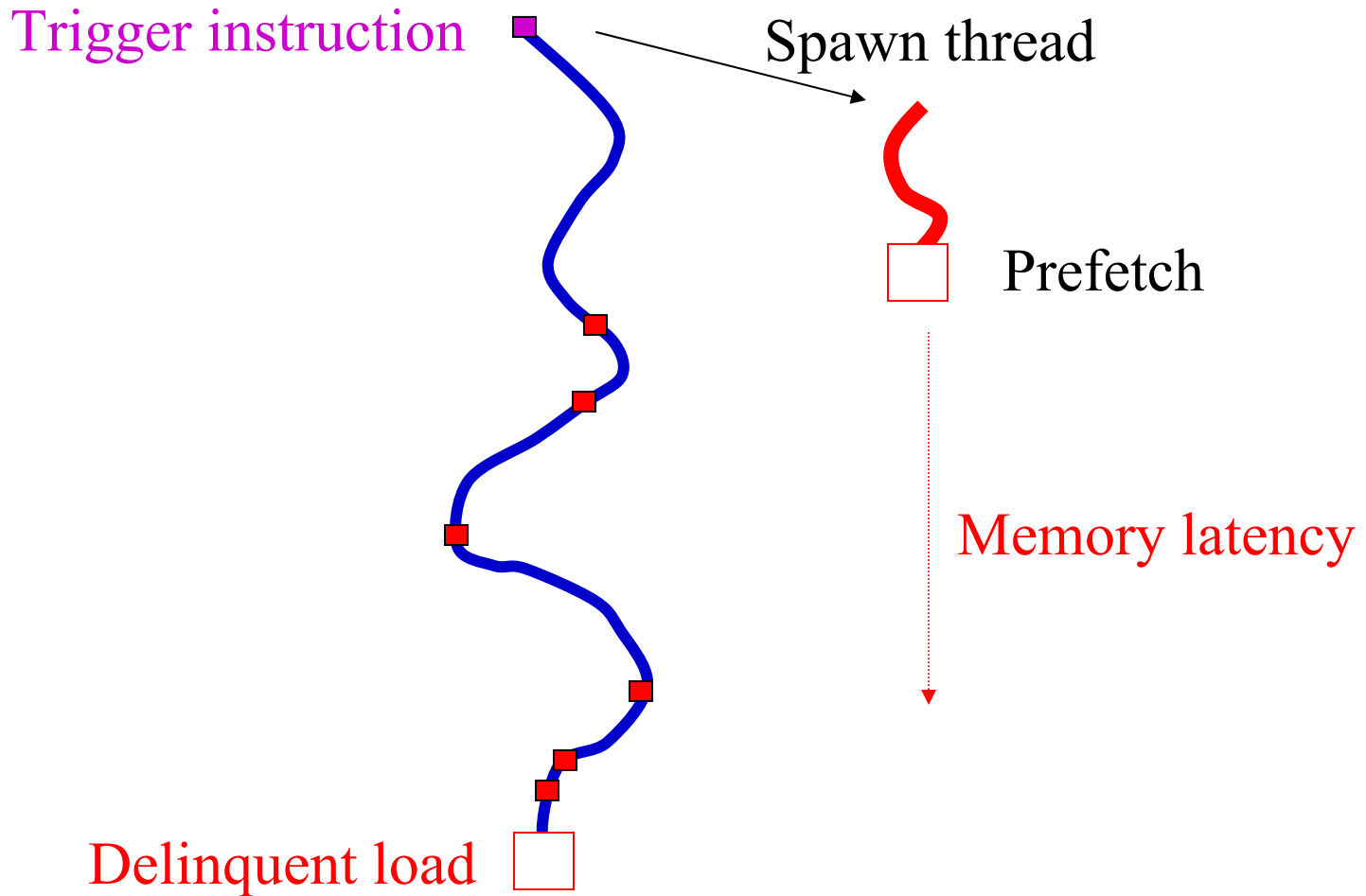


Helper Thread Parallelism

Thread 1 Thread 2 Thread 3 Thread 4



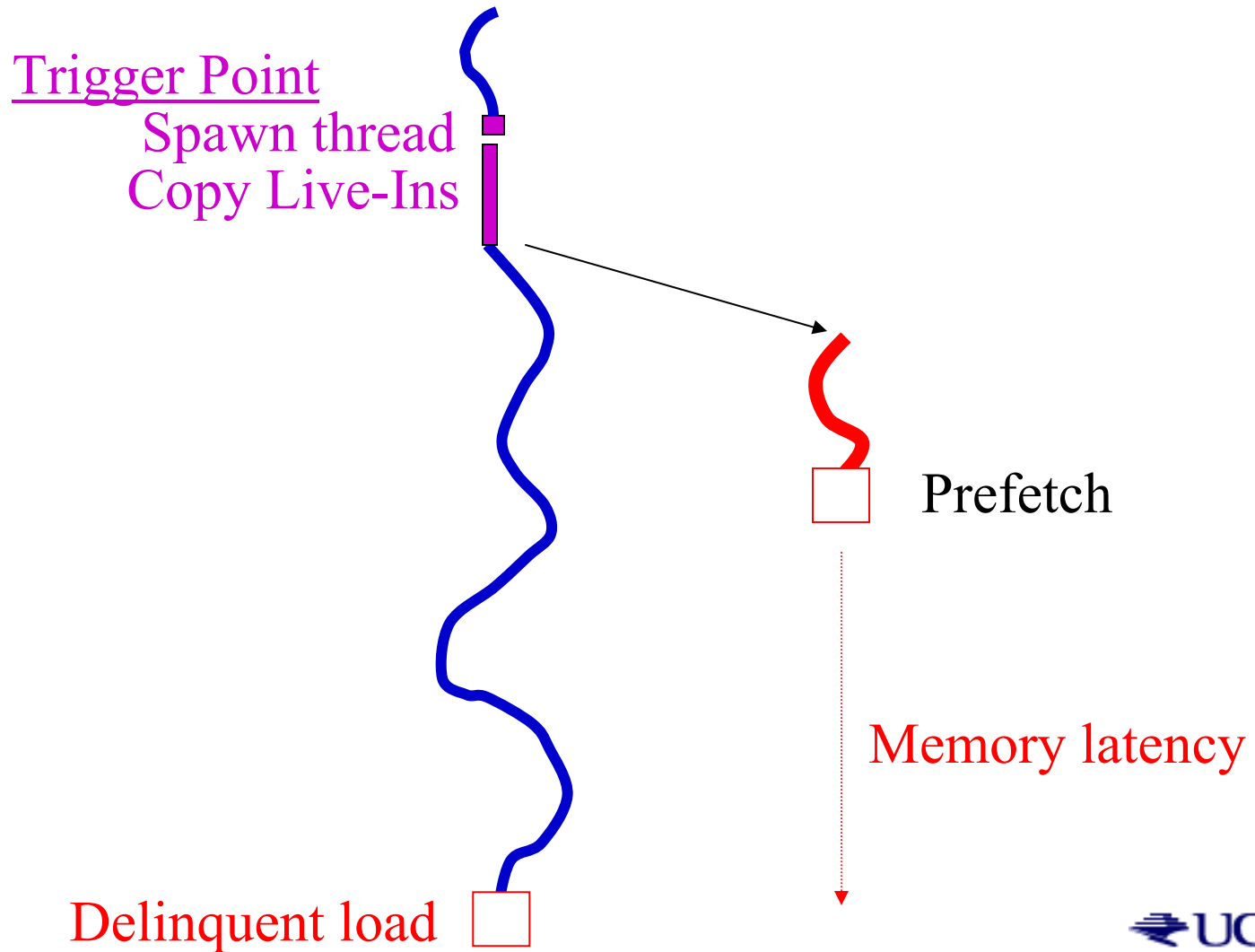
Speculative Precomputation



Other Helper Thread Models

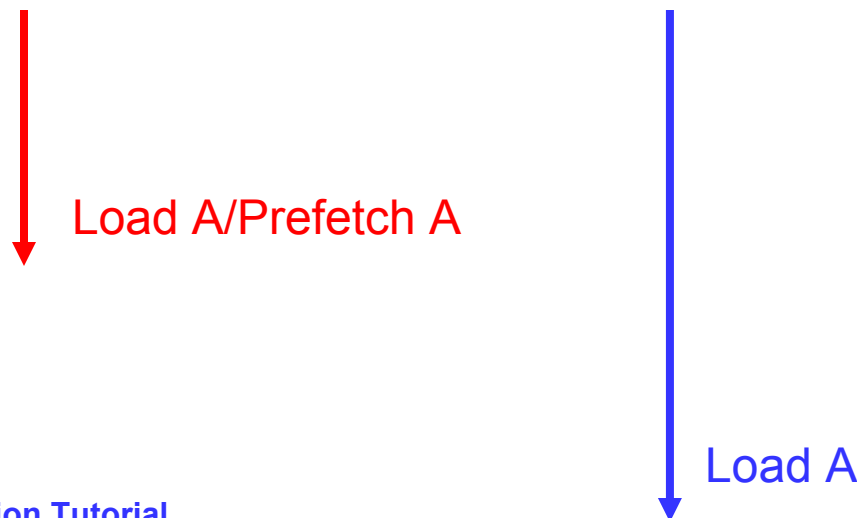
- For a description of helper threads that do not derive their code from the original thread, see:
 - Chappell, Stark, Kim, Reinhardt, Patt, "Simultaneous Subordinate Microthreading (SSMT)," ISCA 26

Helper Thread Model



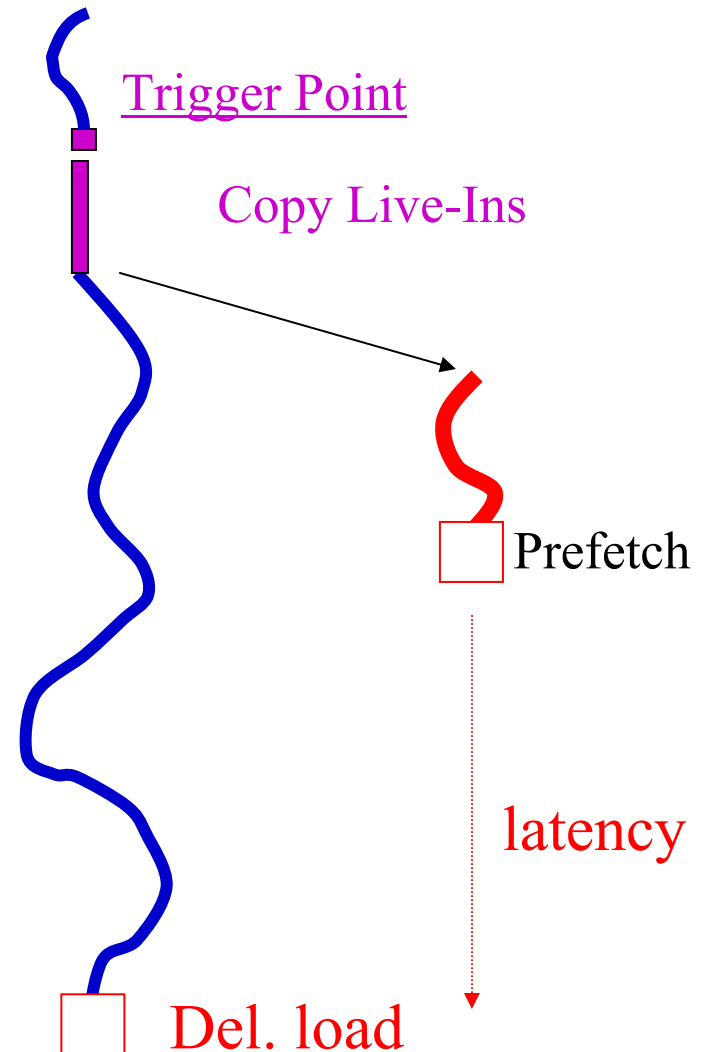
Cache Prefetching Architectural Support – **Minimum**

- None. Cache is a shared structure. One thread can bring in a cache line that is needed by another as a side effect.



Cache Prefetching Architectural Support – Useful


- fast thread spawns
- support for live-in transfer
- automatic triggering
- directed prefetches
- thread management
- thread creation!
- retention of computation



Branch Precomputation

Architectural Support – **Minimum**

- Although branch predictor is (possibly) shared, depending on branch side effects is ineffective.



BEQ R1, R2, label



BEQ R1, R2, label

Branch Precomputation Architectural Support – **Minimum**

- ISA support
- Outcome storage
- **Correlator**
- Ability to override branch predictor

Branch Precomputation Architectural Support – Useful

- fast thread spawns
- support for live-in transfer
- automatic triggering
- thread management
- thread creation
- retention of computation

Arch Support for other types of helper threads

■ Access to hardware structures

- branch predictor, BTB
- trace cache
- TLB
- Caches

■ Triggering

- branch predictor, BTB
- value profiler
- trace cache
- TLB
- Caches

Outline -- Architectural Support for Helper Threads

- **Branch Correlation Support**
- **Dynamic Speculative Precomputation -- Arch Support for Creation and Management of Helper Threads**
- **Register Integration – Arch Support for Reuse of Values in Helper Threads**

Prediction/Branch Correlation

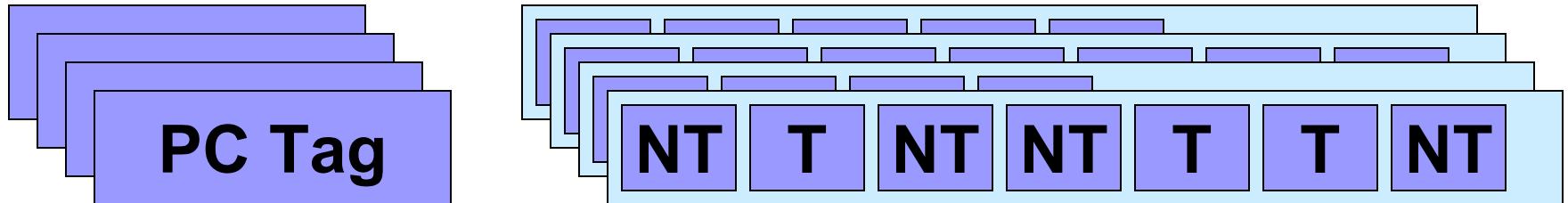
- This material heavily based on:
 - Zilles, Sohi, “Execution-Based Prediction Using Speculative Slices” ISCA 28

The Problem

- Even assuming the ability to match instructions in the helper thread with branch PCs in the main thread, we still must correlate *dynamic* instances of the helper thread predictions with *dynamic* instances of the branch in the main thread.

Overall Solution

- Tagged Prediction Queue

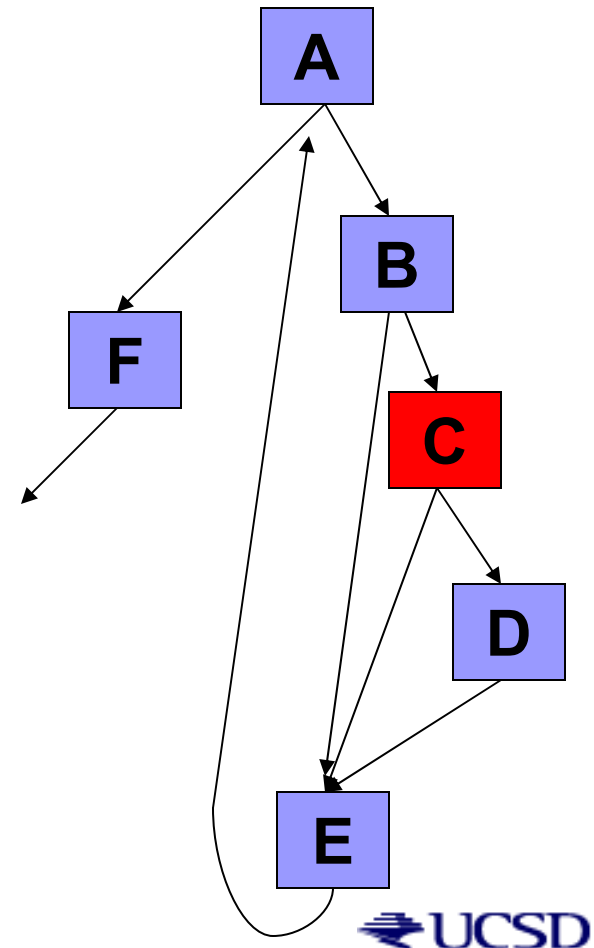


Challenges

- Re-ordering predictions produced out of order
 - ➔ allocate entries at fetch of prediction generating instruction
- Main Thread (MT) Mis-speculation recovery
 - ➔ consume at fetch of MT branch, free at commit
- Late predictions
 - ➔ MT must still consume empty entries, possibly establishing correlation with in-flight prediction
- Conditionally-executed branches

Conditionally Executed Branches

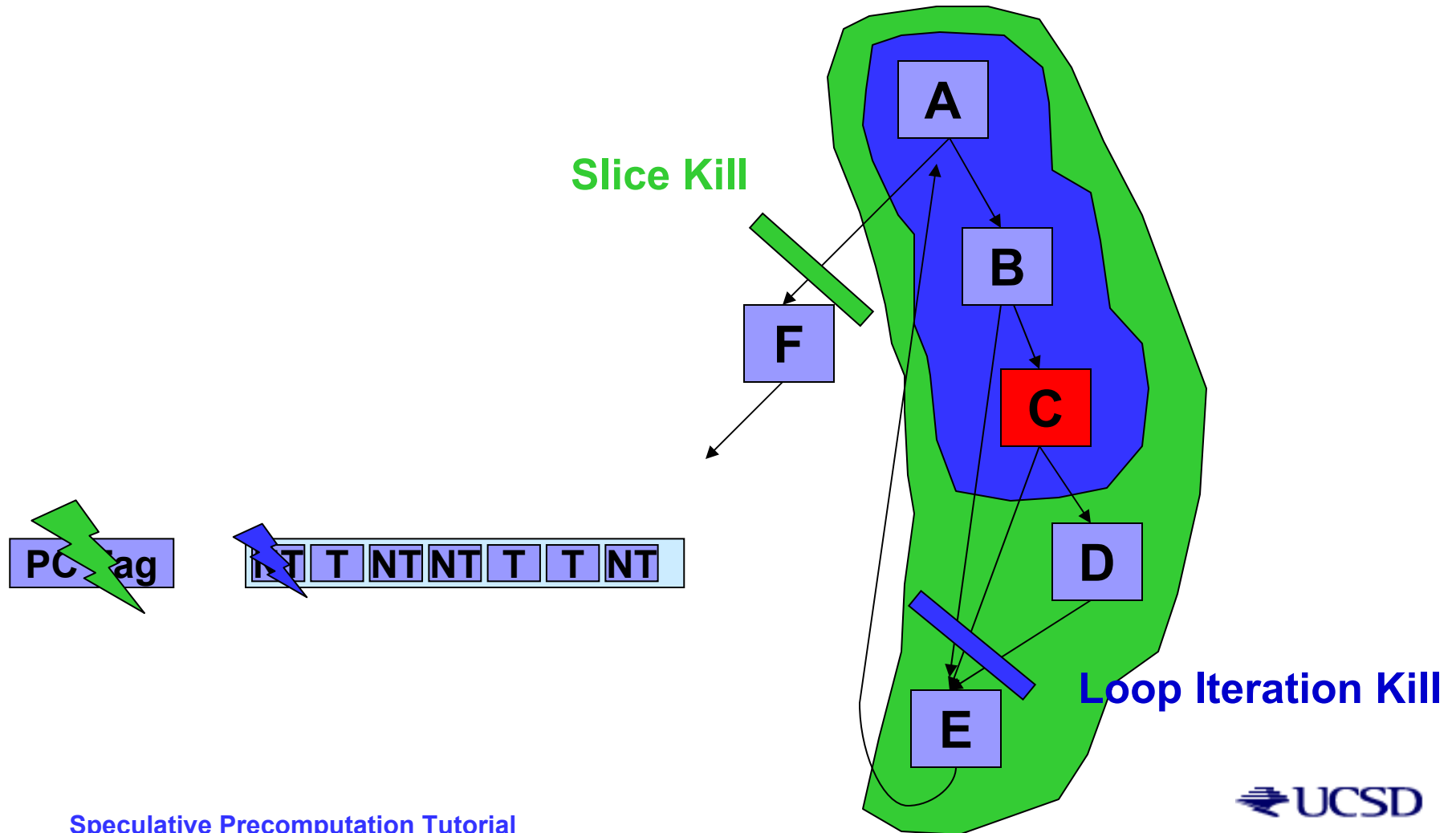
- Issue – Helper threads typically contain no control flow (except maybe a single loop back), and thus will generate a prediction for every iteration.



Conditionally Executed Branches

- Do not want to introduce control flow into the slice to conditionally consume predictions.
- Key – since the helper thread produces a prediction every iteration, we just consume one every iteration.
- Zilles and Sohi used fetch PC's to determine when a prediction should be killed (consumed). Could also use explicit instructions in main thread.

Conditionally Executed Branches



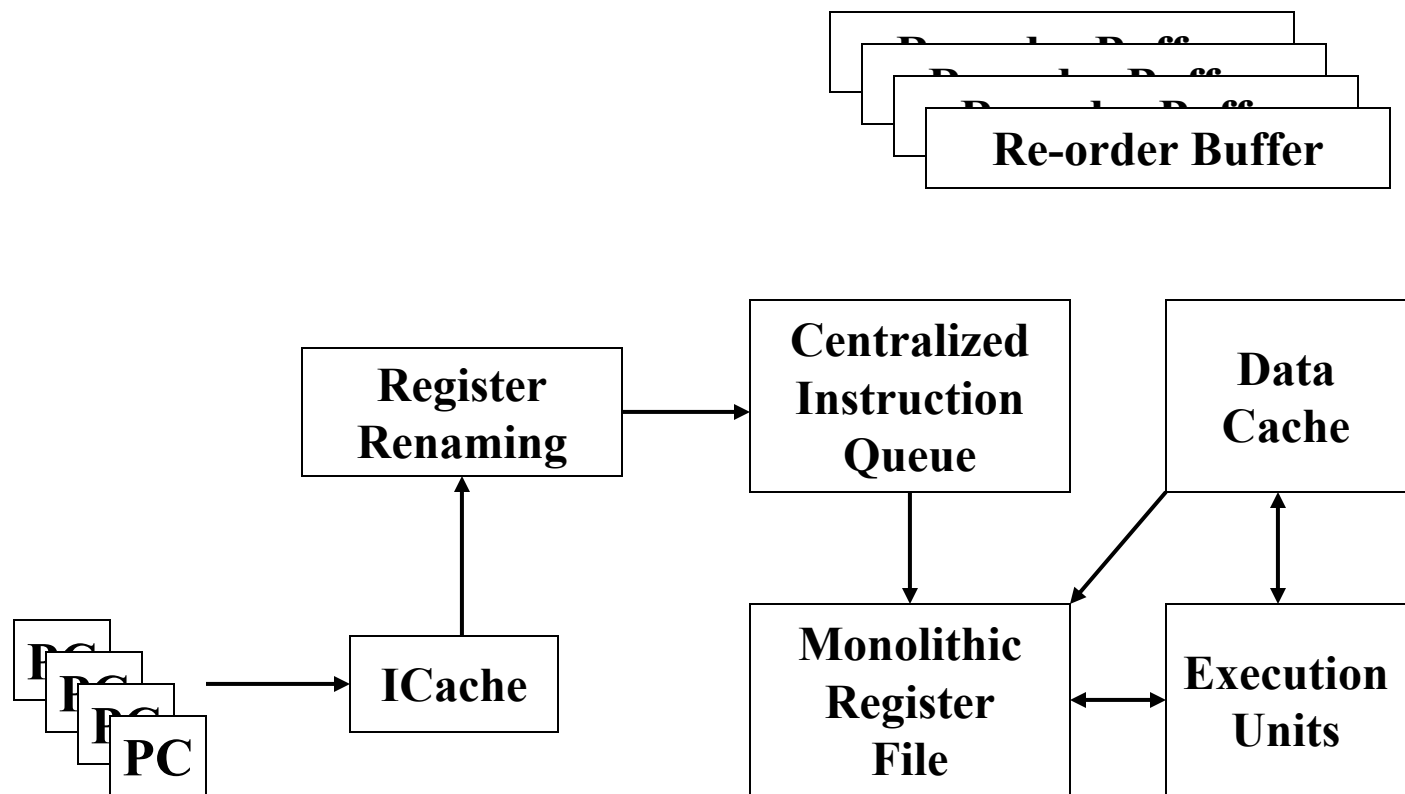
Outline -- Architectural Support for Helper Threads

- Branch Correlation Support
- **Dynamic Speculative Precomputation -- Arch Support for Creation and Management of Helper Threads**
- Register Integration – Arch Support for Reuse of Values in Helper Threads

Why Create Threads in Hardware – Why *Dynamic* Speculative Precomputation?

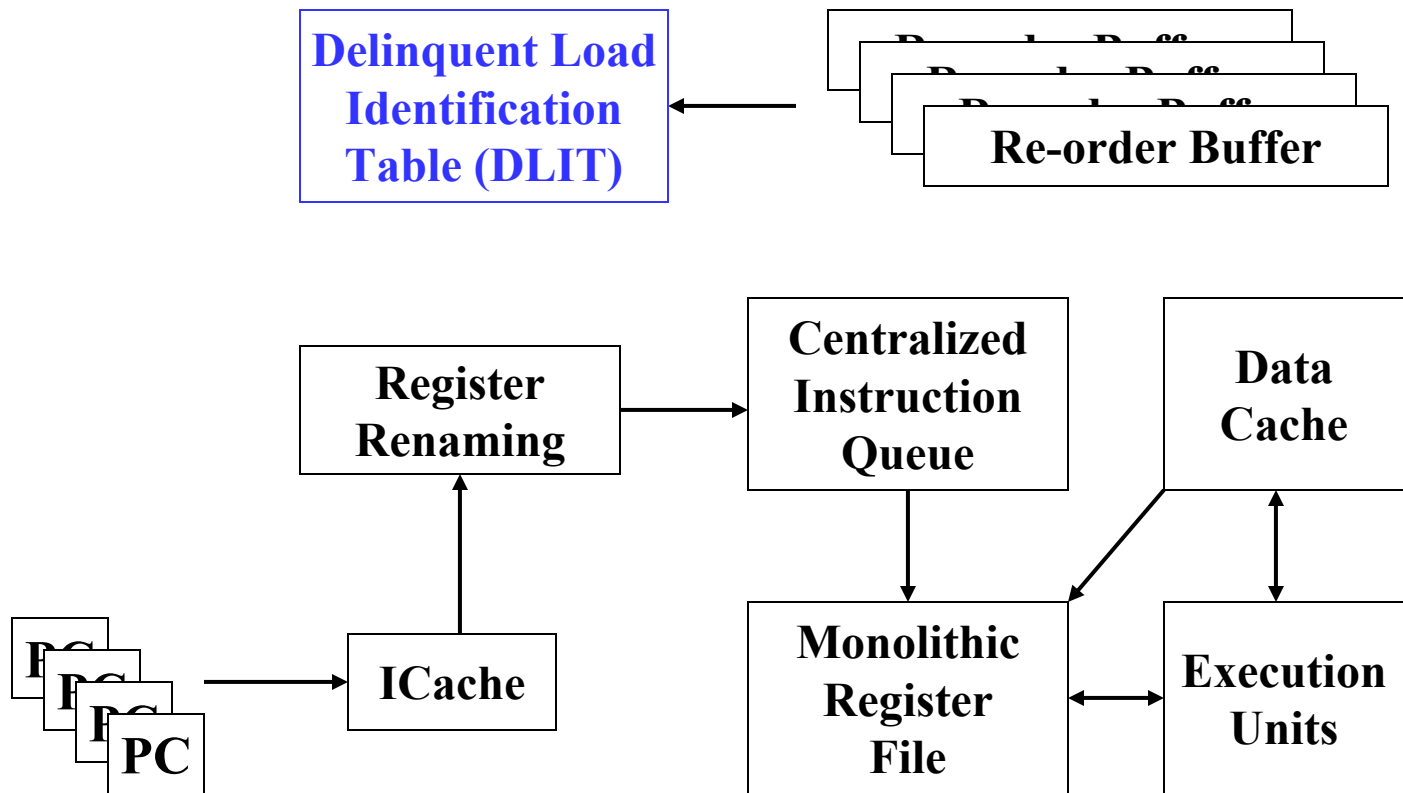
- SW Speculative Precomputation provides significant speedup, but
 - Requires offline program analysis
 - Creates threads for fixed number of thread contexts
 - Does not target existing code
 - Platform specific code
- A completely hardware-based version will use dynamic program analysis via back-end instruction analyzers.

Example SMT Processor Pipeline



Modified Pipeline

- Identify delinquent loads
- Construct P-slices
- Spawn and manage P-slices



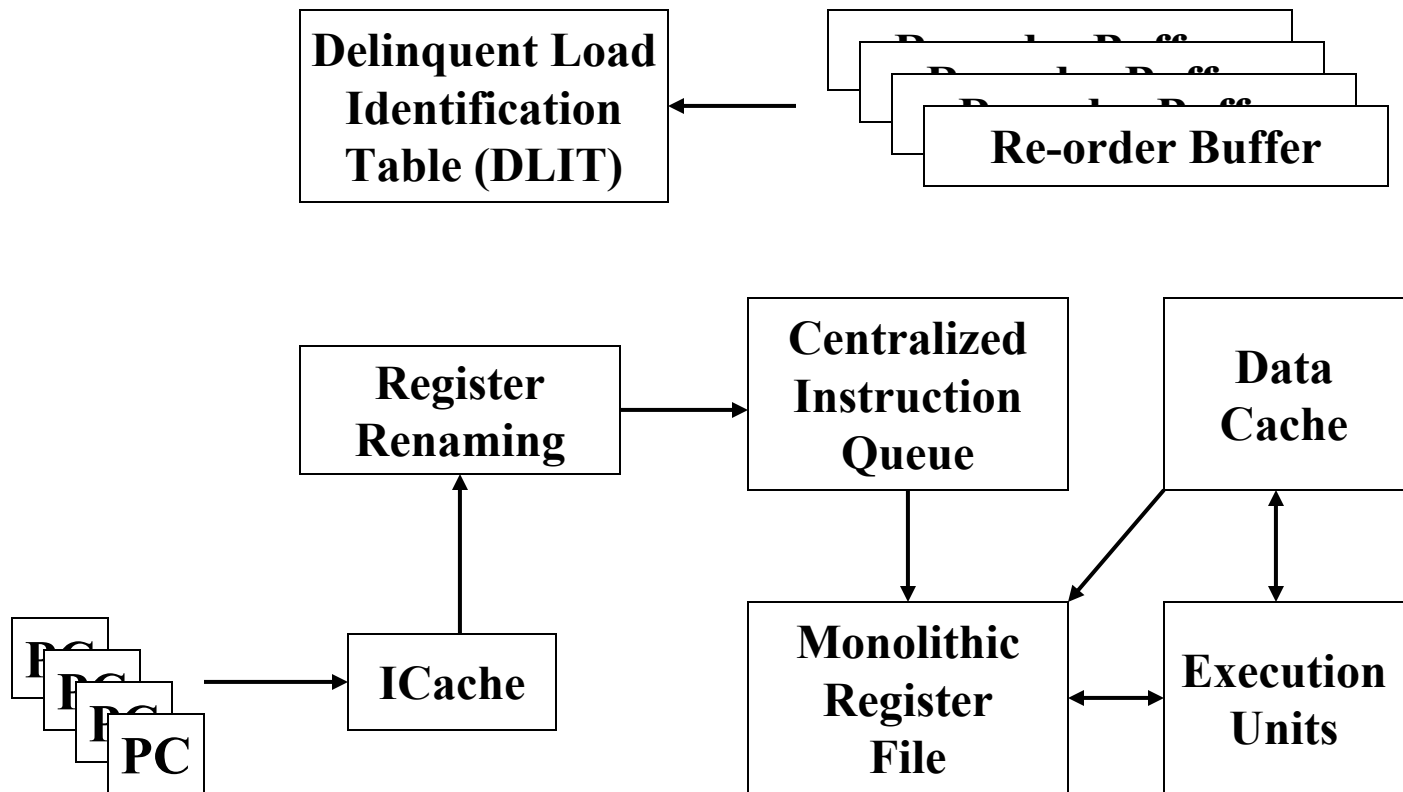
Delinquent Load Identification Table

- Identify PCs of program's delinquent loads
- Entries allocated to PC of loads which missed in L2 cache on last execution
 - First-come, first-serve
- Entry tracks average load behavior
- After 128k total instructions, evaluated for delinquency

Summary – finds delinquent loads

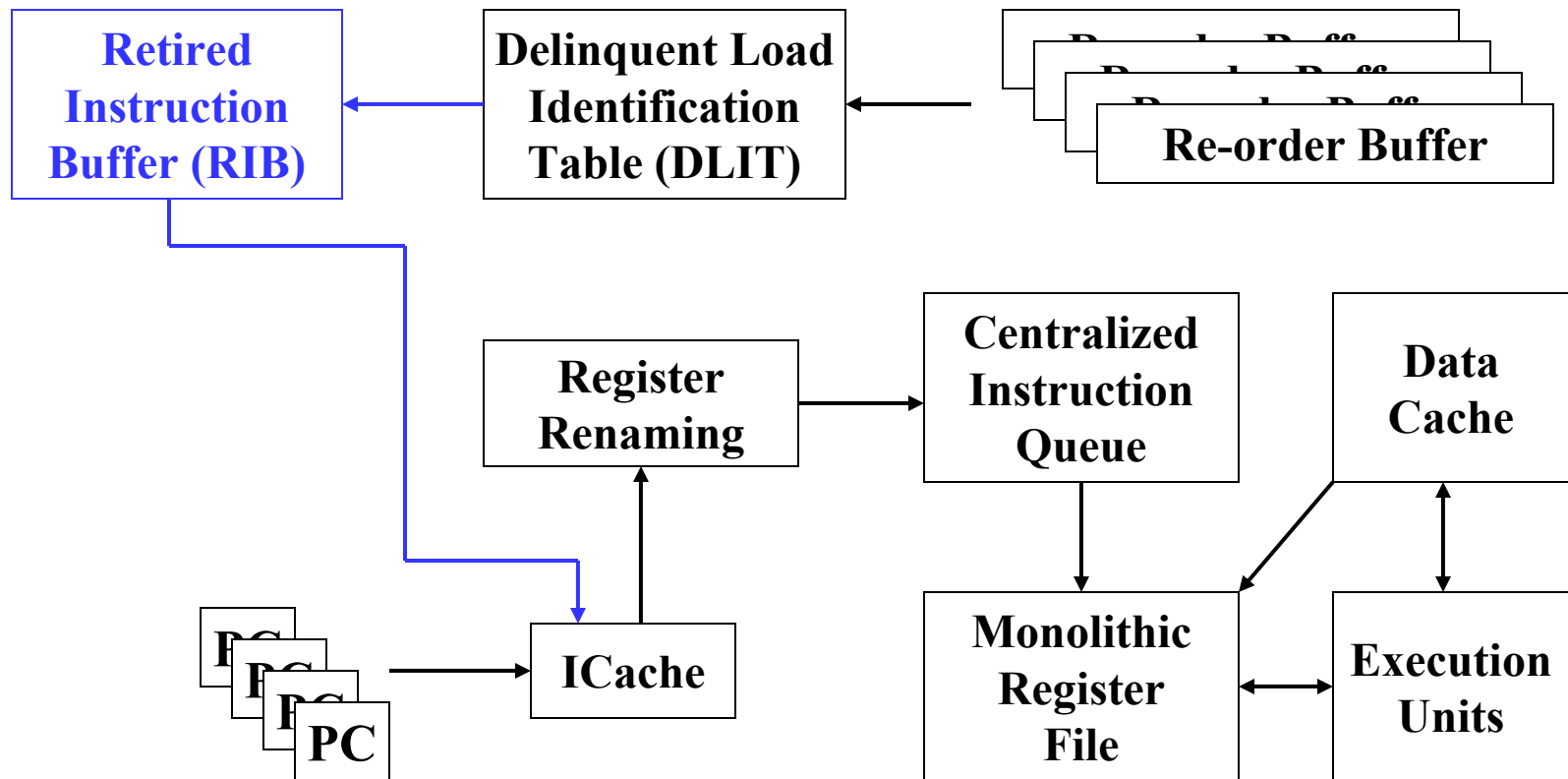
Modified Pipeline

- ✓ Identify delinquent loads
- Construct P-slices
- Spawn and manage P-slices



Modified Pipeline

- ✓ Identify delinquent loads
- Construct P-slices
- Spawn and manage P-slices



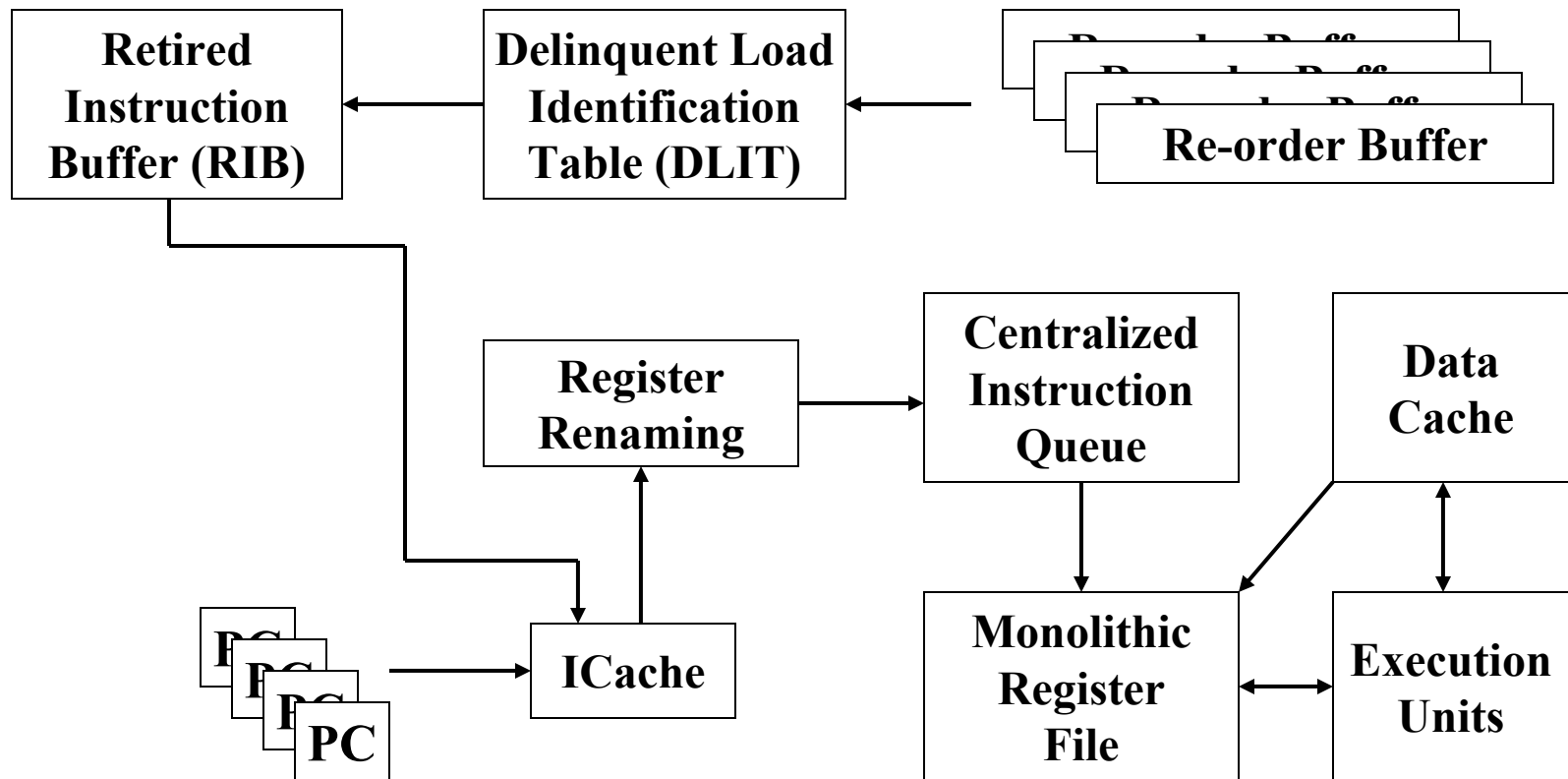
Retired Instruction Buffer

- Construct p-slices to prefetch delinquent loads
- Buffers information on an in-order run of committed instructions
 - Comparable to trace cache fill unit
- FIFO structure
- RIB normally idle (> 99% of the time)

➡ We'll spend more time on this.

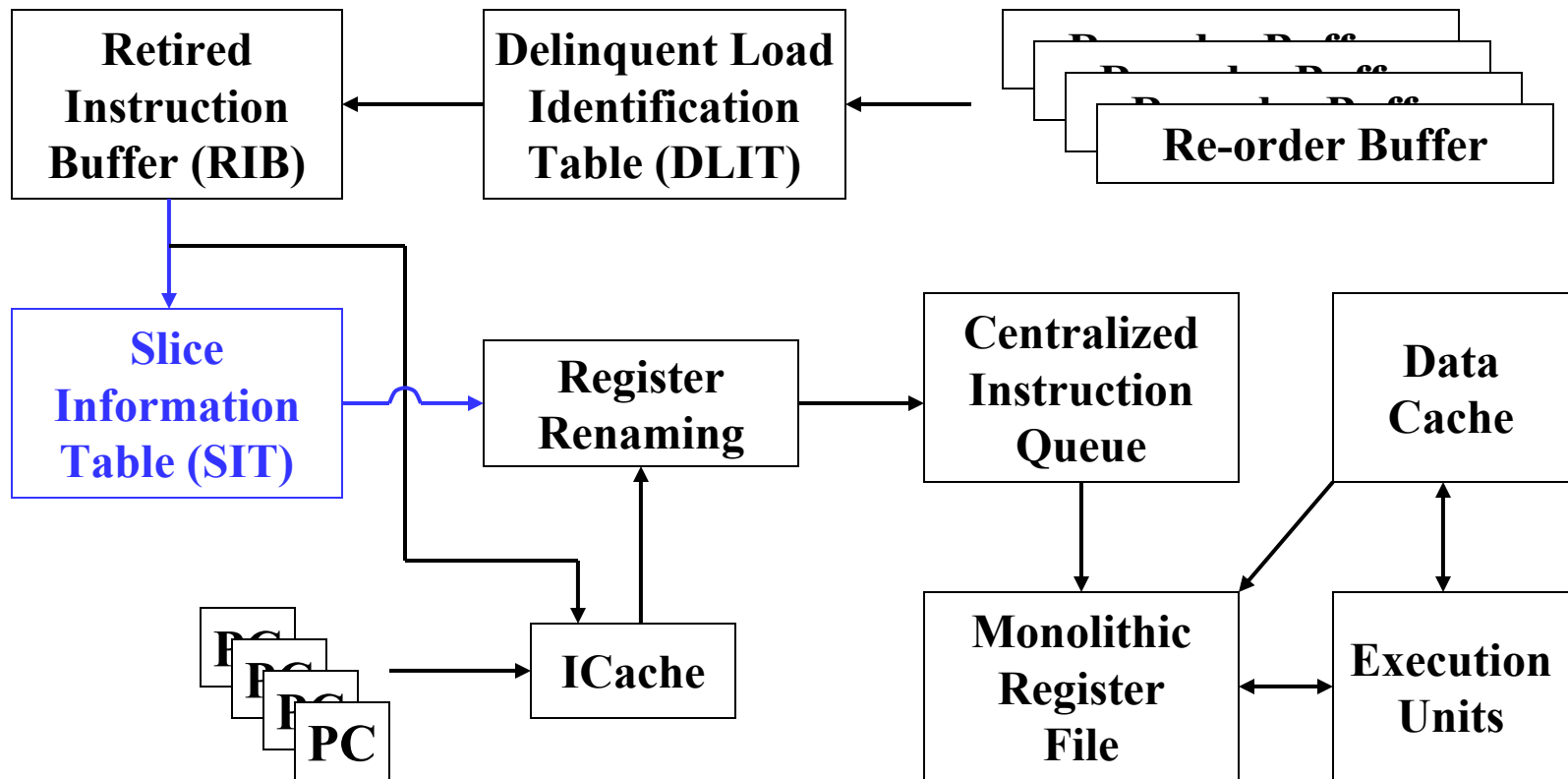
Modified Pipeline

- ✓ Identify delinquent loads
- ✓ Construct P-slices
- Spawn and manage P-slices



Modified Pipeline

- ✓ Identify delinquent loads
- ✓ Construct P-slices
- Spawn and manage P-slices



Slice Information Table

- Queried each cycle with addresses of main thread instructions decoded on that cycle
 - If trigger instruction decoded, rename stage notified
- Eliminates ineffective p-slices
 - P-slice evaluated every 128K committed instructions

P-slice Construction with RIB

- Analyze instructions between two instances of delinquent load
 - Most recent to oldest
- Add to p-slice instructions which produce live-in set register
 - Update register live-in set
- When analysis terminates, p-slice has been constructed and live-in registers identified

Example

```
struct DATATYPE {
    int val[10];
};

DATATYPE * data [100];

for(j = 0; j < 10; j++) {
    for(i = 0; i < 100; i++) {
        data[i]->val[j]++;
    }
}
```

```
loop:
I1  load    r1=[r2]
I2  add     r3=r3+1
I3  add     r6=r3-100
I4  add     r2=r2+8
I5  add     r1=r4+r1
I6  load    r5=[r1]
I7  add     r5=r5+1
I8  store   [r1]=r5
I9  blt    r6, loop
```

P-slice Construction Example

	Instruction	Included	Live-in Set
To oldest	<code>load r5 = [r1]</code>		
↑ Analyze from recent	<code>add r5 = r5+1</code>		
	<code>store [r1] = r5</code>		
	<code>blt r6, loop</code>		
	<code>load r1 = [r2]</code>		
	<code>add r3 = r3+1</code>		
	<code>add r6 = r3-100</code>		
	<code>add r2 = r2+8</code>		
	<code>add r1 = r4+r1</code>		
		<code>load r5 = [r1]</code>	

P-slice Construction Example

Instruction	Included	Live-in Set
load r5 = [r1]		
add r5 = r5+1		r2, r4
store [r1] = r5		r2, r4
blt r6, loop		r2, r4
load r1 = [r2]	√	r2, r4
add r3 = r3+1		r1, r4
add r6 = r3-100		r1, r4
add r2 = r2+8		r1, r4
add r1 = r4+r1	√	r1, r4
load r5 = [r1]	√	r1

P-slice Construction Example

Instruction

load r5 = [r1]
add r5 = r5+1
store [r1] = r5
blt r6, loop
load r1 = [r2]
add r3 = r3+1
add r6 = r3-100
add r2 = r2+8
add r1 = r4+r1
load r5 = [r1]

P-Slice

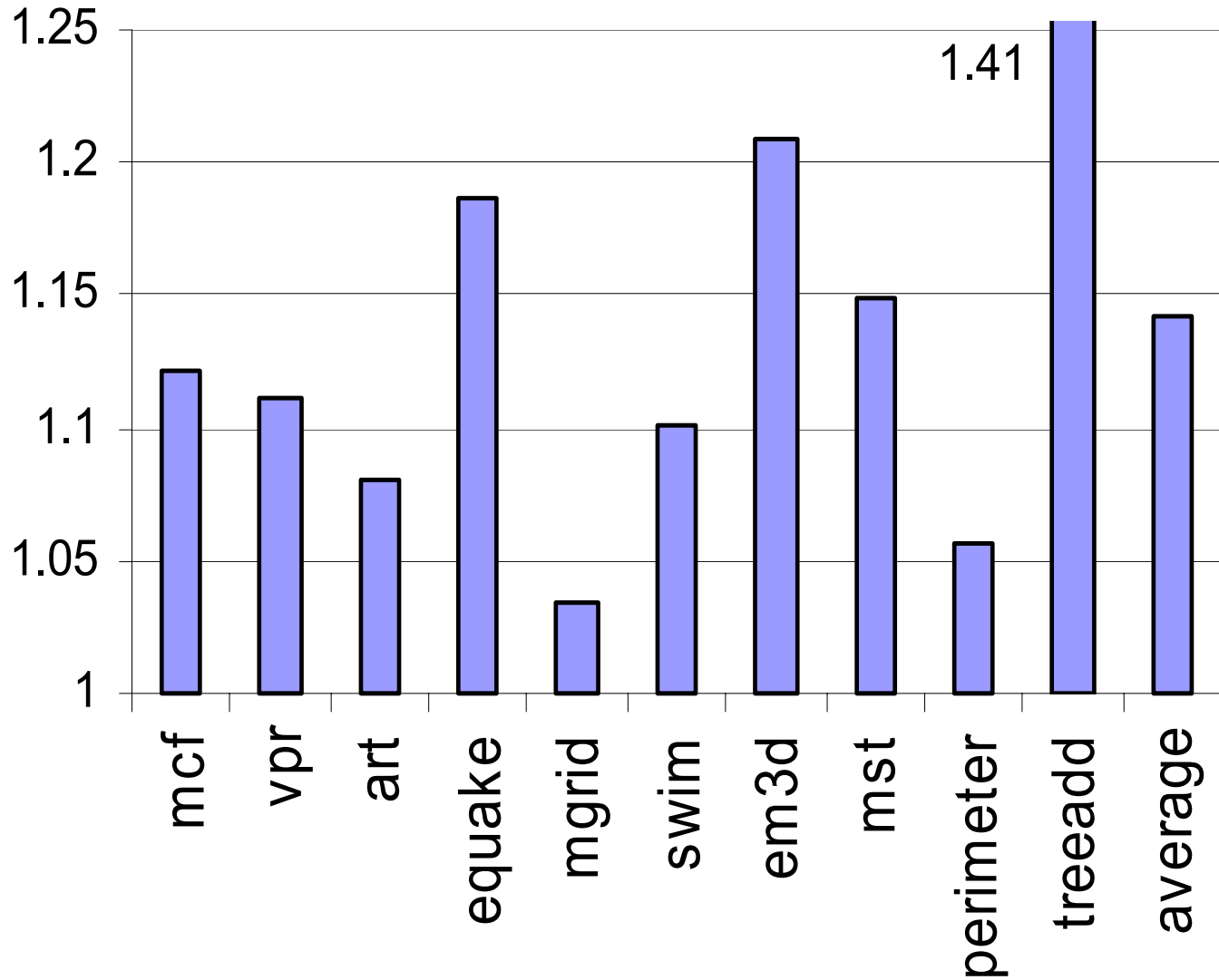
load r1 = [r2]
add r1 = r4+r1
load r5 = [r1]

Live-in Set

r2, r4

Delinquent Load
is trigger

Speedup Over no Dynamic SP



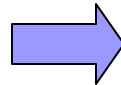
Advanced SP Optimizations

- All aimed at earlier prefetch initiation
- All require *two instances* of delinquent load in RIB
- Simply implemented with multiple passes through RIB

RIB

```
load    r5=[r1]
add     r5=r5+1
store   [r1]=r5
blt     r6, loop
load    r1=[r2]
add     r3=r3+1
add     r6=r3-100
add     r2=r2+8
add     r1=r4+r1
load    r5=[r1]
```

trigger



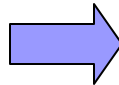
```
add     r5=r5+1
store   [r1]=r5
blt     r6, loop
load    r1=[r2]
add     r3=r3+1
add     r6=r3-100
add     r2=r2+8
add     r1=r4+r1
load    r5=[r1]
```


Advanced SP Optimizations

RIB

```
load    r5=[r1]
add     r5=r5+1
store   [r1]=r5
blt     r6, loop
load    r1=[r2]
add     r3=r3+1
add     r6=r3-100
add     r2=r2+8
add     r1=r4+r1
load    r5=[r1]
```

trigger

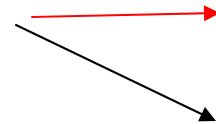


```
add     r5=r5+1
store   [r1]=r5
blt     r6, loop
load    r1=[r2]
add     r3=r3+1
add     r6=r3-100
add     r2=r2+8
add     r1=r4+r1
load    r5=[r1]
add     r5=r5+1
store   [r1]=r5
blt     r6, loop
load    r1=[r2]
add     r3=r3+1
add     r6=r3-100
add     r2=r2+8
add     r1=r4+r1
load    r5=[r1]
```

Trigger Placement

Live ins:
r2, r4

trigger

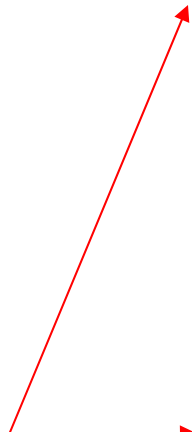


```
add    r5=r5+1
store  [r1]=r5
blt    r6, loop
load   r1=[r2]
add    r3=r3+1
add    r6=r3-100
add    r2=r2+8
add   r1=r4+r1
load  r5=[r1]
add    r5=r5+1
store  [r1]=r5
blt    r6, loop
load  r1=[r2]
add    r3=r3+1
add    r6=r3-100
add    r2=r2+8
add   r1=r4+r1
load  r5=[r1]
```

Induction Unrolling [Roth, Sohi, HPCA 7]

Live ins:
r2, r4

trigger



```
add    r5=r5+1
store  [r1]=r5
blt    r6, loop
load   r1=[r2]
add    r3=r3+1
add    r6=r3-100
add    r2=r2+8
add    r1=r4+r1
load   r5=[r1]
add    r5=r5+1
store  [r1]=r5
blt    r6, loop
load   r1=[r2]
add    r3=r3+1
add    r6=r3-100
add    r2=r2+8
add    r1=r4+r1
load   r5=[r1]
```

Chaining Slices

- Requires undetermined (typically small) number of passes.
- Uses previous passes' live ins, to continue adding instructions that effect future iterations' delinquent loads.
- Ends when no more changes introduced.
- Loop back branch added to end.

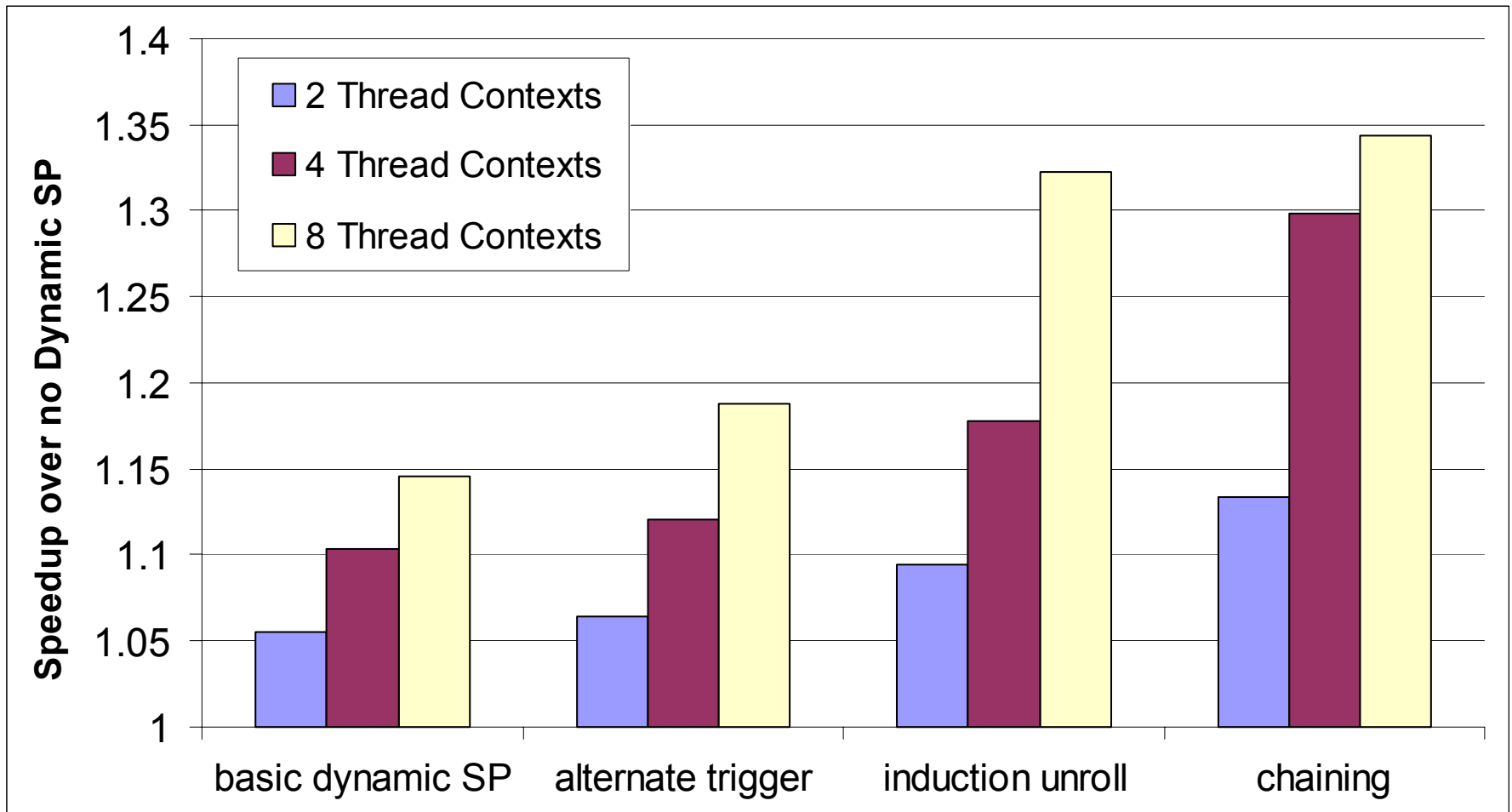
```
add    r5=r5+1
store  [r1]=r5
blt    r6, loop
load   r1=[r2]
add    r3=r3+1
add    r6=r3-100
add    r2=r2+8
add    r1=r4+r1
load  r5=[r1]
```

Dynamic SP Optimizations – Chaining P-slices

- Enable p-slice to repeat its execution in same thread context
 - Reduce contention for thread contexts
 - Eliminate redundant induction variable updates
- Must manage runahead distance
- Kill threads when non-speculative thread leaves program section

Advanced Dynamic SP Optimizations, summary

- Goal – spawn threads earlier
 - Assume control flow repeated
- Perform additional analysis passes
 - Retain live-in set from previous pass
- Increased construction latency but keeps RIB simple
 - Little performance impact



Dynamic SP Conclusion

- Dynamic Speculative Precomputation aggressively targets delinquent loads
 - Thread based prefetching scheme
 - Uses back-end (off critical path) instruction analyzers
 - P-slices constructed with no external software support
- Basic form gives average 14% speedup
- Multi-pass RIB analysis enables aggressive p-slice optimizations
 - Average 33% speedup using chaining with eight contexts

Speculative Precomputation

- *Speculative Precomputation: Long-range Prefetching of Delinquent Loads*, Collins, Wang, Tullsen, Hughes, Lee, Lavery, Shen, In *ISCA 2001*
- *Dynamic Speculative Precomputation*, Collins, Tullsen, Wang, Shen, In *Micro 2001*

Outline -- Architectural Support for Helper Threads

- Branch Correlation Support
- Dynamic Speculative Precomputation -- Arch Support for Creation and Management of Helper Threads
- **Register Integration** – Arch Support for Reuse of Values in Helper Threads

Helper Thread Value Reuse

- Focus on one particular technique, derived from two papers:
 - Register Integration [Roth, Sohi, Micro 2000] – identifies instructions that are being re-executed with the exact same dependencies ([squash reuse](#))
 - Speculative Data Driven Multithreading (DDMT) [Roth, Sohi, HPCA 2001] – constructs helper threads in such a way that register integration kicks in automatically.

Motivation (Register Integration for Squash Reuse)

- Assume Unified Physical Register File (PRF)

- Logical Register Map (LRM) sequentially “manages” PRF

- Conventional mis-speculation recovery

- PR values intact
- LRM restored to prior state, PR’s become “garbage”

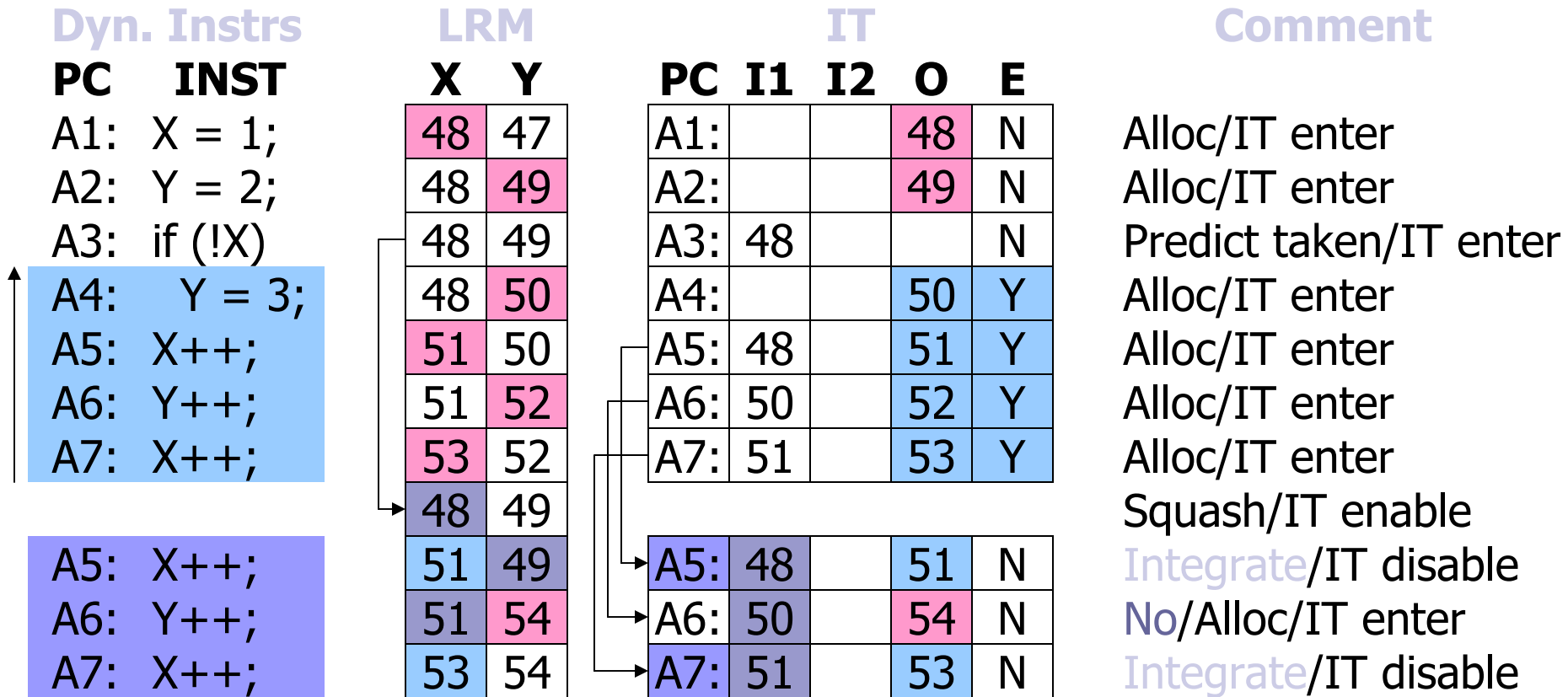
- Register Integration: why write? value is already in PR

```
add
sub
sll
beq
  add
  add
sub
mul
add
```

Register Integration

- Key: must locate PR holding squashed value
 - Use a second mapping of PRF
 - Integration Table (IT): describe each PR using creating instruction
 - Operation (PC) and input PR's
 - Encodes “reusability criteria”

Integration in Action (Squash Reuse)



E = Eligible (can be integrated)

PR cannot simultaneously be mapped by two active instructions

What Integration (Reuse) Accomplishes

- Improved performance (first-order effects)
- Reduced resource contention

Data-Driven Multithreading (DDMT)

- **DDMT**: an implementation of pre-execution which uses register integration to recapture some of the computation done by the main thread.

Example Identify PIs

- Simplified loop from EM3D

STATIC CODE

```
for (node=list; node; node = node->next)
  if (node->neighbor != NULL)
    node->val -= node->neighbor->val * node->coeff;
```

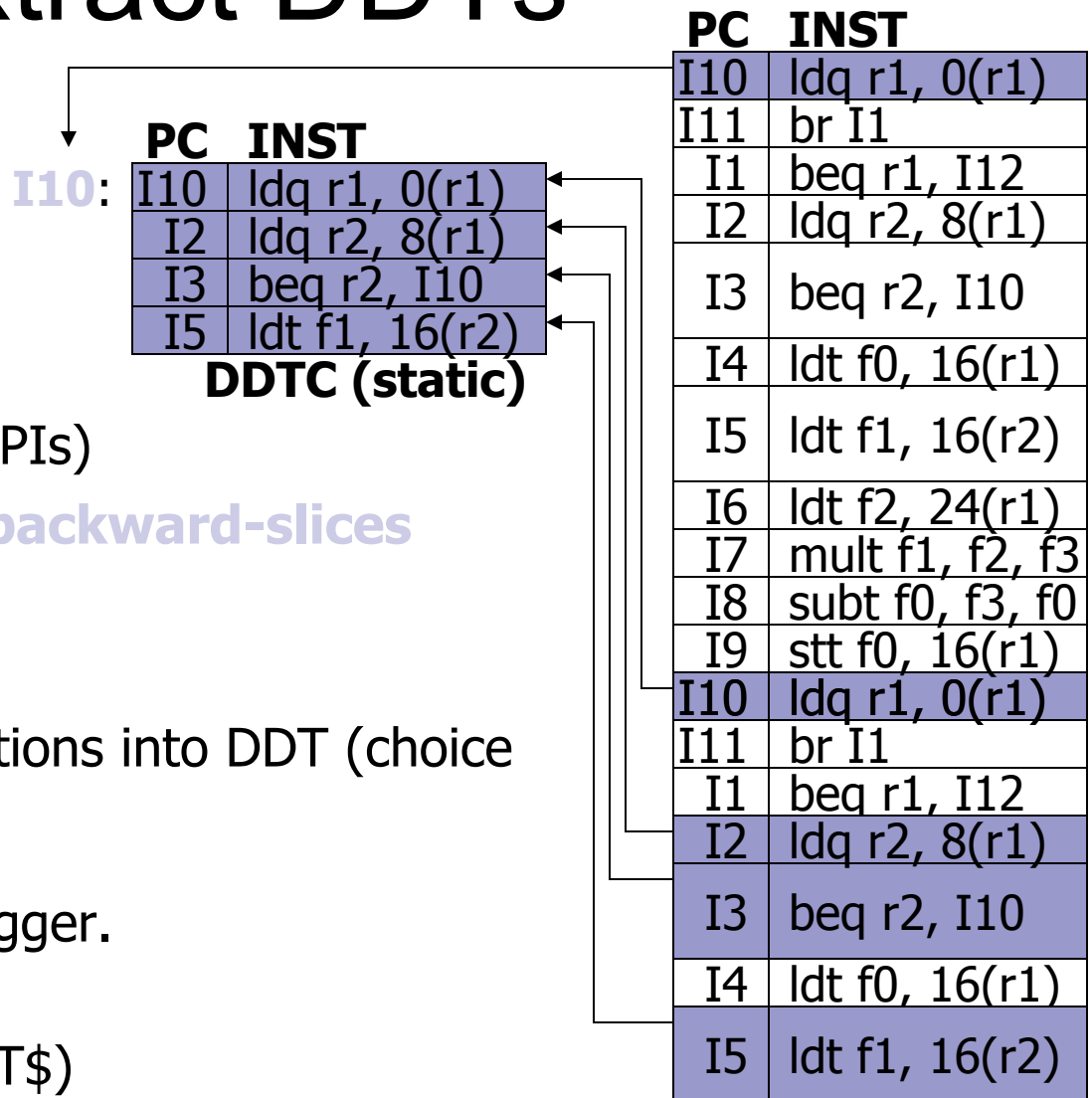
- Use profiling to find PIs

DYNAMIC INSN STREAM

PC	INST
I3	ldq r1, 0(r1)
I5	br I1
I1	beq r1, I12
I2	ldq r2, 8(r1)
I3	beq r2, I10
I4	ldt f0, 16(r1)
I5	ldt f1, 16(r2)
I6	ldt f2, 24(r1)
I7	mult f1, f2, f3
I8	subt f0, f3, f0
I9	stt f0, 16(r1)
I10	ldq r1, 0(r1)
I11	br I1
I1	beq r1, I12
I2	ldq r2, 8(r1)
I3	beq r2, I10
I4	ldt f0, 16(r1)
I5	ldt f1, 16(r2)

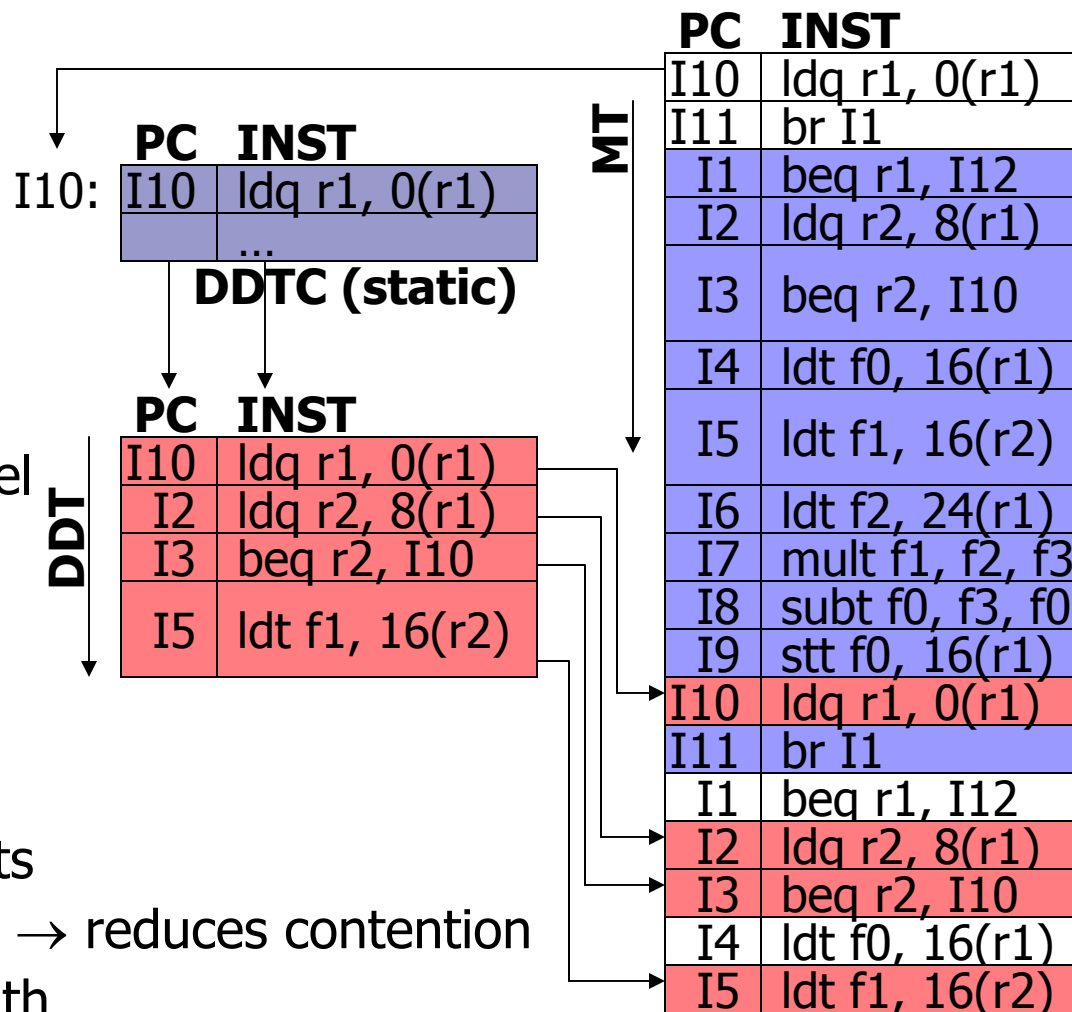
Example Extract DDTs

- Examine program traces
- Start with Problem Insts (PIs)
- Work backwards, gather **backward-slices**
- Pack last N-1 slice instructions into DDT (choice of N a longer topic).
- Use first instruction as trigger.
- Load DDT into DDTC (DDT\$)



Example Pre-Execute DDTs

- Executed a trigger instr?
 - Fork** DDT (μ arch)
- MT, DDT execute in parallel
- DDT initiates cache miss
 - "Absorbs" latency
- MT **integrates** DDT results
 - Instr's not re-executed \rightarrow reduces contention
 - Shortens MT critical path
 - Pre-computed branch avoids mis-prediction



DDMT Performance

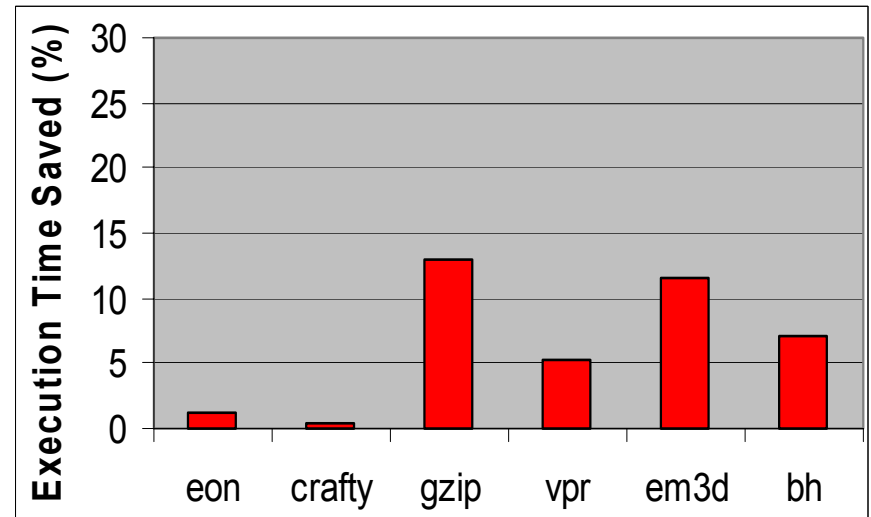
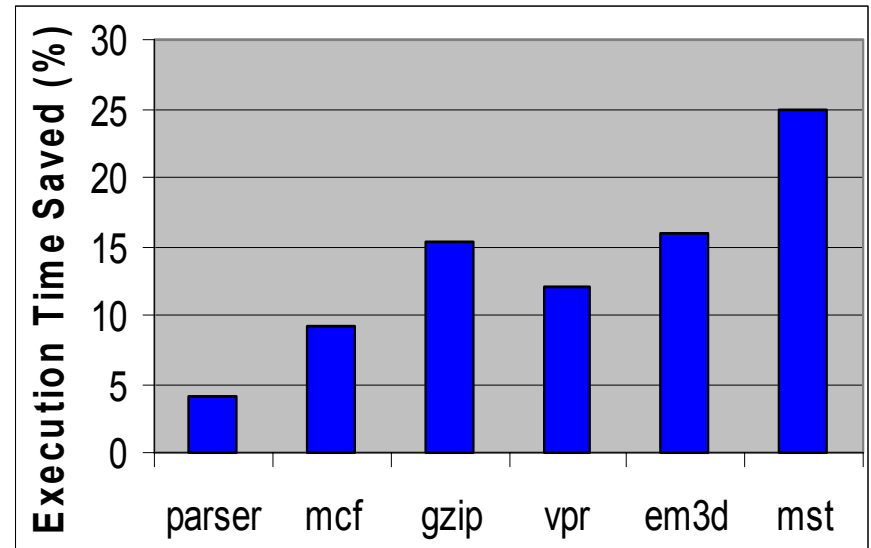
■ Cache misses

- Speedups vary, 10-15%
- DDT “unrolling”: increases latency tolerance (paper)

■ Branch mispredictions

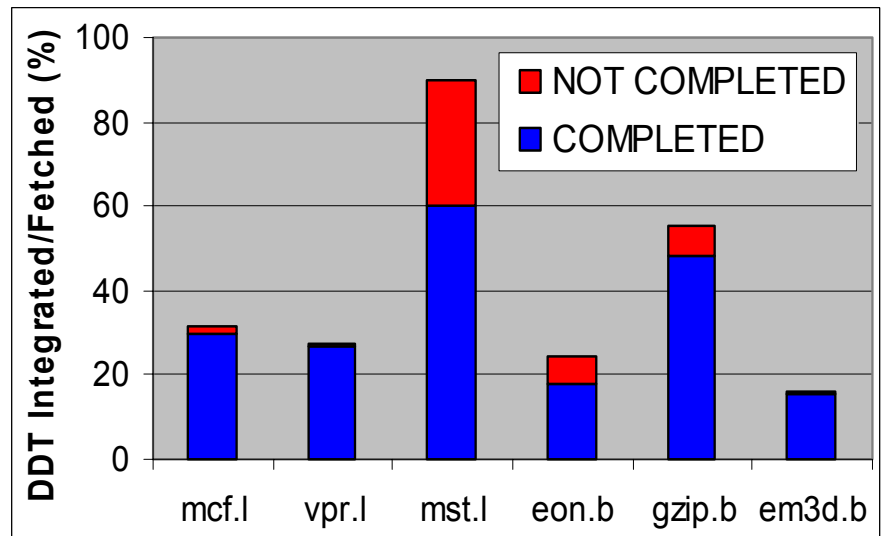
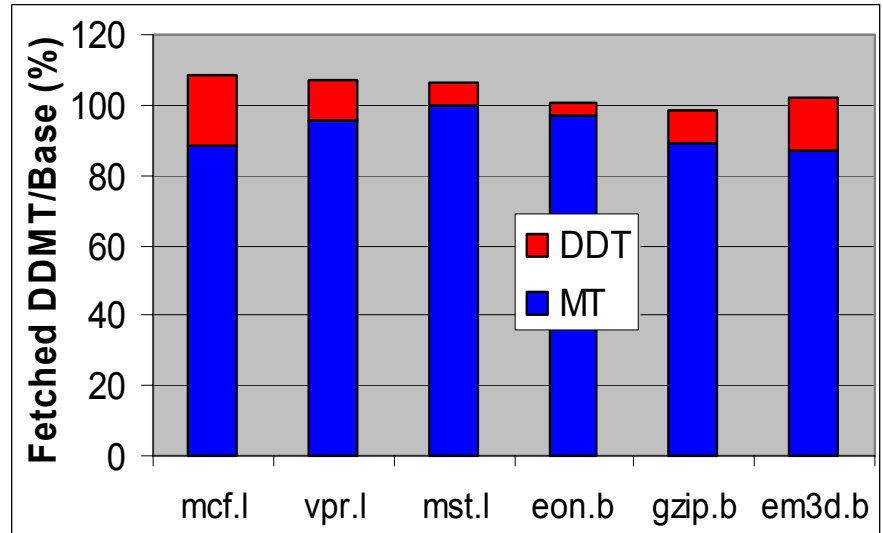
- Speedups lower, 5-10%
- More PIs, lower coverage
- Branch integration
 - != perfect branch prediction

■ Effects mix



More Results

- **DDT overhead:** fetch utilization
 - ~5% (reasonable)
 - Fewer **MT** fetches (always)
 - Contention
 - Fewer **total** fetches
 - Early branch resolution
- **DDT utility:** integration rates
 - Vary, mostly ~30% (low)
 - **Completed:** well done
 - **Not completed:** a little late



DDMT

- Creates pre-execution slices similar to other proposed schemes, with insts from main thread.
- Retains results from helper thread that are valid for main thread, saving time and execution resources.
- Automatically solves the branch correlation problem.

- Cannot trigger a slice early (must have register rename table intact)
- Integration requires exact dataflow match (but prefetching may still happen).
- Cannot pre-execute multiple instances of same instruction in one slice.

Architectural Support for Helper Threading – Summary

- Need architectural support to correlate helper-thread generated predictions with dynamic instances of branches in the main thread.
- Dynamic Speculative Precomputation identifies problem instructions, creates threads, manages threads, all in hardware. Combines advantages of hardware prefetching with thread-based prefetching.
- Register Integration allows some computed values to be reused by the main thread.