The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization

J. Gregory Steffan and Todd C. Mowry

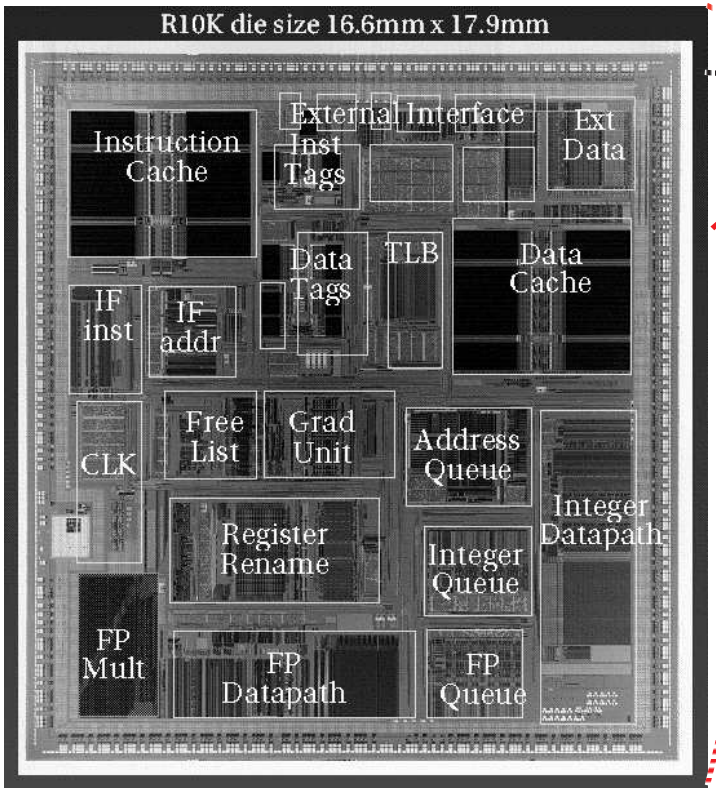
Department of Computer Science

Carnegie Mellon University

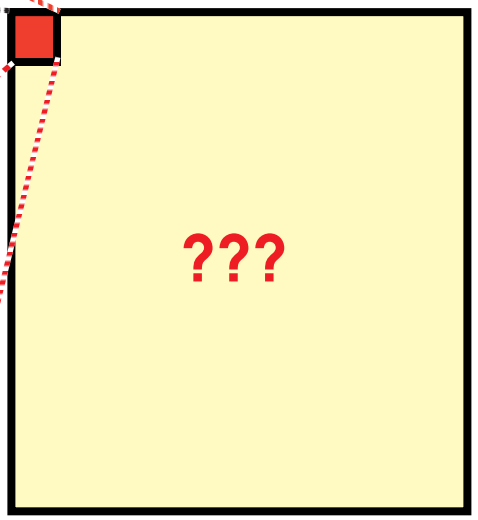
http://www.cs.cmu.edu/~{steffan,tcm}

{steffan,tcm}@cs.cmu.edu

State-of-the-Art vs. Future Processors



MIPS R10000 (6.8 Million Transistors)



In 15 Years
(1 Billion Transistors)

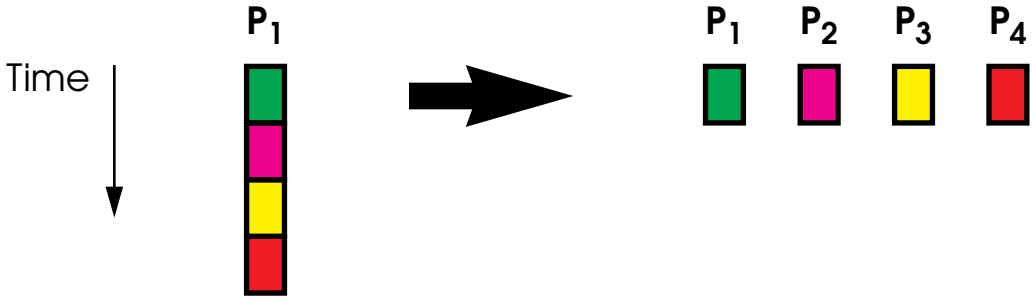
Challenge: *translating these resources into higher performance*

One possibility: multiple processors on a single chip

Performance Benefits of Single-Chip Multiprocessing

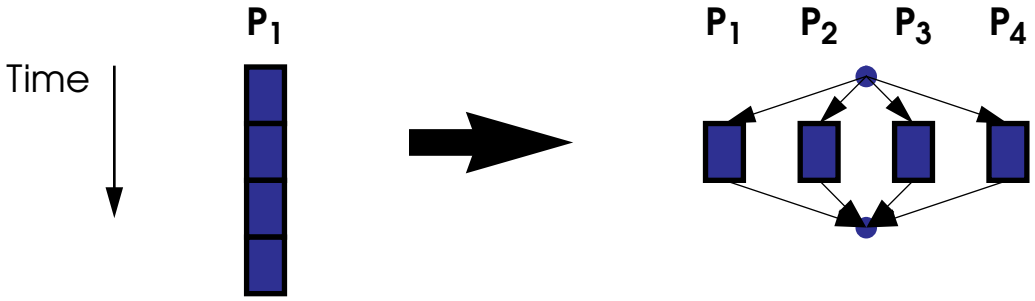
Multiprogramming Workload:

- improved *throughput*



Single Application:

- how to reduce *execution time* => must contain parallel threads



The Big Question:

- ***how do we automatically parallelize all applications?***

State-of-the-Art in Automatic Parallelization

Numeric Applications:

- dominated by regular array references inside loops:

```
FOR i = 1 to N
   FOR j = 1 to N
      FOR k = 1 to N
         C[i][j] += A[i][k] * B[k][j];
```

- significant progress has been made
  - e.g., fastest SPECfp95 number (Bugnion *et al.* 1996)

Non-Numeric Applications:

- access patterns & control flow may be highly irregular
  - pointer dereferences, recursion, unstructured loops, etc.
- little (if any) success in automatic parallelization
- but these applications are important!

☞ ***we must expand the scope of automatic parallelization***

Why Is Automatic Parallelization So Difficult?

Current Approach:

☞ *parallelize only if we can statically prove independence*

```
FOR i = 1 to N
  A[i] += i;
```

Parallel

```
FOR i = 1 to N
  A[i] += f(A[i-1]);
```

Sequential

- transformations can help to eliminate dependences

For Non-Numeric Codes:

☞ *understanding memory addresses is extremely difficult*

```
while (foo()) {
  x = *q;
  ...
  *p = bar();
  ...
}
```

data dependence?

Major Limitation:

☞ *when the compiler is uncertain, it must be conservative*

Expanding the Scope of Automatic Parallelization

The Problem:

- statically proving independence is hopelessly restrictive
  - a full understanding of memory addresses is unrealistic
- instead, we should be focusing on performance issues

Our Solution:

Thread-Level Data Speculation (TLDS)

Overview

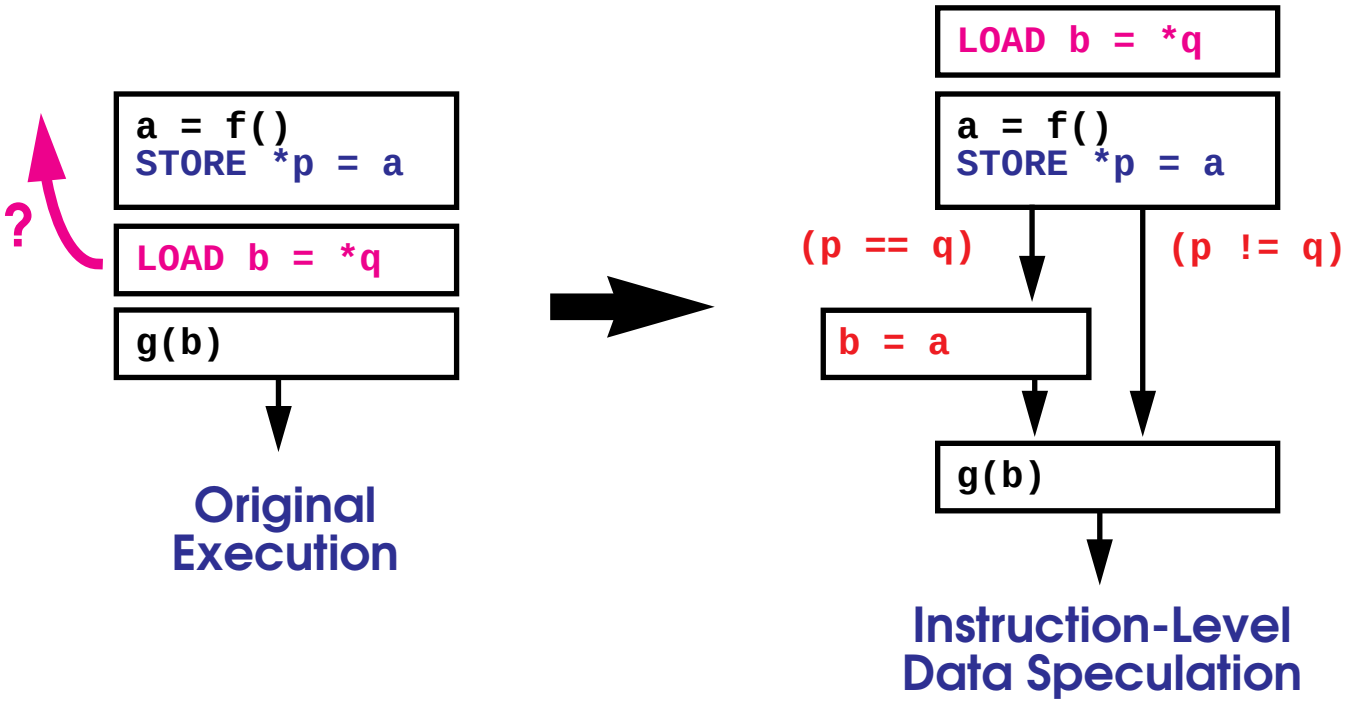
- Thread-Level Data Speculation (TLDS)
- An Example: Compress
- Experimental Results
- Architectural Support
- Conclusions

Data Speculation

Basic Idea:

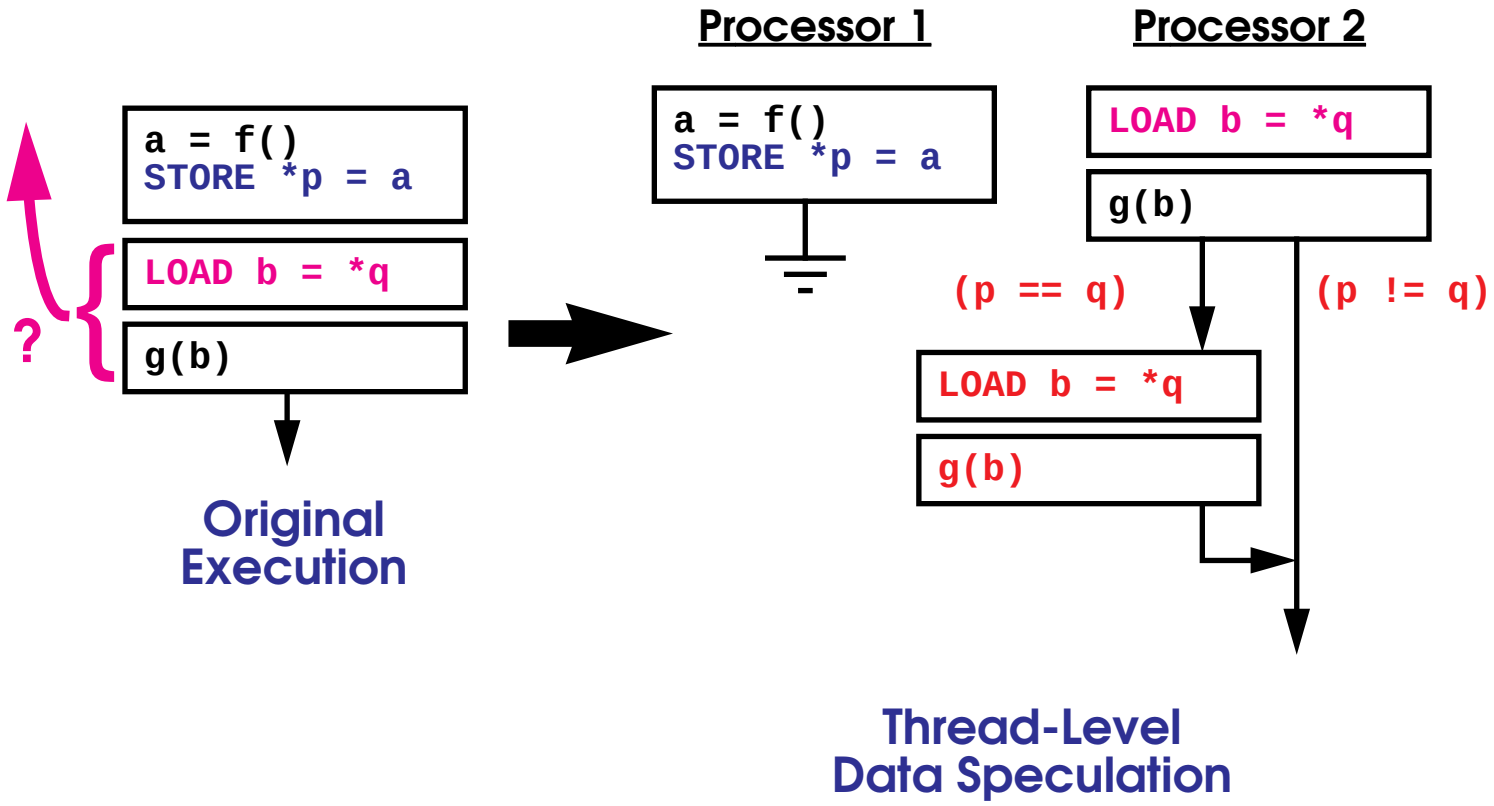
- optimistically perform access assuming no dependence
- if speculation was unsafe, invoke recovery action

Example:



Thread-Level Data Speculation

- **Analogous to instruction-level data speculation**
  - except that it involves separate, parallel threads of control



Related Work

Prior to this Study:

- **Wisconsin Multiscalar Architecture** (Sohi et al, 1995)
  - tightly-coupled ring architecture with register forwarding
  - "ARB" detects memory dependences, hardware rollback

  \+ requires relatively little software support

  \- large, centralized ARB may increase load latency

  \- ring architecture limits flexibility (multiprogramming, locality)

Other Recent Work:

- **Stanford Hydra** (Oplinger et al, 1997)
- **Wisconsin Speculative Versioning Cache** (Gopal et al, 1997)
- **Illinois Speculative Run-Time Parallelization** (Zhang et al, 1998)

Objectives of This Study

- **Does TLDS offer compelling performance improvements?**
  - study of the SPEC92 and SPEC95 integer benchmarks
- **Can we provide cost-effective hardware support for TLDS?**
  - detecting dependence violations
  - recovering from failed speculation

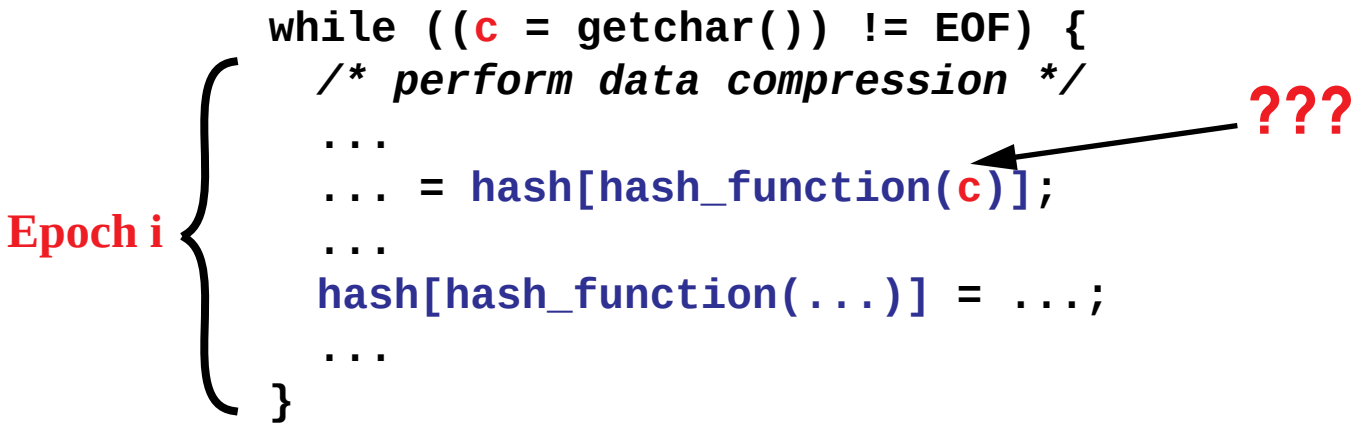
☞ **goal:** ***performance of non-TLDS code is not sacrificed***

- **What compiler support is necessary to exploit TLDS?**
  - optimizations, scheduling, etc.

Overview

- ✓ Thread-Level Data Speculation (TLDS)
- ☞ An Example: Compress
- Experimental Results
- Architectural Support
- Conclusions

An Example: Compress



Potential Source of Parallelism:

- data parallelism across the input stream?

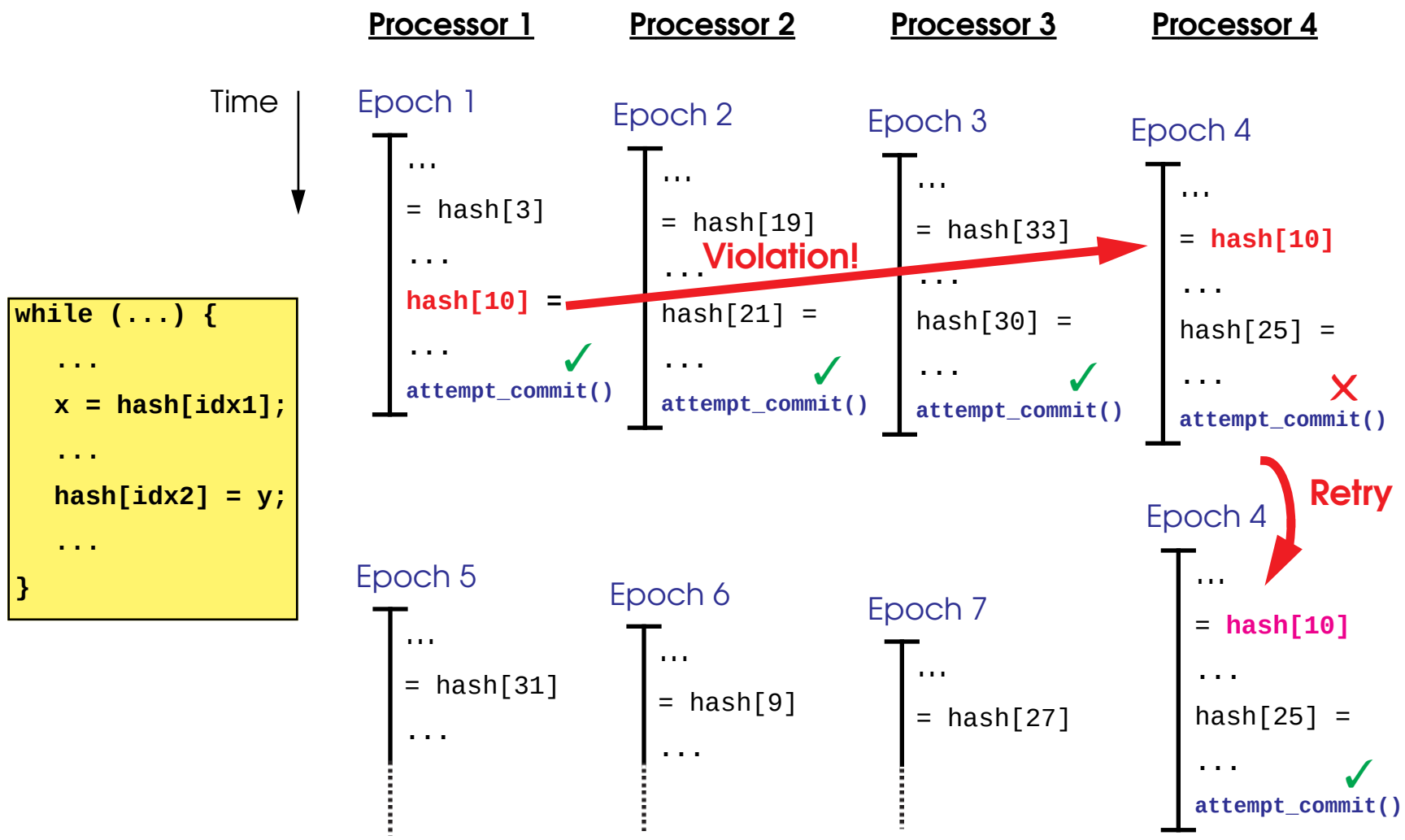
From the Compiler's Perspective:

- hash accesses cannot be statically disambiguated

In Reality:

☞ ***consecutive characters rarely hash to the same entry***

TLDS Execution of Compress



Other Data Dependences in Compress

```
while ((c = getchar()) != EOF) {
  /* perform data compression */
  in_count++;
  ...
  if (...) {out_count++; putchar();...}
  if (free_entries < ...)
     free_entries = ...
  ...
}
```

- **in_count**:
  - induction variable → *implicit in the epoch number*
- **out_count**:
  - reduction operation → *compute partial sums*
- **getchar()**, **putchar()**:
  - *use parallel library routines* (also, `malloc()`, etc.)
- **free_entries**:
  - cannot eliminate dependence → *forward between epochs*

Overview

- ✓ **Thread-Level Data Speculation (TLDS)**
- ✓ **An Example: Compress**
- ☞ **Experimental Results**
  - Relaxing Memory Dependences
  - Forwarding Data Between Epochs
  - Speedups
- **Architectural Support**
- **Conclusions**

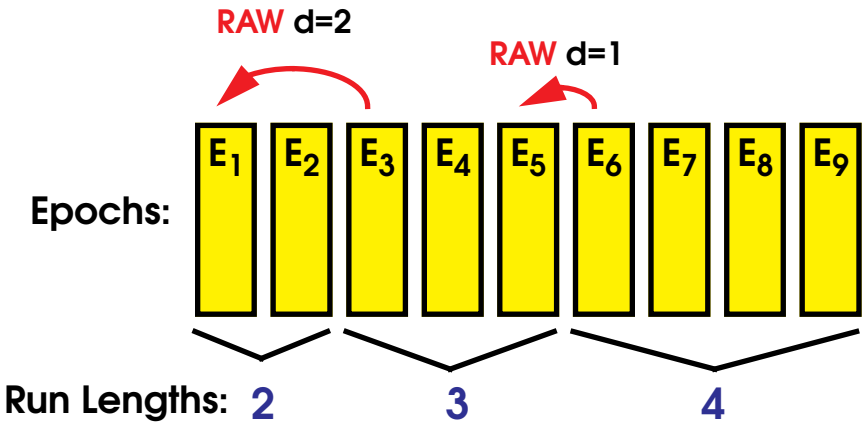
Benchmarks

Suite	Name	Region	Average Dynamic Instrs per Epoch	% of Total Dynamic Instrs
SPEC92	compress	r1	89	99.9
	gcc	r1	1092	8.1
		r2	1593	4.0
	espresso	r1	32	19.4
	li	r1	19	21.9
		r2	286	51.2
	sc	r1	36	69.3
SPEC95	m88ksim	r1	1232	99.3
	ijpeg	r1	9406	15.3
	perl	r1	67	35.8
	go	r1	80	6.8
NAS Parallel	buk	r1	26	16.5
		r2	18	11.4

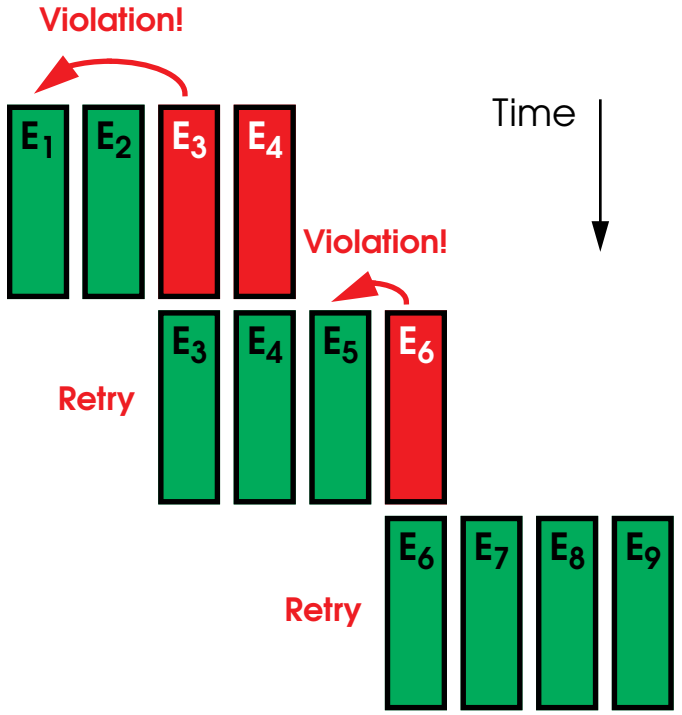
Measuring Memory Dependences: Run Lengths

Run Length:

- # of epochs between Read-After-Write (RAW) dependences



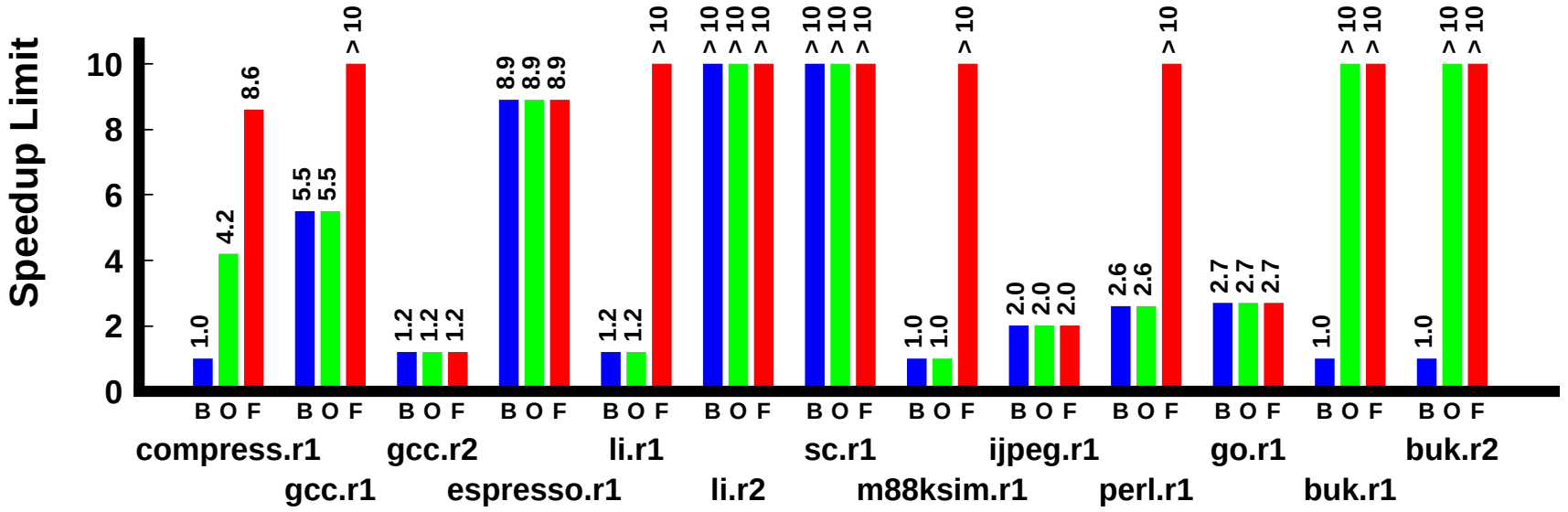
Average Run Length = 3



Execution on 4 Processors

☞ *average run length = rough limit of potential parallelism*

Relaxing Memory Data Dependences



B = base case

O = compiler optimizations applied to remove dependences

F = dependences due to forwardable scalars also removed

- eliminating dependences and forwarding scalars are important
  - ☞ ***significant parallelism is available in many cases***

Forwarding Data Between Epochs

Scalar Memory Values:

- forward if dependences occur frequently
  - synchronization is faster than speculation recovery

☞ ***helpful for performance, not necessary for correctness***

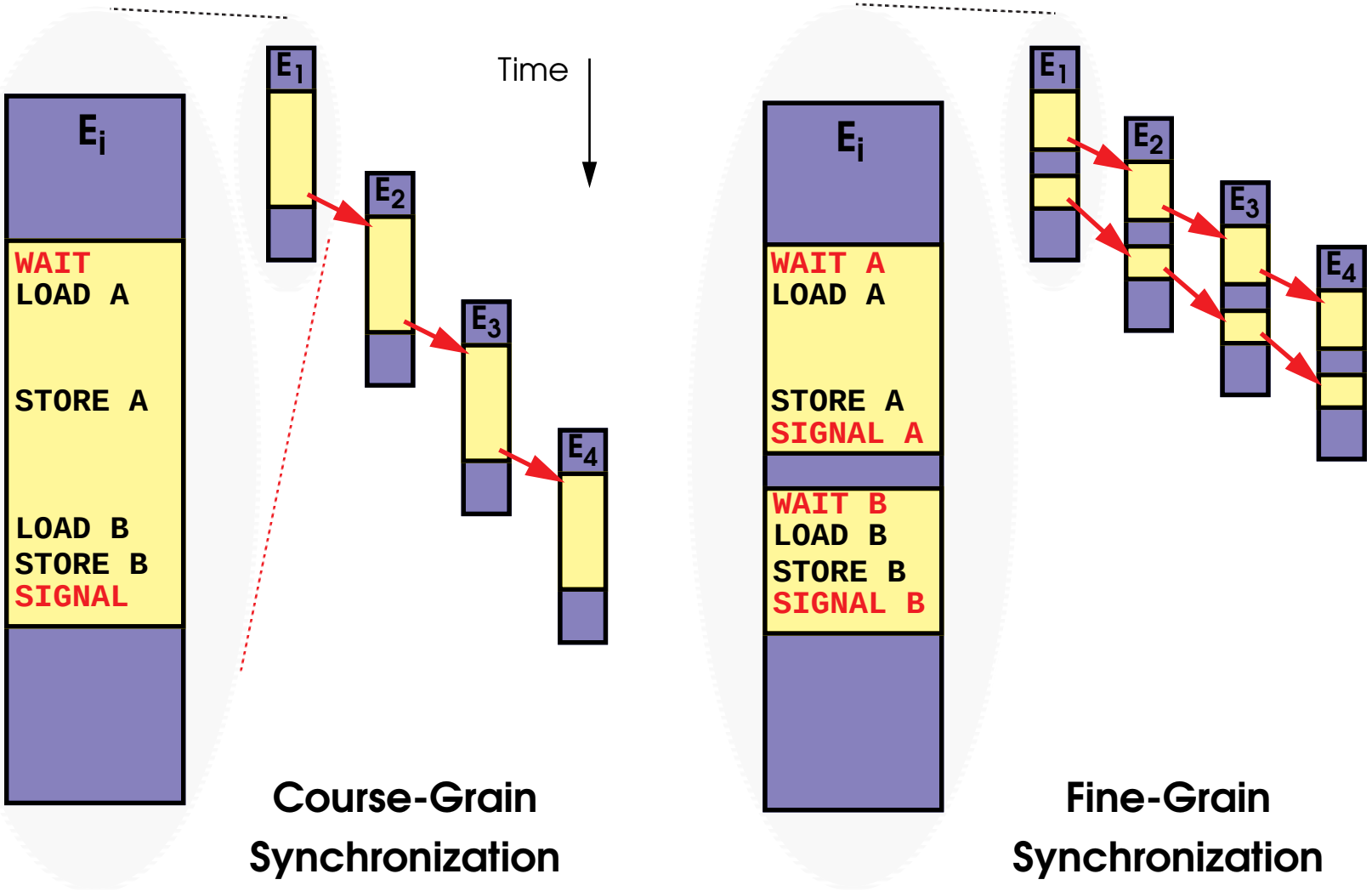
Register Values:

☞ ***must be forwarded to maintain correctness***

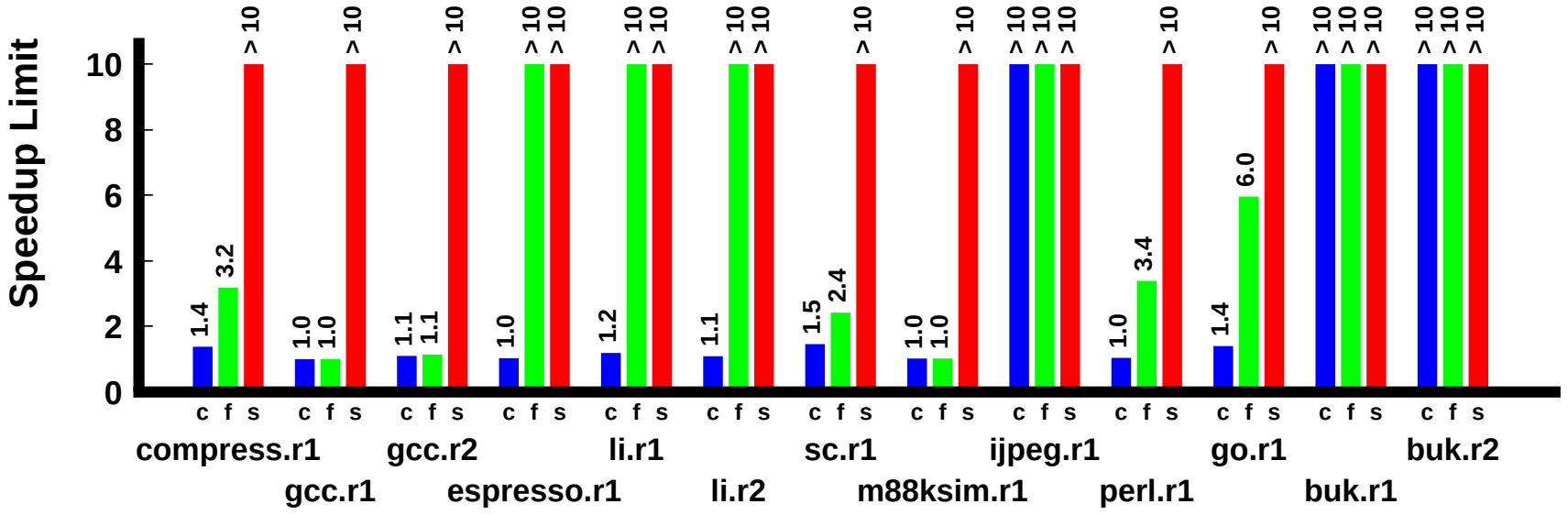
- some register dependences may be eliminated:
  - induction variables
  - through simple loop rescheduling
- all other register dependences forwarded through memory

☞ ***what is the impact on performance?***

Critical Path Lengths



Forwarding Data Between Epochs



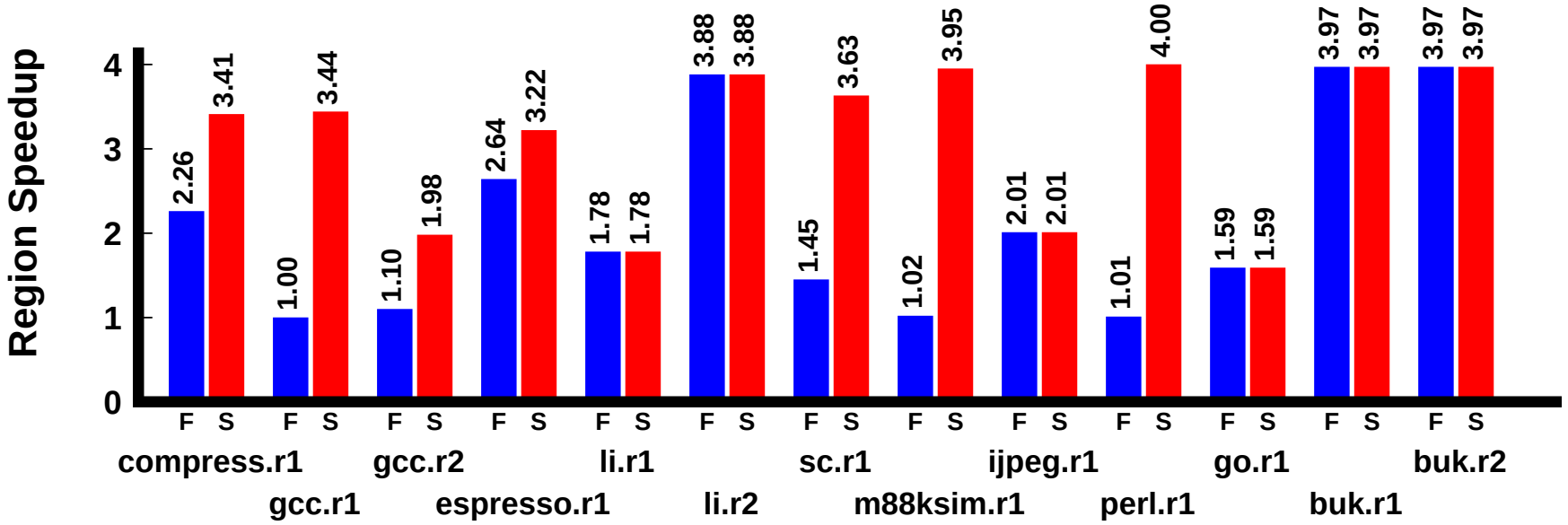
c = coarse-grain synchronization

f = fine-grain synchronization

s = fine-grain synchronization w/ aggressive instruction scheduling

- fine-grain synchronization is helpful
- with aggressive instruction scheduling, forwarding is not a bottleneck

Potential Region Speedup on 4 Processors



F = optimize dependences, forward w/ fine-grain synchronization

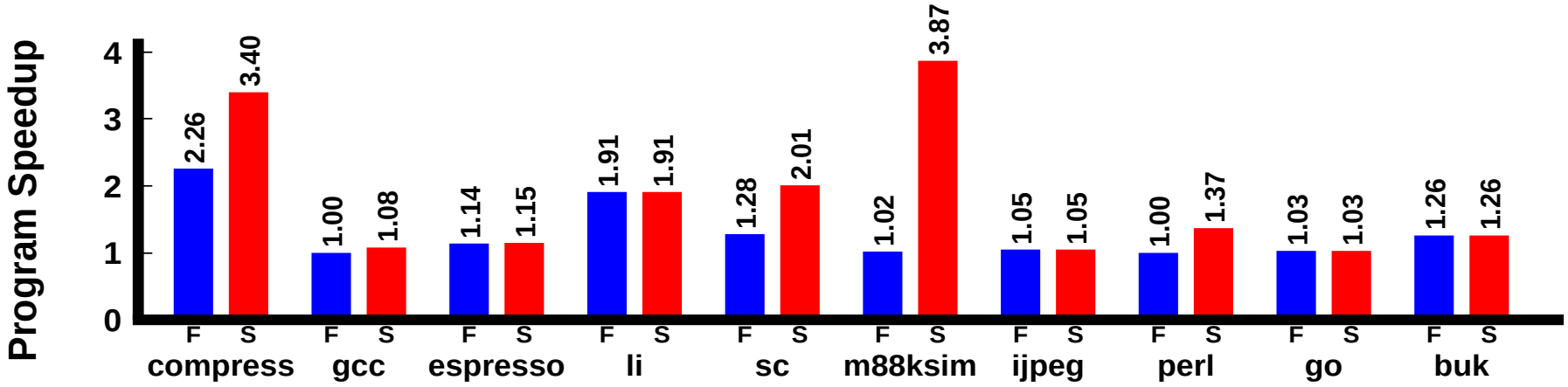
S = also perform aggressive instruction scheduling

communication latency = ***10 cycles***

- aggressive instruction scheduling is a major performance win

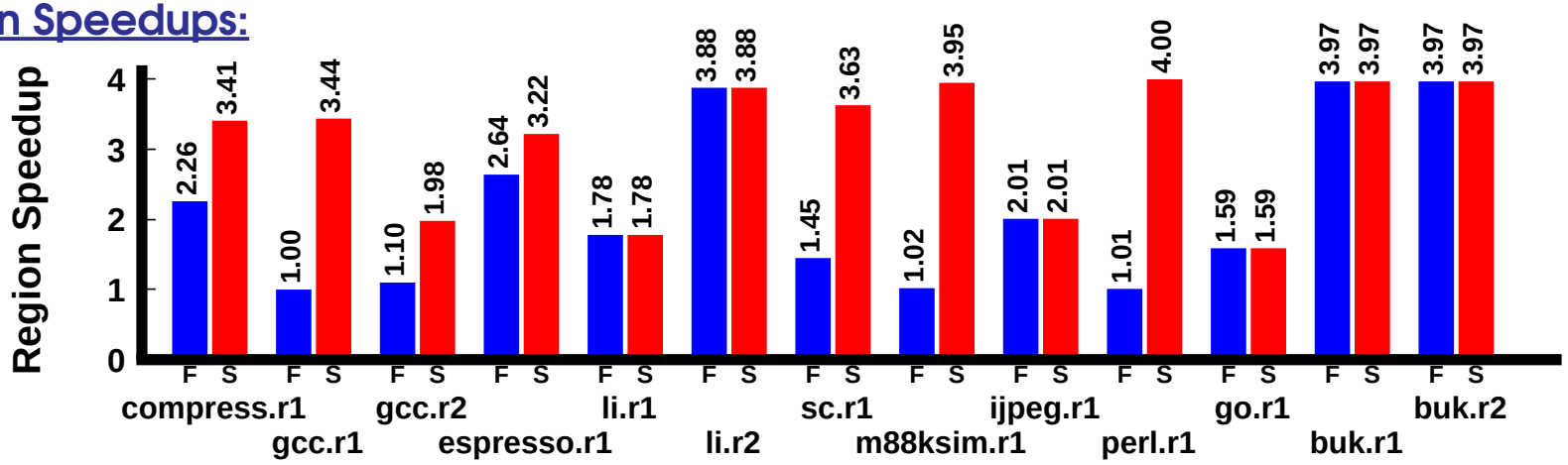
☞ ***potential speedups of twofold or more in 11 of 13 regions***

Program Speedups



Coverage: 99.9% 12.1% 19.4% 73.1% 69.3% 99.3% 15.3% 35.8% 6.8% 27.9%

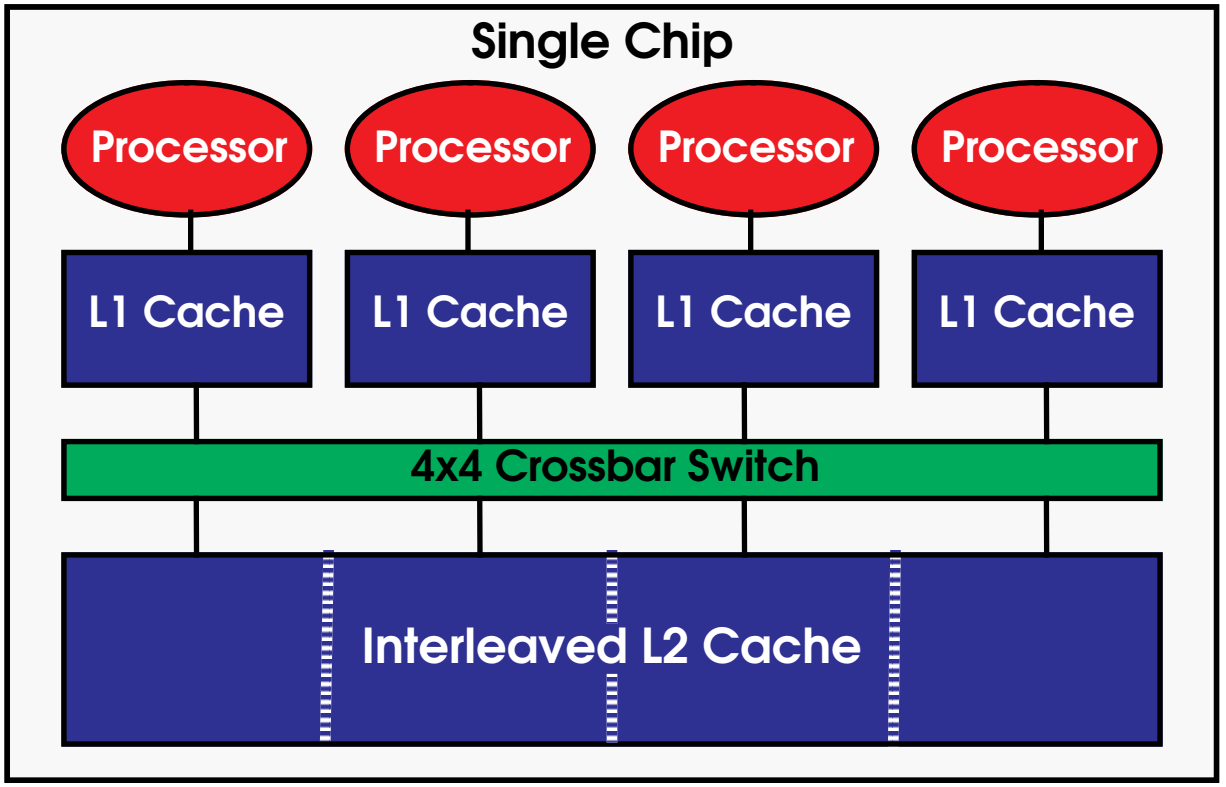
Region Speedups:



Overview

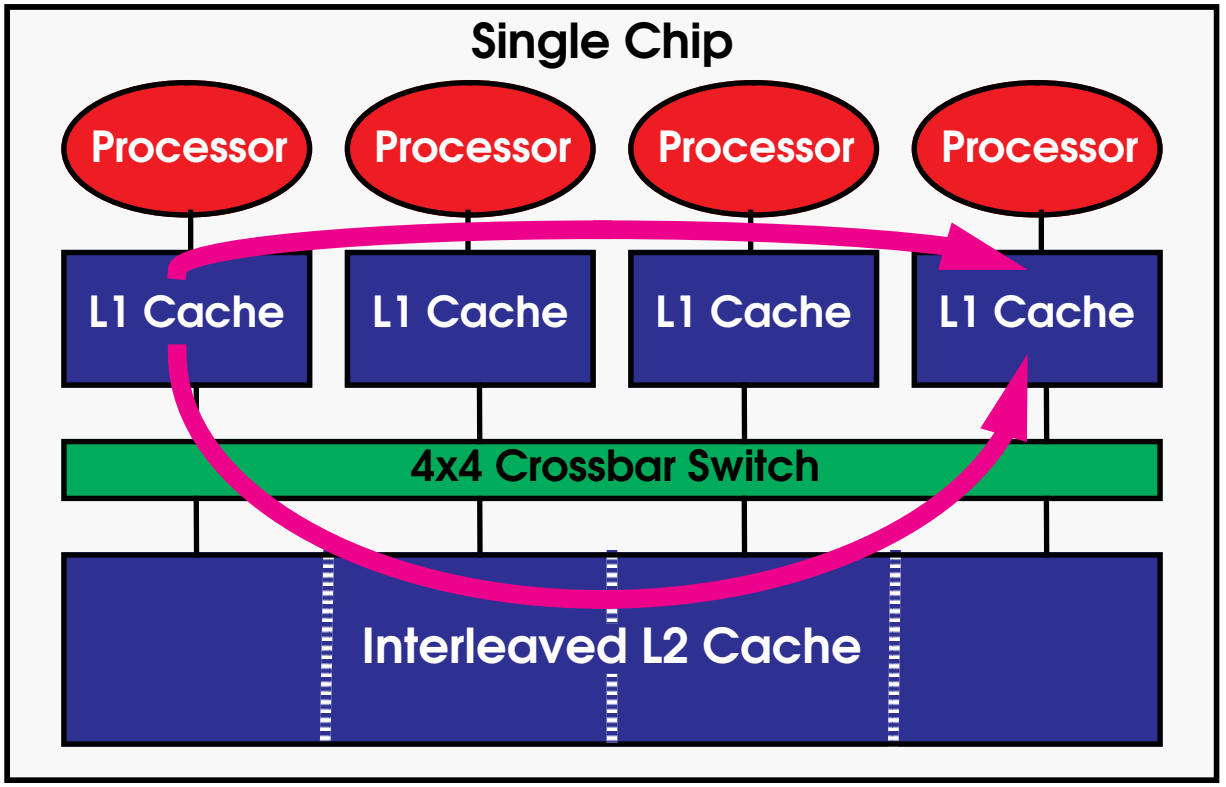
- ✓ **Thread-Level Data Speculation (TLDS)**
- ✓ **An Example: Compress**
- ✓ **Experimental Results**
- ☞ **Architectural Support**
  - Communication Latency
  - Key Architectural Issues
  - Detecting Data Dependence Violations
  - Buffering Speculative State
- **Conclusions**

Base Architecture



- Each processor has its own L1 data cache
  - ☞ *maintain single-cycle load latency*
- L1 caches are kept coherent
  - ☞ *shared-memory programming model*

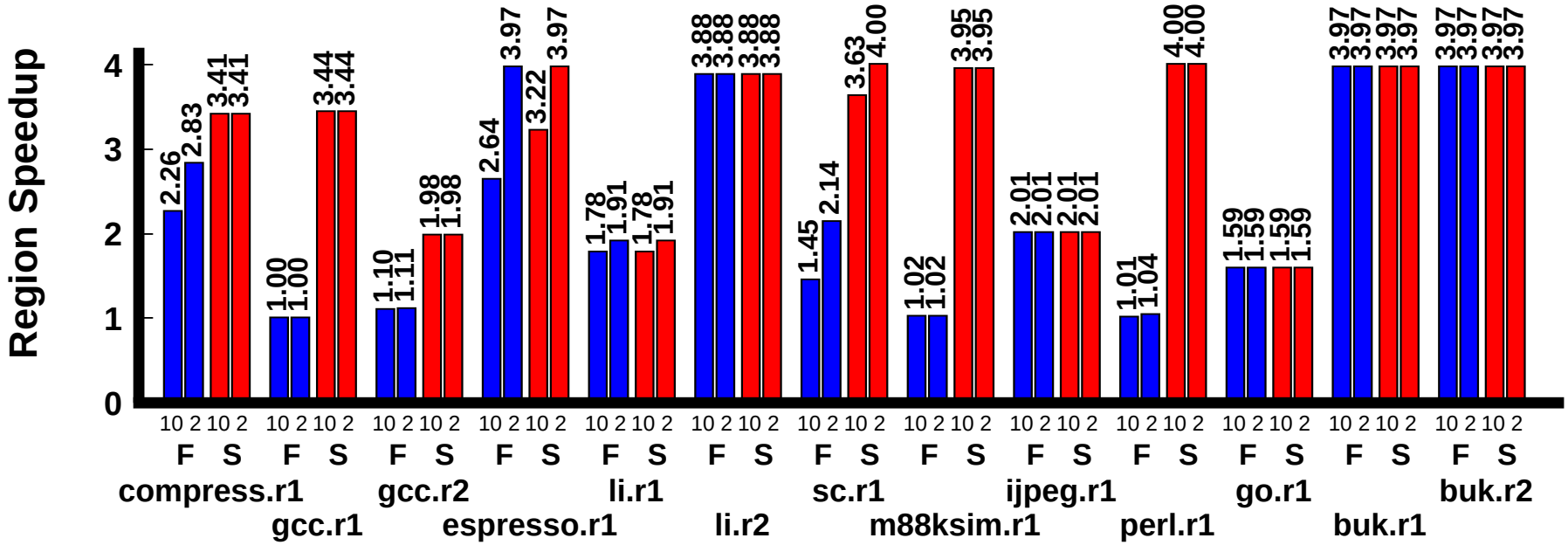
How Important Is Communication Latency?



Some Options:

- direct L1-to-L1 communication → *~2 cycles*
- communicate through the L2 cache → *~10 cycles*

Impact of Communication Latency



F = optimize dependences, forward w/ fine-grain synchronization

S = also perform aggressive instruction scheduling

communication latency: "10" = *10 cycles*, "2" = *2 cycles*

- instruction scheduling reduces the sensitivity to communication latency

☞ ***communicating through the L2 cache is a viable option***

Key Architectural Issues

- **Thread Management**
  - Thread creation and epoch scheduling
  - Epoch numbers must be visible to the hardware
  - Distinguishing speculative vs. non-speculative memory accesses
  - Recovering from data dependence violations:
    - hardware notifies software of violation
    - software performs the bulk of the recovery

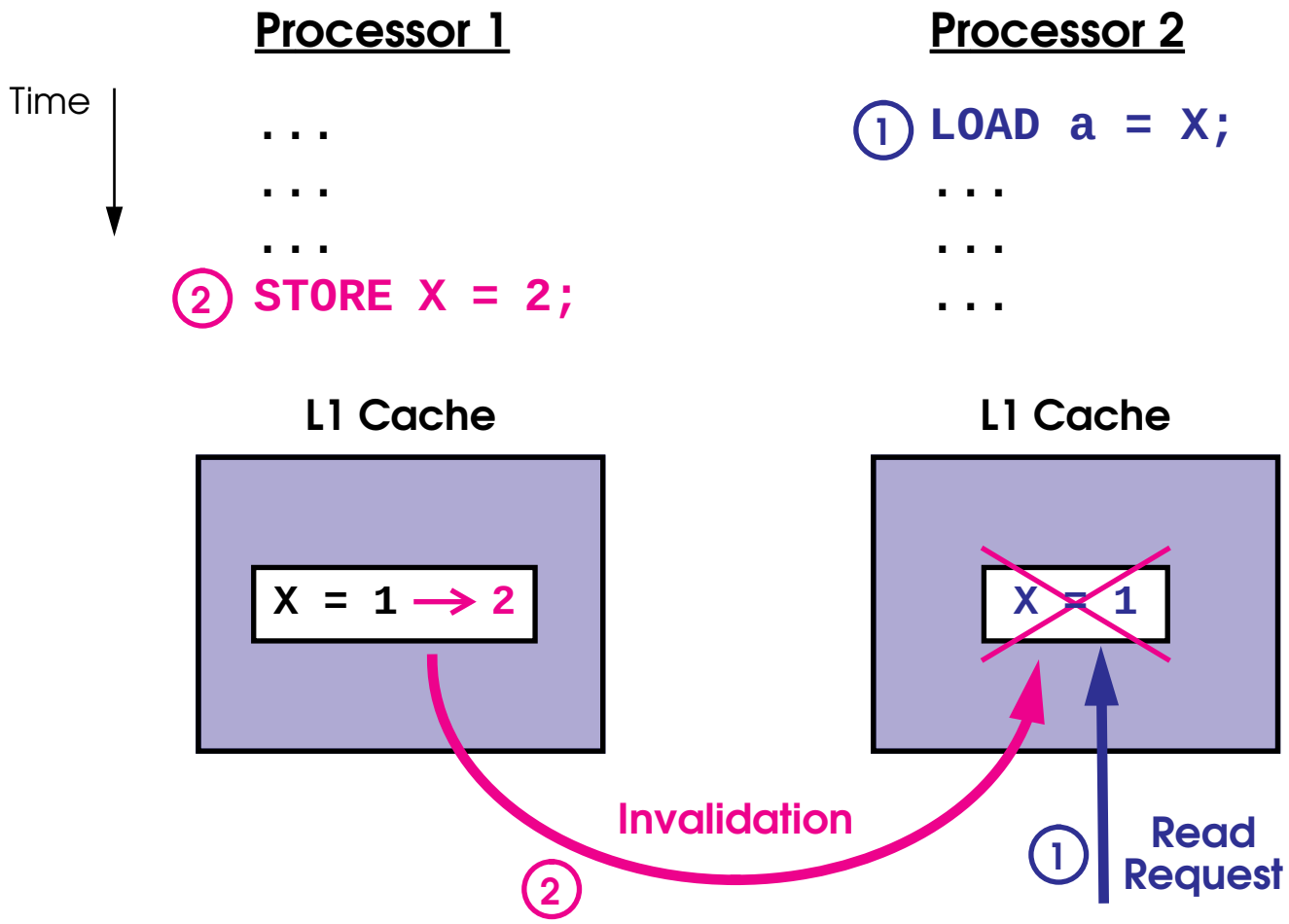
- **Detecting Data Dependence Violations**

  ☞ ***extend invalidation-based cache coherence***

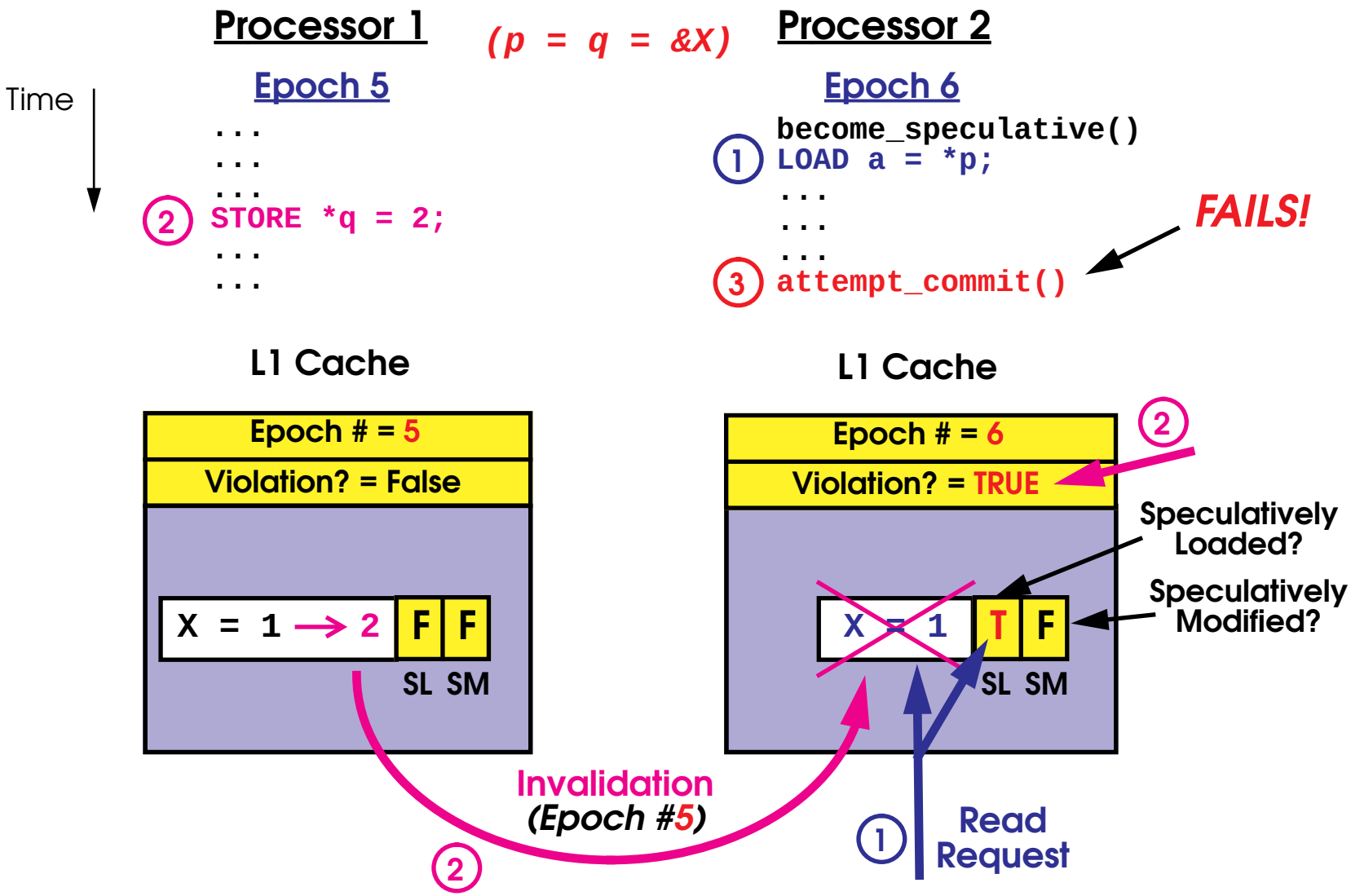
- **Buffering Speculative State**

  ☞ ***extend the functionality of the primary data caches***

Invalidation-Based Cache Coherence



Detecting Data Dependence Violations



Buffering Speculative State

Speculative Stores:

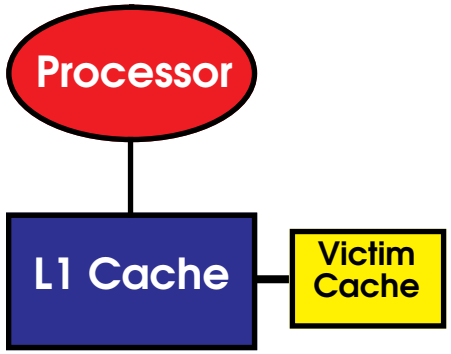
- software cannot realistically roll back memory side effects

☞ **our solution:** *buffer in L1 cache until safe to commit to memory*

Speculative Loads:

- if displaced, then we can no longer track dependence violations
- ***set violation flag upon eviction of a speculatively accessed line***

☞ *correctness is preserved, but performance may suffer*



☞ **a 16KB 2-way set-associative cache with 4 victim entries suffices**

Overview

- ✓ Thread-Level Data Speculation (TLDS)
- ✓ An Example: Compress
- ✓ Experimental Results
- ✓ Architectural Support
- ☞ Conclusions

Conclusions

- **TLDS potentially offers compelling performance improvements**
  - 12 of 13 regions: speedups of 1.78 - 3.97 on 4 processors
  - 7 of 10 programs: speedups of 1.15 - 3.87 on 4 processors
- **Only modest hardware modifications are required**
  - cache coherence protocol augmented to detect violations
  - primary data cache is used to buffer speculative state
- **Compiler support is crucial yet feasible**
  - eliminating data dependences
  - aggressive scheduling to minimize critical path (forwarding)
- **Ongoing and future work**
  - refining the architecture (described in technical report)
  - building the compiler