Hydra

A Chip Multiprocessor with Support for Speculative Thread-Level Parallelism

Lance Hammond

May 31, 2001

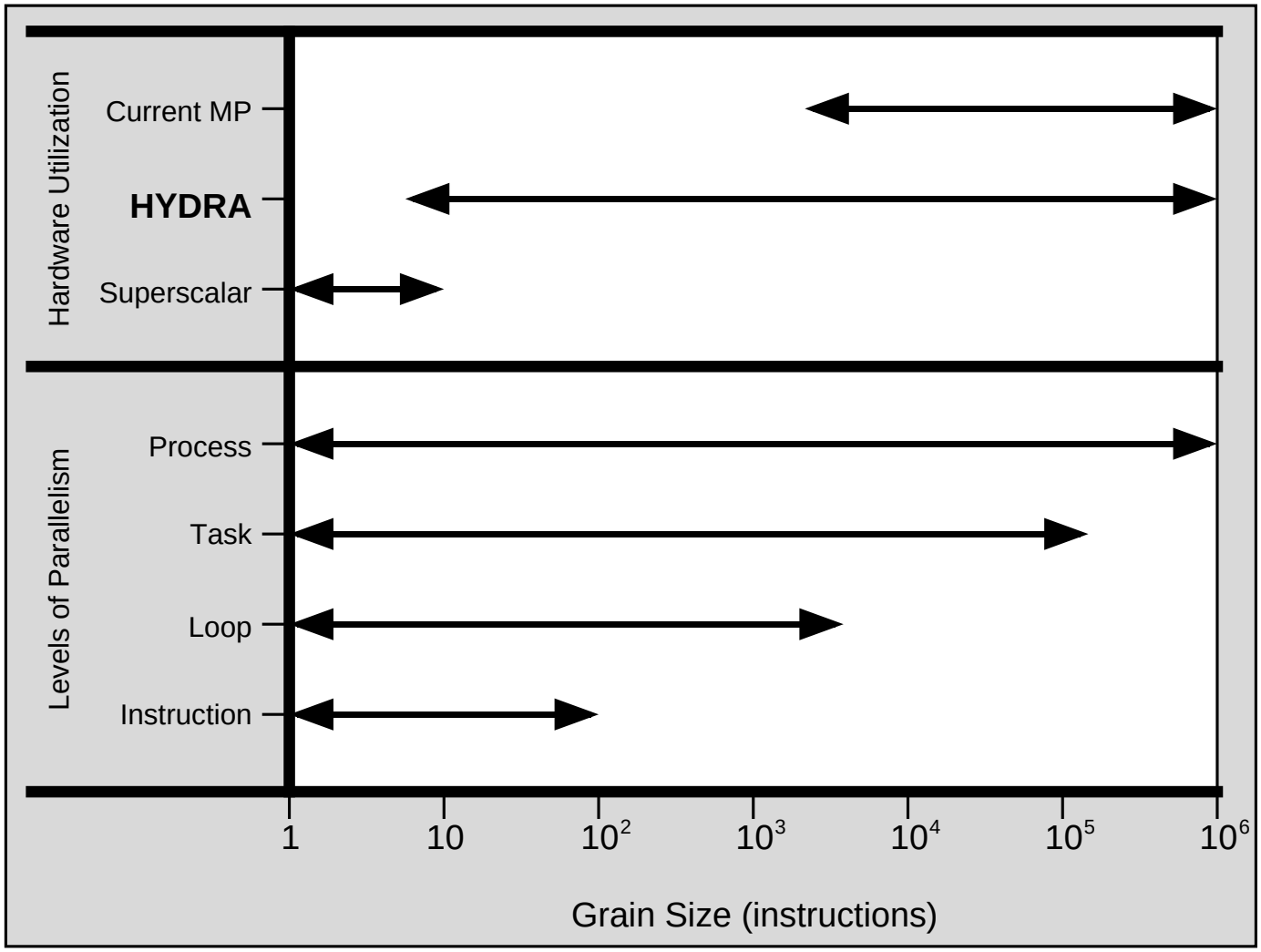
Computer Systems Laboratory
Stanford University
http://www-hydra.stanford.edu

Technology Considerations

- Transistors are plentiful in chip design
  - Moore's Law is still in effect, for now
  - 1 billion transistor chips are on the horizon
  - How should we put all of these transistors to use?
  - But first, what are the limiting factors we need to overcome?
- Wires are becoming "slower"
  - Transistors and simple circuits are getting smaller and faster
  - But designers still want to cross chips with wires
  - However, if wires don't scale with the transistors they effectively become slower over time
- Designer productivity is not keeping up with the transistor supply
  - Current processors are harder to design and especially to *verify*

Parallelism Considerations

- Existing processors limit their ability to use parallelism



What We Want

- A single-chip processor composed of small, relatively independent logic blocks
  - — Keeps low-latency signals localized to short wires
  - — Longer wires between blocks can be latency-tolerant
- Modular, repetitive design
  - — Simplify the design of gargantuan chips
  - — Allow reuse of existing designs as much as possible
- Multiple threads of execution
  - — Exploit many levels of parallelism
- Put all of these concepts together and we get **Hydra: A Chip Multiprocessor** (CMP)

Thesis Contributions

- Design of the Hydra Chip Multiprocessor
- Design of the Hydra Thread-Level Speculation System
  - — Hardware to support transactional memory
  - — Hardware-Software combination to handle threads
  - — API to interface programs with the system
- LESS simulator: a C model of Hydra
  - — Used to develop and refine Hydra and speculation
  - — Used to characterize the performance of Hydra
- Hydra RTL model
  - — Complete model of Hydra with simple processors
  - — Currently being targeted to an FPGA board

Outline

- The Hydra CMP
  - — Design overview
- Thread-Level Speculation (TLS)
  - — Hardware requirements
  - — Software threading system
- Optimization of TLS code
  - — Some basic useful optimizations
- Conclusions

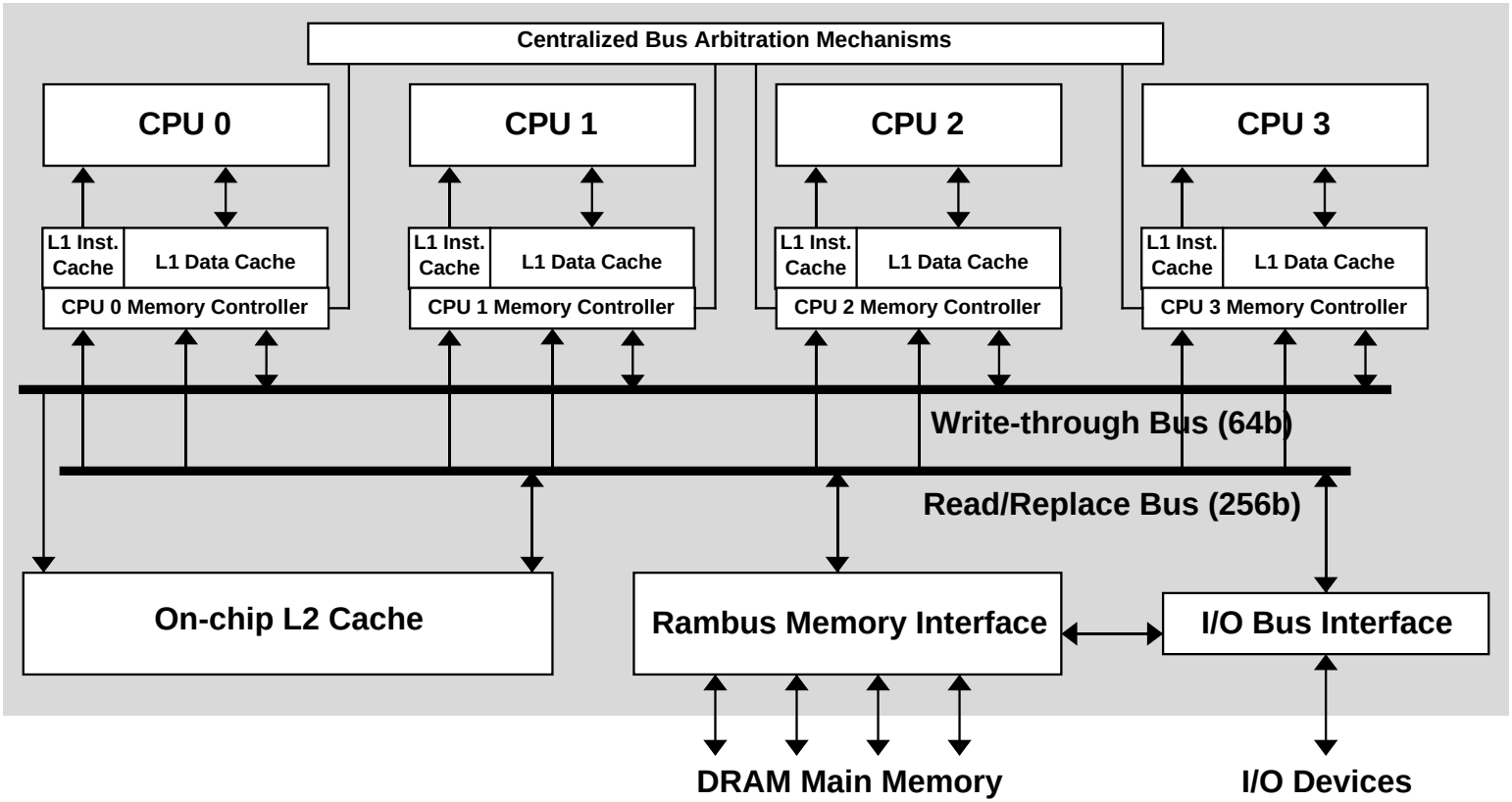
A Chip Multiprocessor

- A CMP offers implementation benefits
  - — Wires: High-speed signals are localized within individual CPUs
  - — Design: A proven CPU design may be replicated across the die
- Allows us to use all levels of parallelism
  - — Individual CPUs can exploit instruction-level parallelism (ILP)
  - — Multiple processors can use *threads* to exploit large-granularity parallelism (TLP)
    - On parallelized programs
    - With multiprogrammed workloads
- Supports fast, on-chip inter-processor communication
  - — Eases parallelization of code
  - — Allows the use of very small threads, closing the gap between TLP and ILP

Previous Work in CMPs

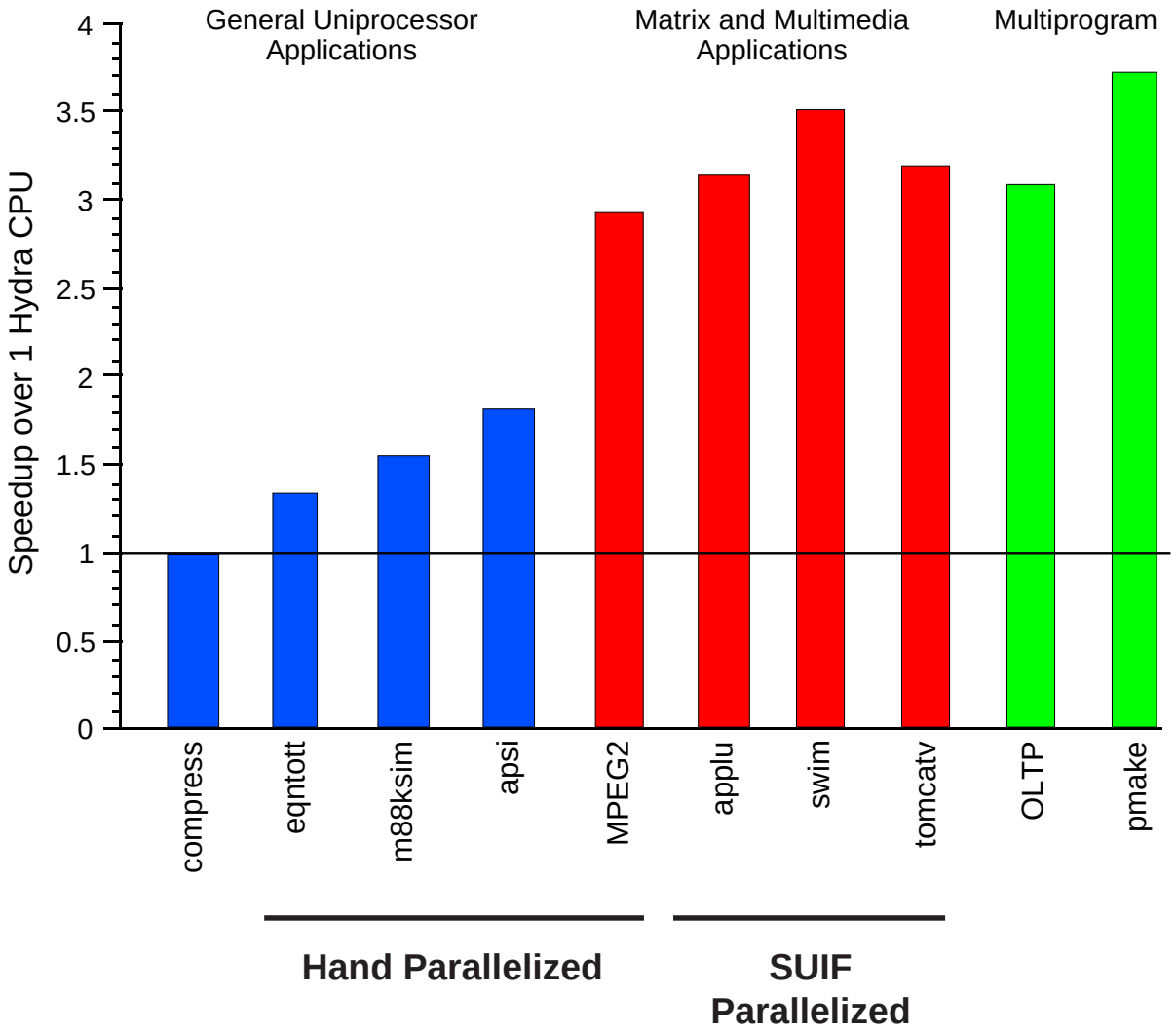
- Pre-Hydra work by our group
  - Analysis of on-chip cache hierarchies
  - Comparison of the effects of interprocessor communication through different levels of the cache hierarchy
  - Comparison of CMP performance with superscalar performance
  - Comparison of CMPs to other single-chip architectures
- Simultaneous multithreading at Washington (Eggers)
  - A “CMP” with all of the “processors” combined together into one huge core executing multiple threads
  - More flexible, so it allows better single-thread performance
  - Much more difficult to implement (Pentium 4, Alpha 21464)

Baseline Hydra Architecture



- Single chip multiprocessor with four processors
- Separate pair of primary caches with each processor
- Write-through data caches to maintain coherence without MESI protocol
- Shared second-level cache
- Supports very low-latency interprocessor communication (~10 cycles)
- Separate read & write-through buses connect everything

Parallel Performance



- Varying levels of performance
  - — Multiprogrammed workloads work well
  - — Very parallel apps (matrix-based FP and multimedia) are excellent
  - — Acceptable only with a few less parallel (i.e. integer) apps

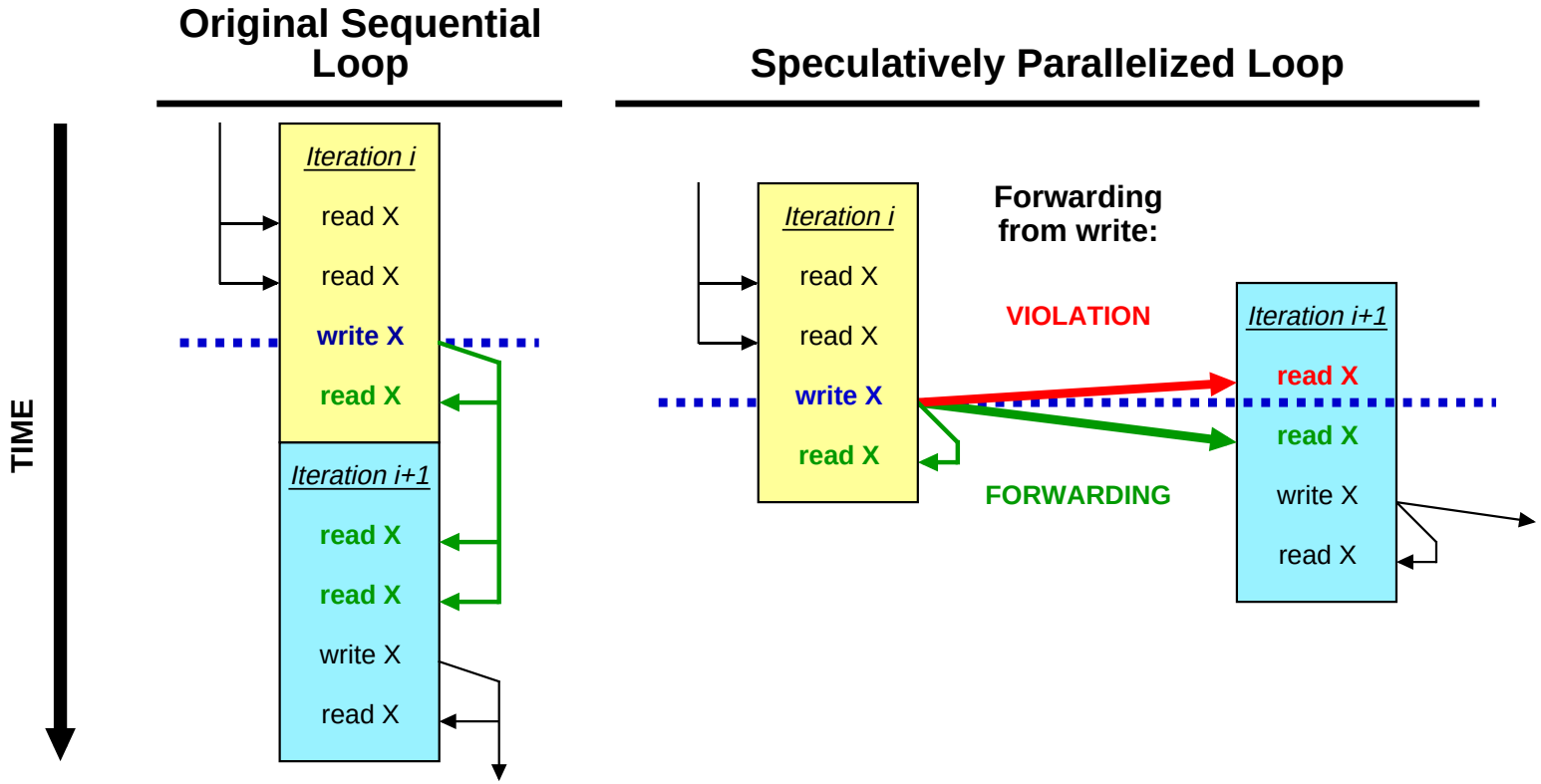
Problem: Parallel Software

- Current parallel software is limited
  - — Some programs just don't have significant parallelism
  - — Parallel compilers generally require dense matrix FORTRAN applications
- Many applications only hand-parallelizable
  - — Parallelism may exist in algorithm, but code hides it
  - — Compilers must *statically* verify parallelism
  - — Data dependencies require synchronization
  - — *Pointer disambiguation* is a major problem for this!
- Can hardware help the situation?

Solution: Thread-Level Speculation

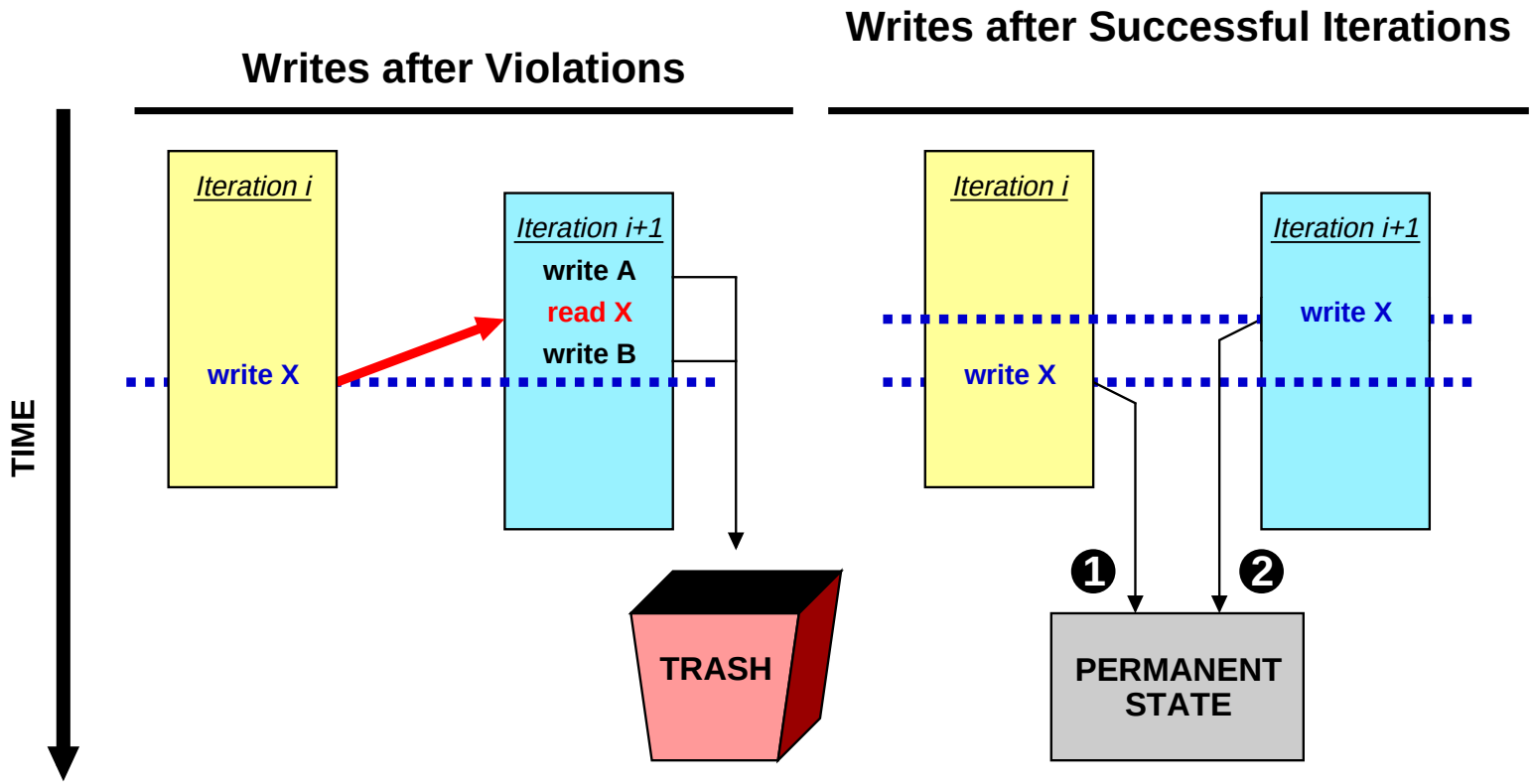
- Thread-level speculation enables parallelization without regard for data dependencies
  - — Normal sequential program is arbitrarily broken up into threads
  - — Speculative threads are now run in parallel on CPUs
  - — Speculation hardware ensures correctness
- Speculation eases the search for parallelism
  - — Loop parallelization is now *easily automated*
  - — More “arbitrary” threads are also possible (such as subroutines)
  - — Parallelizing compilers can be optimistic instead of conservative
  - — Add synchronization only when necessary for performance
  - — Makes manual parallelization much easier
- We just need speculation support mechanisms
  - — 5 memory system requirements to support transactional memory
  - — Speculative thread control mechanism

Memory Requirements I



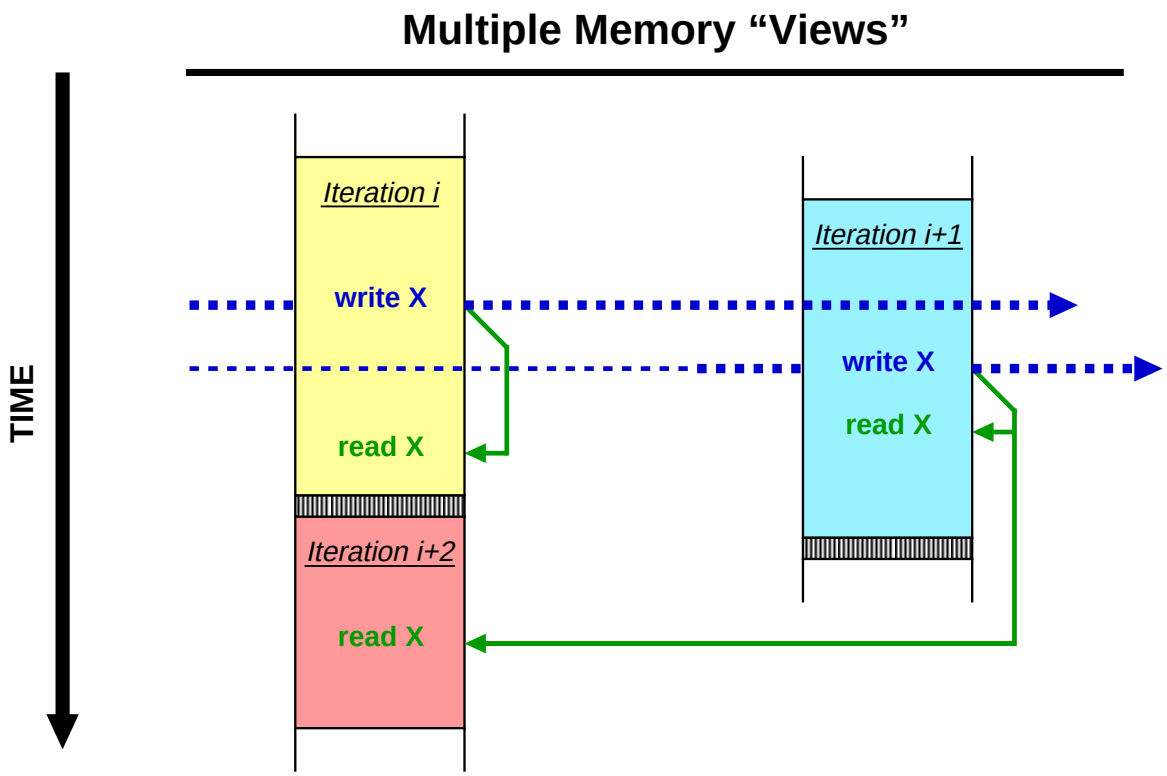
1. Forward data between parallel threads
2. Detect violations when reads occur too early (RAW)

Memory Requirements II



3. Safely discard bad state after a violation
4. Correctly retire speculative state (WAW)

Memory Requirements III

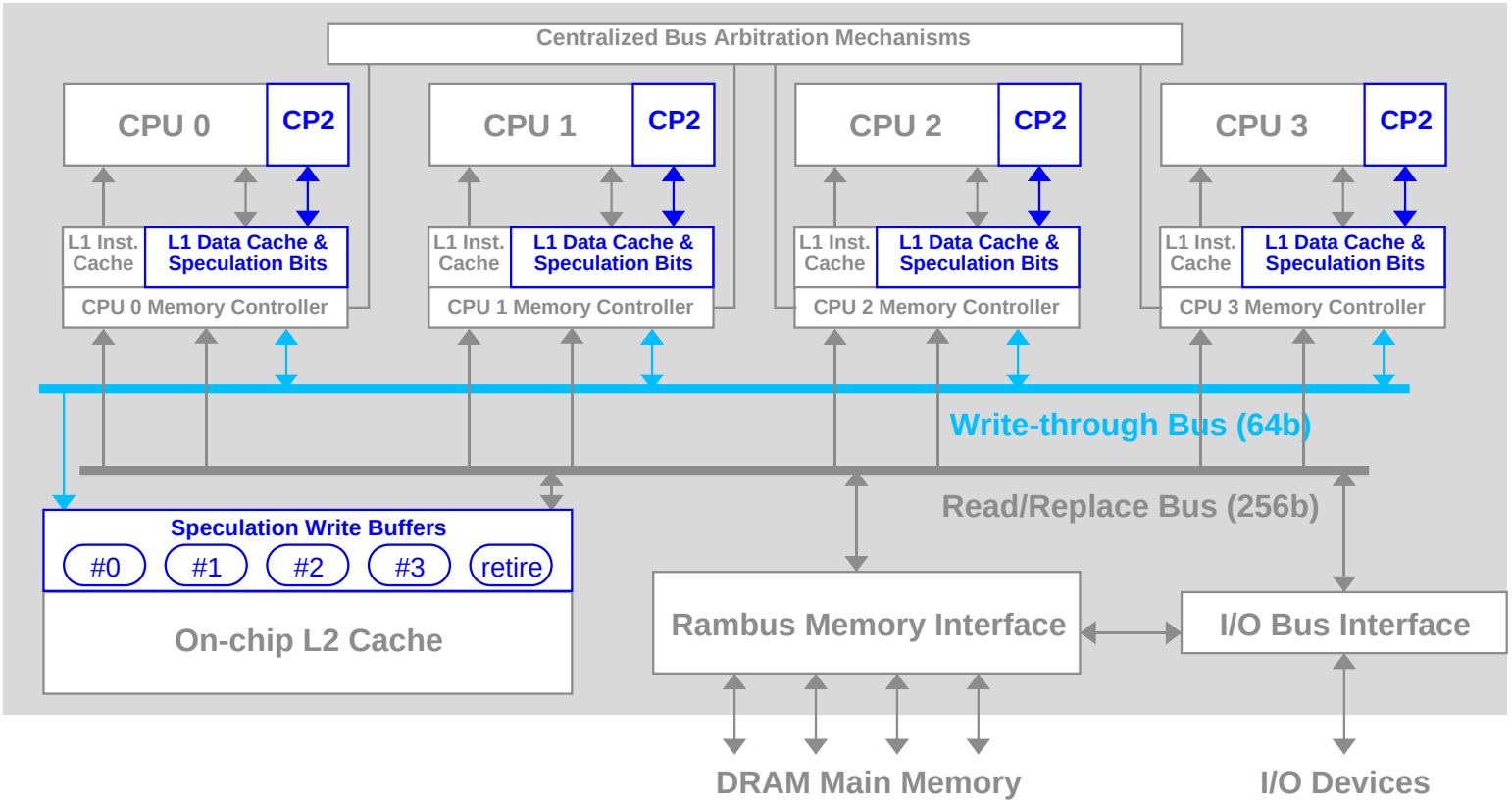


5. Maintain multiple, independently renamed "views" of memory in different processors (WAR)

Previous Work in Speculation

- The Wisconsin Multiscalar (Sohi): Hardware Intensive
  - — Ring of processors, tightly coupled together with ring bus
  - — Hardware monitors both registers *and* memory continuously
  - — Initial design called for unified L1 cache for all processors (ARB)
  - — Later design used a distributed L1 cache with an extremely complex coherence protocol (SVC)
- CMU TLDS (Mowry): Minimal Hardware
  - — Simple CMP, with limited hardware to check for violations in part of memory only periodically
  - — Requires explicit compiler specification of dependent variables
  - — Requires excellent compiler support to maintain performance
- What we want . . .
  - — Limited hardware changes to Hydra, especially in CPU cores
  - — Good performance even without extensive compiler support

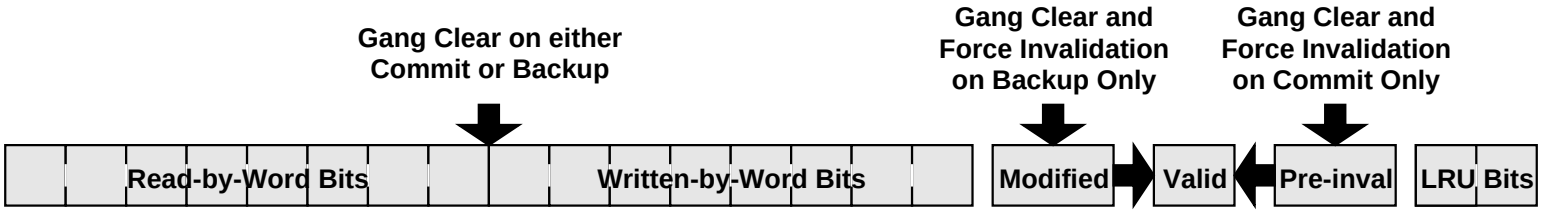
Speculative Hydra Architecture



1. Write bus and L2 buffers provide forwarding
2. Write bus and “Read” L1 tag bits detect violations (RAW)
3. “Dirty” L1 tag bits and write buffers provide backup capability
4. Write buffers reorder and retire speculative state (WAW)
5. Separate L1 caches with pre-invalidation & smart L2 forwarding provide “view” (WAR)

— Speculation coprocessors help to control & sequence threads

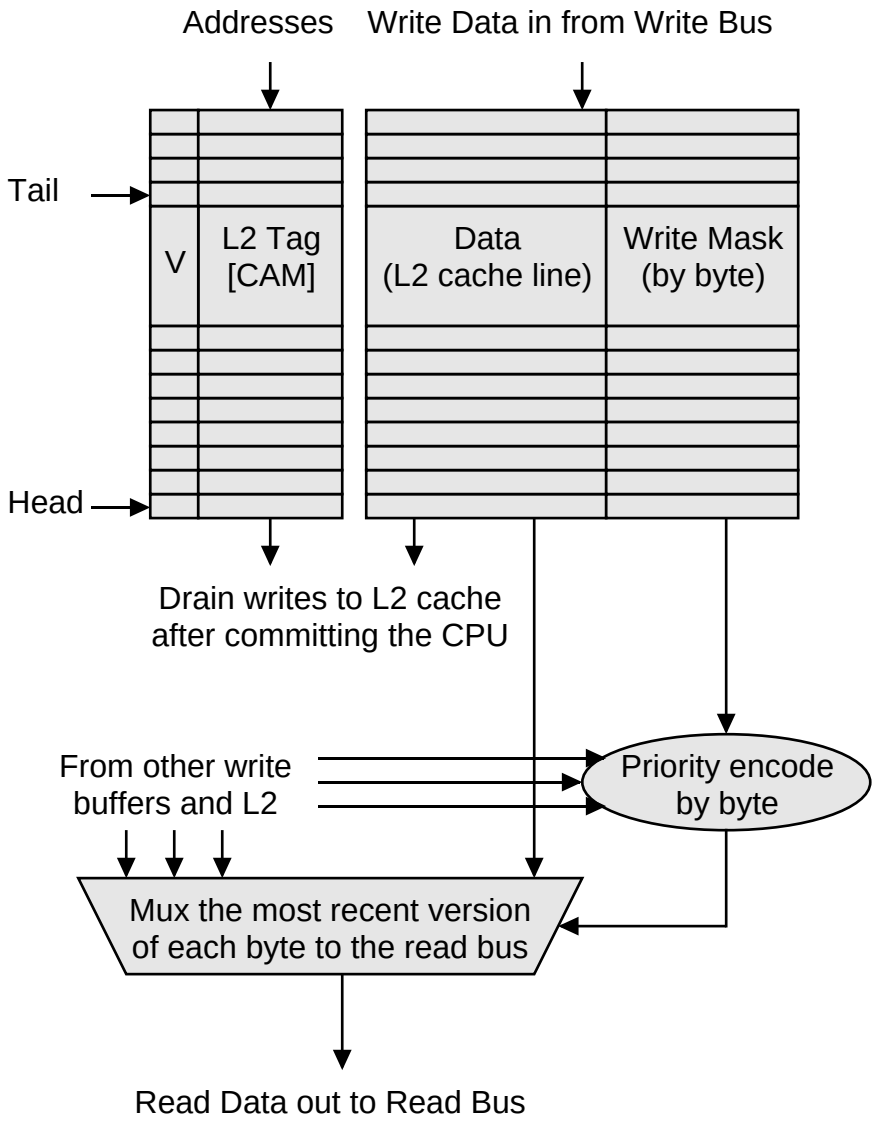
L1 Cache Tag Details



- Speculation requires 4 extra types of bits
  - — **Read-by-word:** Allow violation detection (#2, RAW)
  - — **Written-by-word:** Allow memory renaming (#5, WAR)
  - — **Pre-invalidation:** Allow us to process all pending invalidations as we advance to a new thread (#5, WAR)
  - — **Modified:** Allow us to discard changed lines after violations (#3)
- Special circuits are required in the array
  - — Gang clear of all bits on commits and backups
  - — Set modified bits cause valid bits to clear on backups
  - — Set pre-inval bits cause valid bits to clear on commits

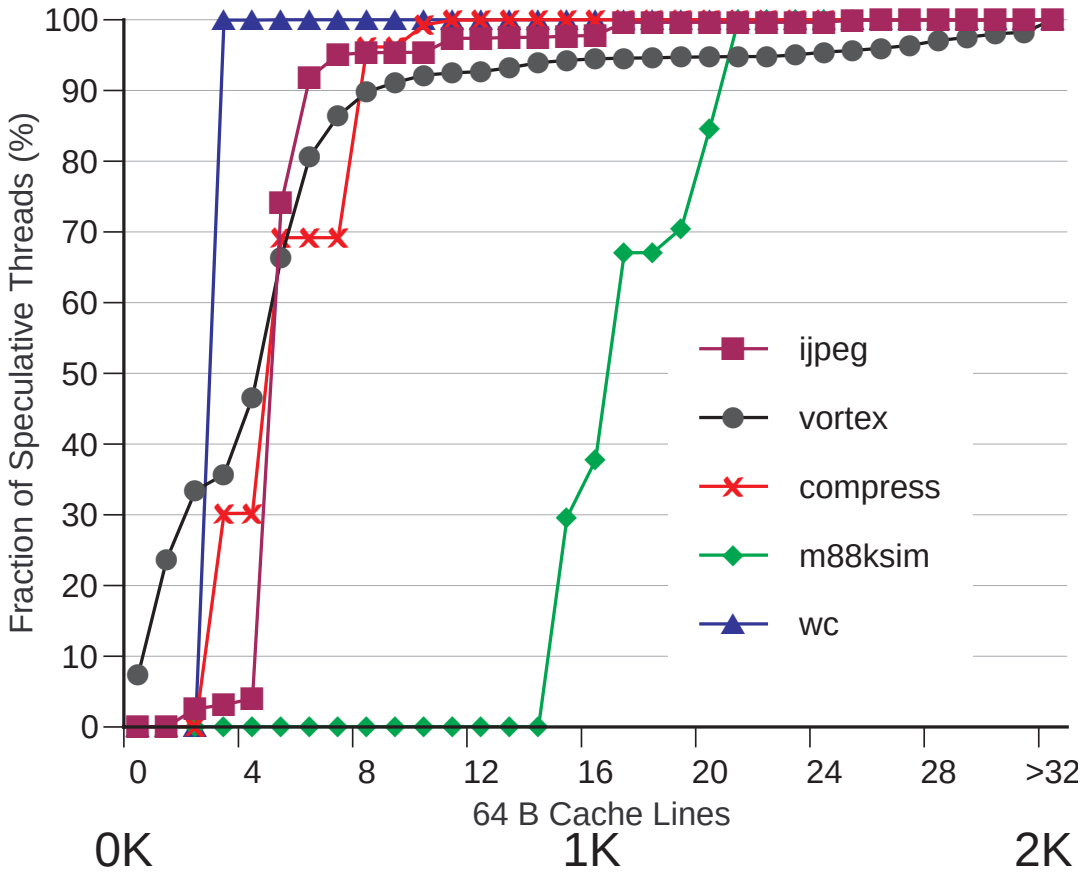
L2 Buffer Details

- Speculative writes are held here until commit time
  - — Collected by cache line
  - — CAM tag array, tail pointer
  - — Byte write mask for each line
  - — Drains into L2 when complete
- Reads are tricky
  - — Line read from L2 cache
  - — Data from any “earlier” buffers is substituted, if present
  - — Requires byte-by-byte priority encoding & muxing to select most “recent” bytes



L2 Data Buffering

- Results show that small buffers are sufficient
  - We used a fully associative line buffer
  - < 1 KB per thread captures most writes (total: about a pair of L1s)
  - Larger threads generally were poor speculative targets



Speculative Thread Control

- Thread control can be implemented in different ways
  - — TLDS: All control done with code added by a compiler
  - — Multiscalar: Compiler marks threads, hardware sequences
  - — Illinois (Torrelas): Threads found *and* sequenced by hardware
  - — Hydra: Similar to TLDS, but with additional hardware support
- Our core is assembly-language software handlers
  - — Control speculative threads
    - By issuing commands to the speculative memory hardware using CP2
    - By communicating with other processors & L2 using special stores
    - By maintaining thread state in registers
    - By maintaining speculative memory structures
  - — Some handlers called directly by modified user code
  - — Others are invoked as exception handlers (violations!)
  - — Software is more flexible than hardware but adds overhead
- Works with speculative hardware to sequence...

Subroutine Speculation

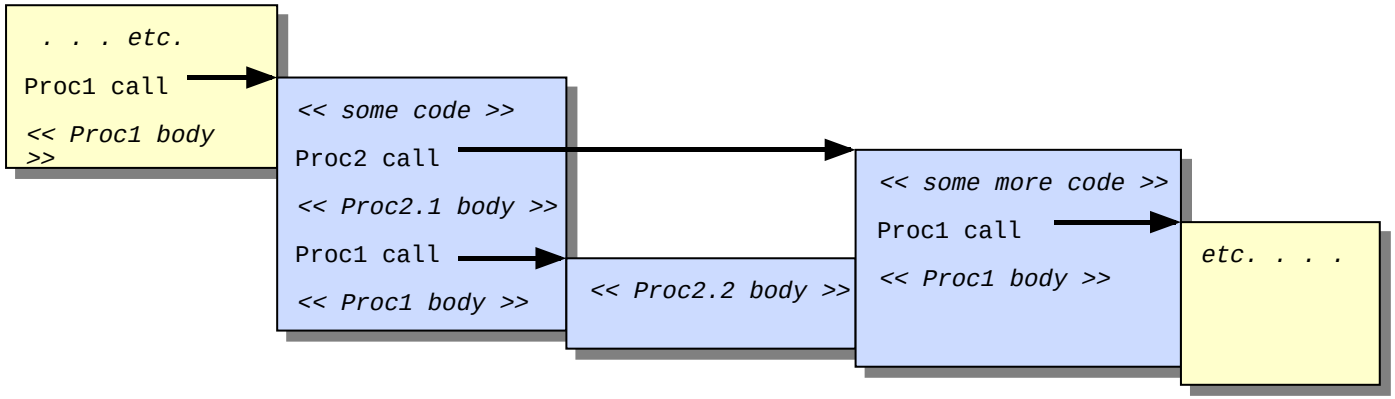
Original Program

```
void Proc1()
{
     << Proc1 body >>
}

void Proc2()
{
     << Proc2.1 body >>
     Proc1();
     << Proc2.2 body >>
}
```

```
 . . . etc.
Proc1();
<< some code >>
Proc2();
<< some more code >>
Proc1();
etc. . . .
```

Speculative Threads



— Post-subroutine call continuations are dynamically forked off by speculation software as calls are made

— Requires subroutines with VOID or predictable return values

- Software handlers generate & check return predictions

— Speculation hardware prevents errors from side effects

— Complex handlers are expensive to implement in software

Loop Iteration Speculation

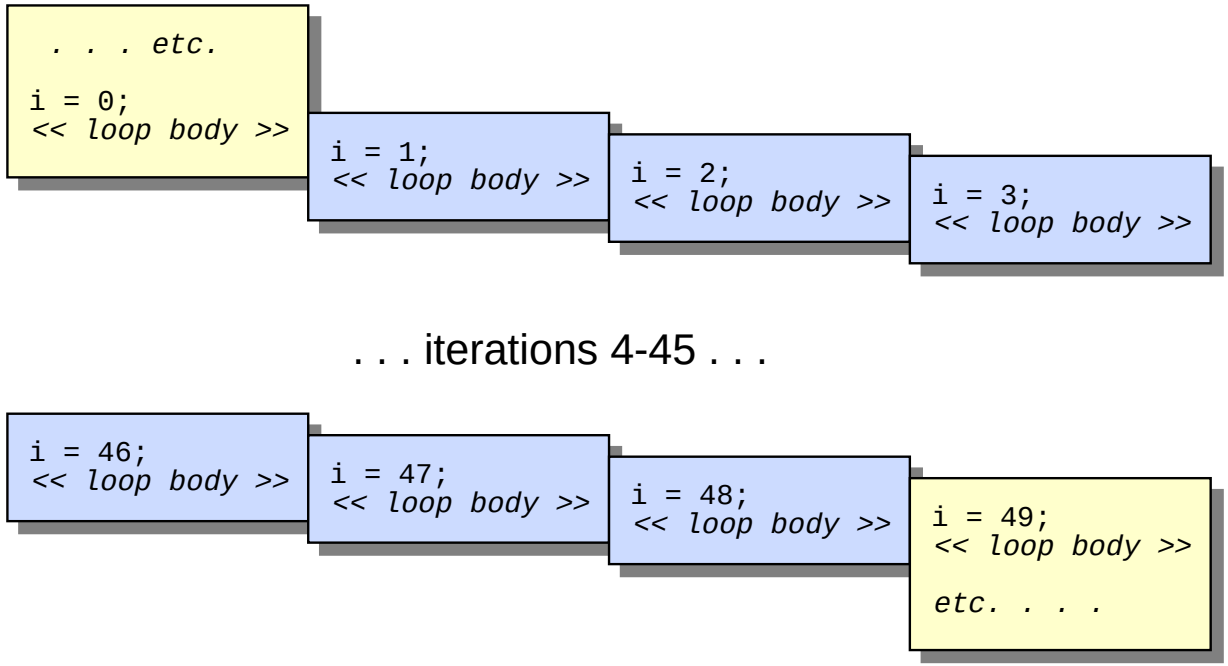
Original Loop

```
 . . . etc.

for (i=0; i < 50; i++)
{
     << loop body >>
}

etc. . . .
```

Speculative Threads



— Loop iterations are dynamically distributed among available CPUs when a specially-marked loop is executed

— Requires loop bodies with limited and/or easily controlled loop-carried dependencies

— Speculation hardware enforces loop-carried dependencies

Software Overheads

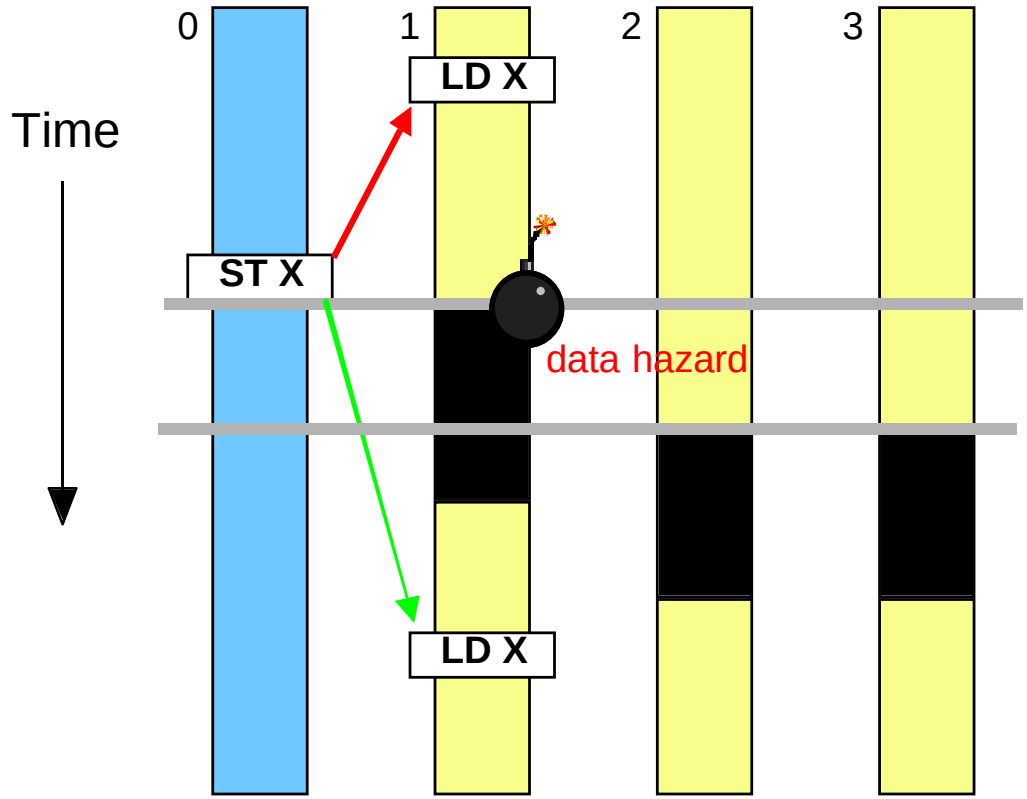
- Subroutine thread control overhead is significant
  - — Callee-saved registers must be passed through memory
  - — Return value predictions must be made and checked
  - — Complex OS-like data structures require locks and management
- Loops *with* subroutines incur subroutine compatability overhead
  - — "Slow" loops can execute subroutines and nested loops *within* loops
  - — "Quick" loops disable subroutines and nests within loops for speed
  - — Additional overhead comes from fact that loop bodies are all subroutines
- "Improved" loop management uses a simplified system
  - — *All* subroutine speculation is disabled, eliminating *all* overhead from it

Routine	Subroutine Overhead	" Slow" Overhead	" Quick" Overhead	" Improved" Overhead
Start Subroutine / Loop	~70	~75	~70	~30
End of each loop iteration	—	~80	16	12
Finish Subroutine / Loop	~110	~80	~80	~22
Violation: Local	~30	~30	20	7
Violation: Receive from another CPU	~80	~80	11	7

Units: instructions per handler invocation

An Example: Violation

- All parts of the system work together to run speculatively
- An example: The handling of a violation event . . . .



1 reads address X

0 writes address X

1 notices the data hazard, starts the VIOLATION exception handler

The handler sends out a KILL message to more speculative processors

They respond by executing the KILL handler

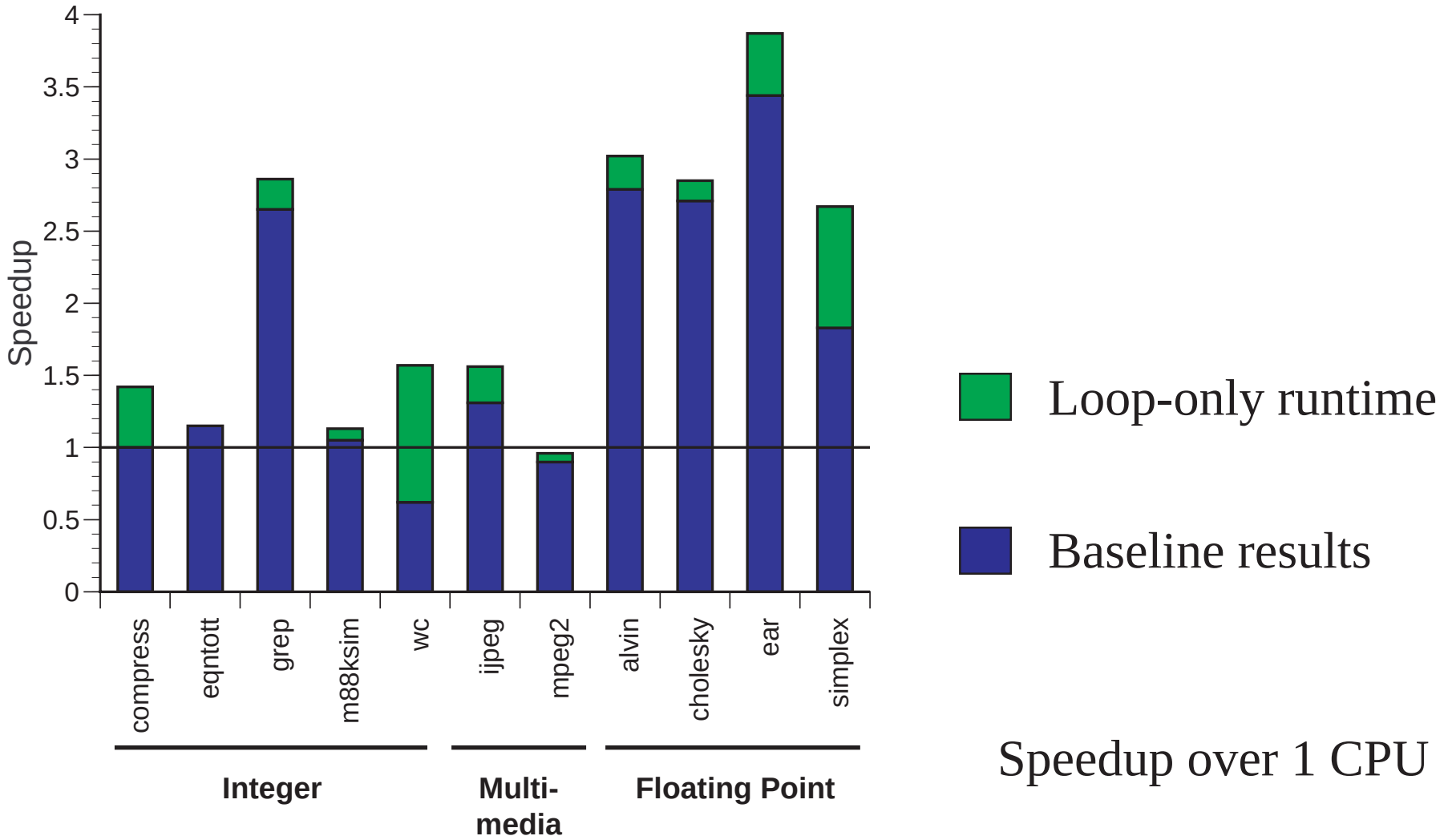
All 3 CPUs restart

The correct value is now read

Lines indicate a write broadcast to all processors

Speculation Performance

— Results representative of entire uniprocessor applications
— Simulated with accurate modeling of Hydra's memory

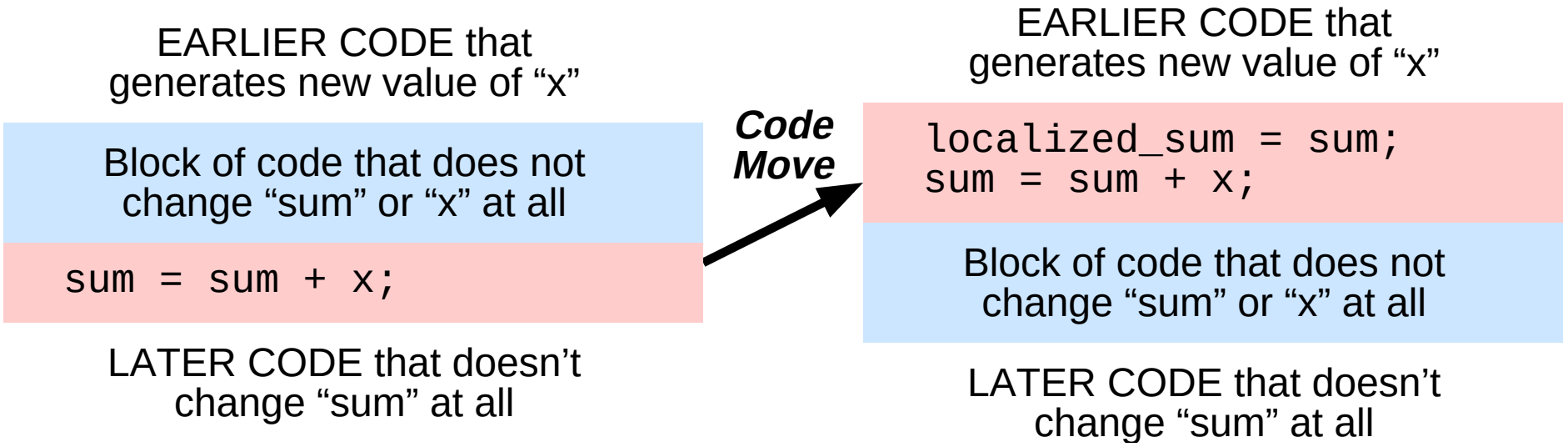


Feedback Optimization

- Speculation always allows automatic parallelization
  - — Sequential program semantics are always maintained
- *However* code without parallelism may not speed up
  - — Algorithm may be inherently very serial
  - — Code may have been written so that parallelism is hidden
- *Hence* performance tuning may be necessary
  - — But is easier because speculation support can provide feedback statistics during early, untuned runs of speculative applications
    - Unlike hand parallelization, where you can't run it until you're done!
  - — Programmer can then make adjustments *just* in problem areas
    - A few small, quick improvements can help a lot!
    - Meanwhile, let speculation handle the less important parts of code for you
    - Allows *most* parallelism in code to be found in just hours
  - — Future work: Automating the feedback process

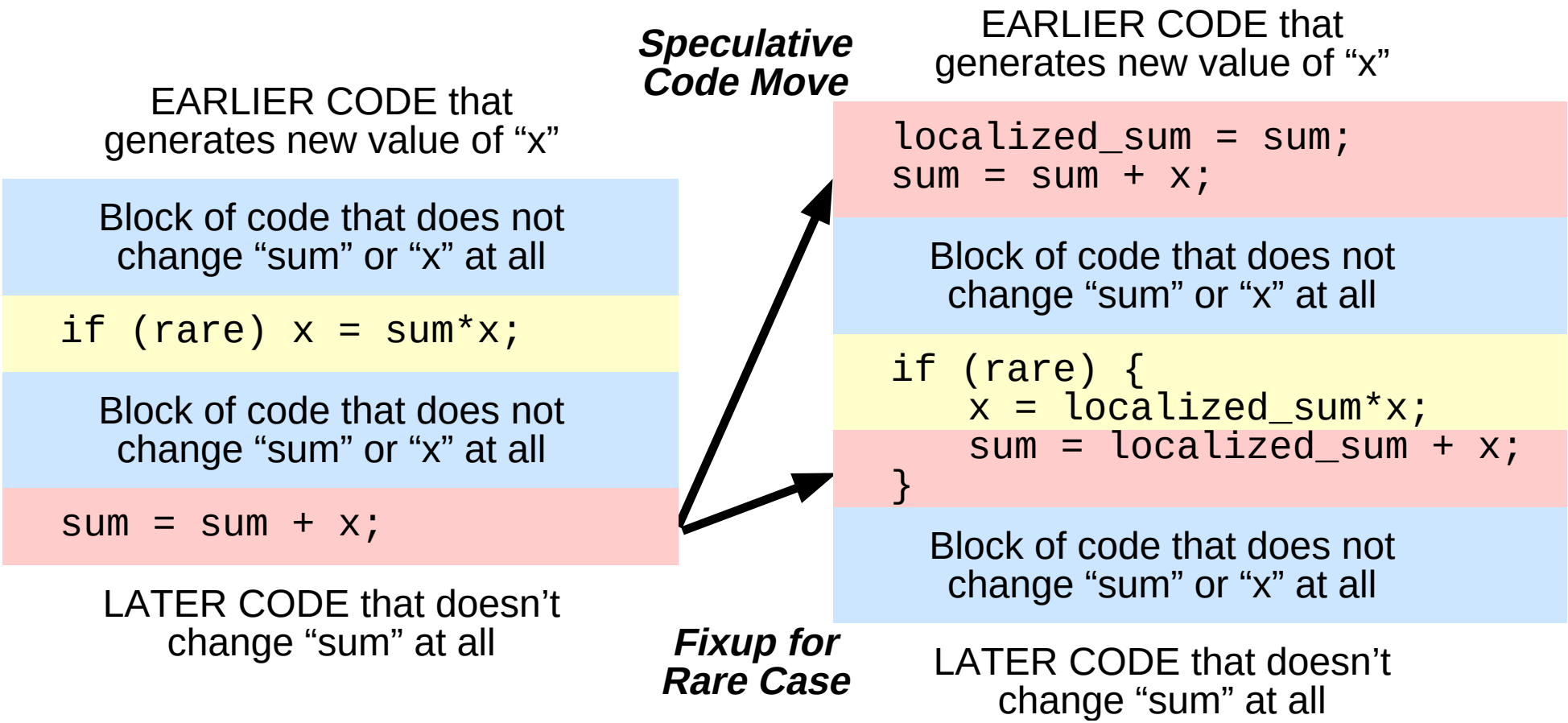
Optimization: Code Motion

- Simply moving a write to a loop-carried variable closer to the top of each iteration can increase parallelism
  - Local copy of value should be made first so code continues to run unmodified



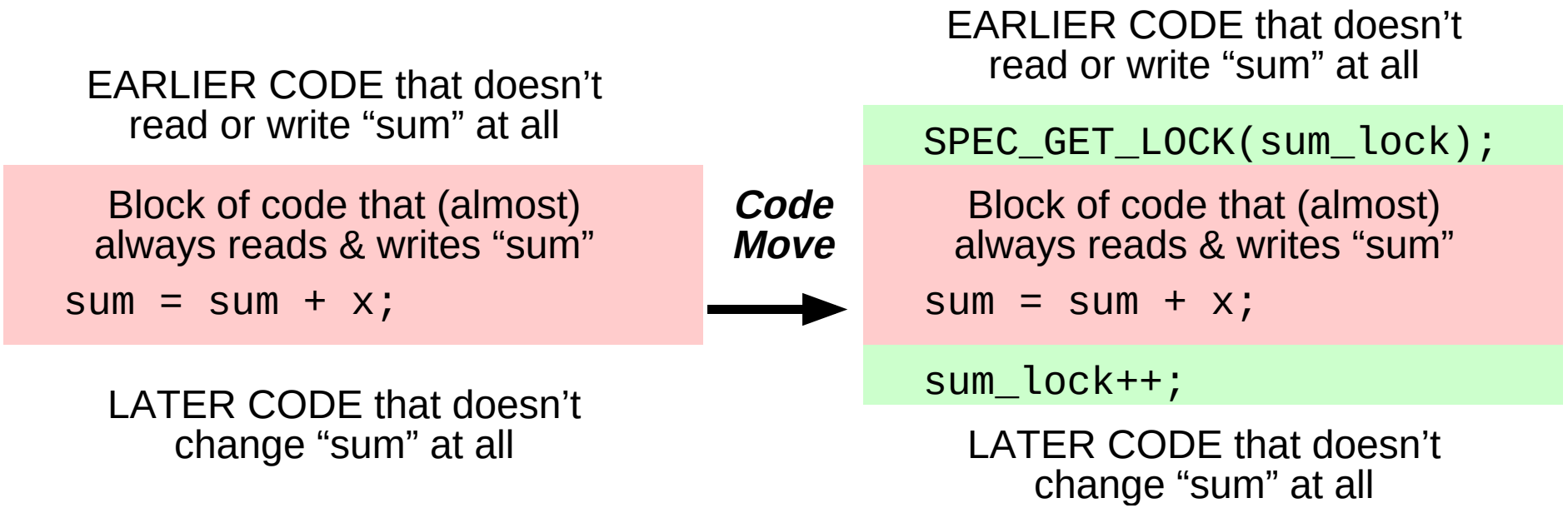
Optimization: Value Prediction

- With speculation, code movement can even be performed when there is a chance for failure
- Prediction tries to make *common case* parallel



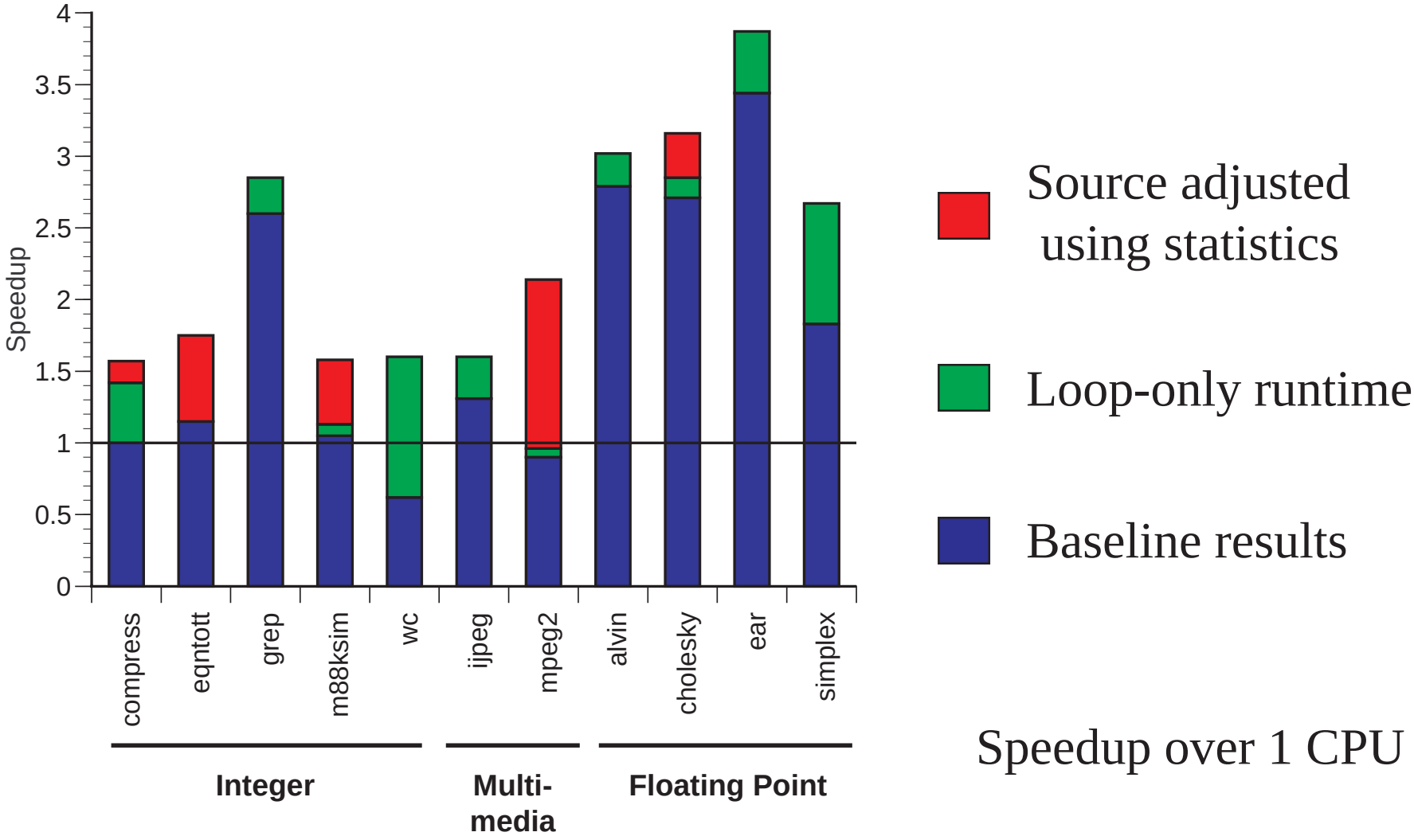
Optimization: Synchronization

- Explicit synchronization can be used to protect frequent and unpredictable dependencies
  - Similar to synchronization in conventional parallel code
- Only the *most critical* dependencies should be synced
  - Speculation can continue to protect more minor dependencies



Improved Performance

— Half of previous sample was improved by optimizations
— Improvements took hours or days, not weeks or months



Conclusions

- Hydra offers many advantages
  - — Great performance on parallel applications
  - — Good performance on most uniprocessor applications using data speculation mechanisms
  - — Scalable, modular design
  - — Speculative hardware does not add much to cost, yet greatly increases the number of parallel applications
- Thread-level speculation is a useful tool for easing code parallelization
  - — Sometimes allows virtually automatic parallelization
  - — Otherwise, can provide feedback to help optimize parallel code
  - — Allows parallel code to be generated *quickly*

Acknowledgements

- The Committee
  - — Chair: Antony Fraser-Smith
  - — Advisor: Kunle Olukotun
  - — Readers: Mark Horowitz, Mendel Rosenblum
- The Research Group
  - — Mikey, Ben, Manohar, Rachid, Vic, Ayo, Melvyn, John
  - — Alumni: Basem, Ken W., Ken C., Jeremy
  - — Others: David, Jeff, and Shih-Wei (compilers), the Bobs (SimOS), earlier SimOS & FLASH folks
- Support Staff: Darlene, Charlie, Thoi
- My parents
- And all of you, for sitting through this . . .