Data Race-Free and Speculative Models

Zach Pomper
Maxwell Johnson

Electrical & Computer ENGINEERING
Carnegie Mellon University

DeNovo: Data Race-Free Model

- Modern consistency provides the programmer too much freedom
  - "Wild shared memory behaviors"
- Requires sophisticated, complicated, high-overhead coherence protocols
- Coherence can be simplified by moving complexity to the compiler
- Compiler can be simplified by restricting the programmer

Deterministic Software

- Deterministic Parallel Java
- Static checker guarantees code is deterministic
- `foreach`, `dobegin` ≡ fork/join, each defines a "phase"
- "DPJ guarantees that the result of a parallel execution is the same as the sequential equivalent"
- Every memory object assigned to named "region"
- Every method annotated with read/write "effects"
  - This is potentially very conservative
- Compiler enforces no interference

DeNovo Protocol

- Three states: Invalid, Valid (read access), Registered (write access)
- L2 lines hold data or, if line is Registered in some L1, that L1's ID
  - Zero directory (registry) overhead
- Compiler inserts self-invalidation instructions at the end of a phase
  - Nice HW optimization: Don't need to invalidate anything we touched in this phase; we already have the current value (by assumption).
- Should only invalidate the region accessed in phase

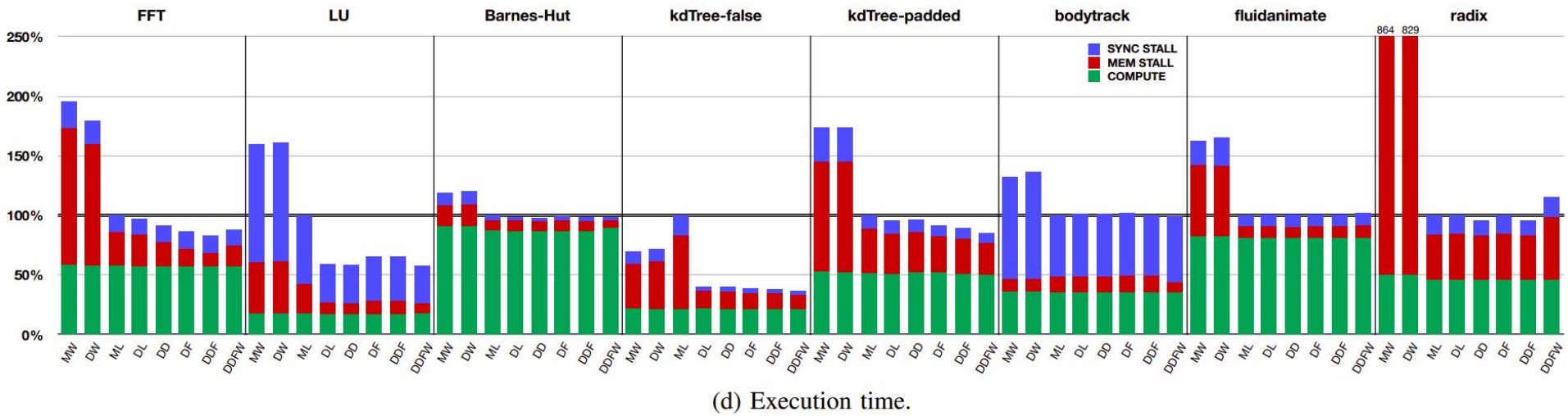
Refinements/Optimizations

- Changing the granularity
  - Can mark each word as valid/invalid, use merge operations
  - Byte-level granularity possible, but uncommon, so inefficient
- Eliminating indirection
  - Predict which L1 holds the data, request from that instead of L2
  - Mispredicts are NACK'd, which is already part of the protocol
- Flexible communication granularity
  - Communication region table can tell HW how data is structured
  - Allows prefetching w/o modifying protocol

Storage Cost

- L1: 12-25% (authors phrase as “1.5-3% of L2”)
  - Per-word: 4-8 bits
    - 2 state bits
    - 1 `touched` bit
    - 1 or 5 (or more?) region bits
- L2: 3.5%
  - 1 bit per word, 2 bits (valid & dirty) per line
- Vs. in-cache full map directory: 5 bits/line in L1, N bits/line in L2
- Vs. duplicate tag directories: Associative lookup is not scalable
- Vs. tagless directories: 3-5% L1 plus state, more invalidations

Performance



(d) Execution time.

Fig. 2: Comparison of MESI vs. DeNovo protocols. All bars are normalized to the corresponding ML protocol.

MW = MESI word-sized
DW = DeNovo word-sized
ML = MESI line-sized
DL = DeNovo line-sized
DD = DL w/ (perfect) direct cache-to-cache transfer
DF = DL w/ flexible communication granularity
DDF = DL w/ both optimizations
DDFW = DW w/ both optimizations

Verifiability

- Formal verification on a very small network in DeNovo vs. MESI
- Found bugs in both
  - DeNovo bugs were simple mistranslations
  - MESI bugs were subtle races
- Order of magnitude difference in verification time
  - DeNovo: 85k states, 9 seconds
  - MESI: 1,250k states, 173 seconds

A Transaction Memory Model (TCC)

- Sequential consistency is slow, weak consistency is difficult to program around
- Enter transactions as **the** memory operation primitive

Fundamental principle:

- All memory operations now local-only
- Operations become visible to other cores only on successful commit
- All but one commit fails on conflict, losers retry

Glaring Problems With TCC

- Who wins in a given commit conflict? It is difficult to make this decision without starving retries, especially as some commits encompass long instruction sequences
- Throughput is exchanged for generality as transactions retry, losing potentially large chunks of work
- Long sequences also increase transaction latency, negatively affecting system responsiveness
- Commit arbitration requires vast memory bus bandwidth, as conflicting transactions need to coordinate among all cores, i.e. broadcast

Subtler Problems With TCC

- Every commit failure will cause a checkpoint rollback -- while this can piggyback off of exception rollback mechanisms, they are typically not designed with performance in mind
- Transactions require cache data for each memory operation, this space is potentially unbounded in transaction length
- Unclear how to handle numa/exotic interconnects. (It may be prohibitively expensive to wait on some remote cores for commit confirmation/abort.)
- Forced to add remote coordination for data-partitioned workloads

Upsides of TCC

- Programmers don’t need to be concerned about parallelism. Not even a little bit!
  - Well okay, all of the usual parallel performance pedagogy still applies, but allowing for longer transactions does allow for the elimination of many/most synchronization primitives.
- Cache coherency becomes outmoded, as remote caches no longer need to be coherent -- saves area and implementation complexity
- Can reuse existing superscalar mechanisms like instruction windowing to speculate across transaction boundaries

Proposed TCC Implementation

- Buffer writes to flush to memory all at once on transaction complete (a commit packet)
- Similarly to coherence protocols, snoop the interconnect and check for locally speculated addresses for conflicts with commit packets
  - Rollback to known-good checkpoint on conflict
- Compiler aware of maximum transaction length, but hardware could automatically partition long instruction sequences into sub-transactions
- Particular loads/stores could be ‘promised’ to be local-only
- Add transaction buffers to do useful work while arbitration is ongoing (expensive)

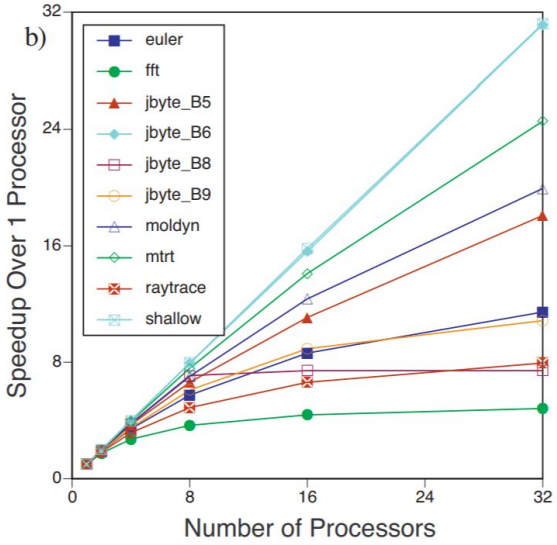
Simulation Results

- Interconnect could be saturated by commit packets at higher core counts
- Performance severely degraded (from perf. increase to loss) with increase in commit arbitration latency
- Most workloads don't overflow the maximum transaction length often
- Reasonably large transaction buffers are not prohibitively expensive
  - ~20KB of added buffers for read write histories

TCC Addendum

Broadcasts in 2020+:

This graph:



Probably looks more like this: