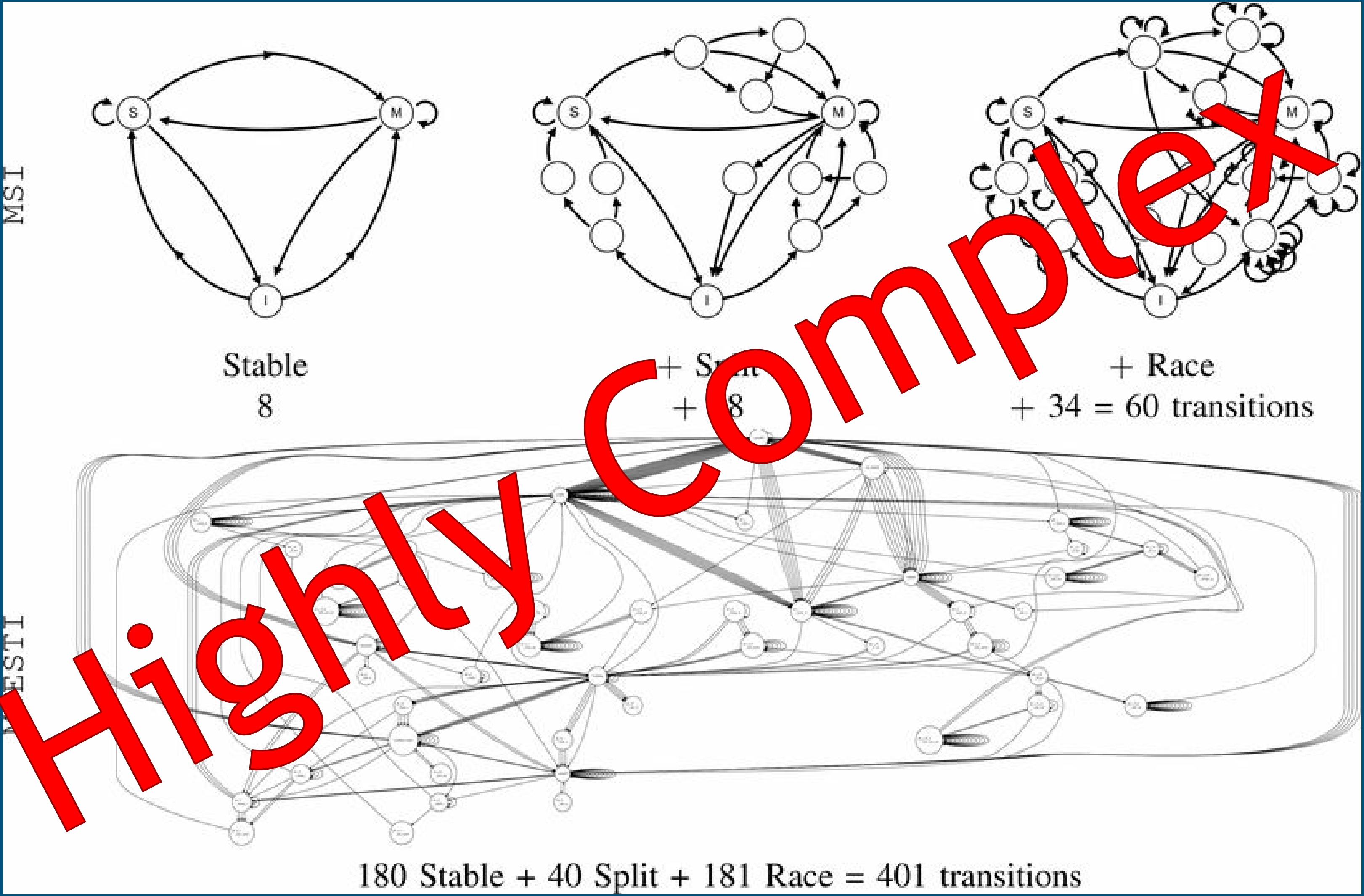
Atomic Coherence in Gem5 with a Neural Network Application

Brian Guttag, Ravi Raju, Carly Schulz, Heng Zhuo

Inspired by Dana Vantrease's "*Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols*"

Cache Coherence

- Atomicity used to be default - interconnect optimizations meant blocking buses were not favored
- Split transitions and races now occur
    - Low performance impact
    - High contribution to design impact
- Race Events are unexpected interruptions
    - Hard to find them all, hard to simulate - heavy traffic workloads might not even find them
    - Therefore, verification is difficult



Highly Complex

http://pharm.ece.wisc.edu/papers/hpca2011-vantrease.pdf

Solution

- Simplify objects to simplify verification
  - Back to atomic protocols
  - Atomic interconnects are unlikely
- Remove races by implementing mutexes
- D. Vantrease claims only 2% performance overhead over state of the art MOEFSI protocol

Our Project

- Atomic Cache Coherence MESI protocol in gem5
- PushS in gem5
- Parallel Neural Network to run on gem5

Atomic Coherence Implementation

- Definition: if, for any block, every transition from one stable coherence state to another stable coherence state occurs atomically with respect to all other stable transitions for that block.

- Atomic substrate: enforcing that only one stable-to-stable transaction to a memory block may be in progress at a time, dedicated to resolving races.
- Coherence substrate: enforcing contracts for block permission and sometimes data.

Atomic Coherence with Mutexes

- In L1 Cache
  - Lock mutex when transitioning from stable states
  - Unlock mutex when transitioning to stable states

- Atomic CResp:
  - Locked until all control responses received and processed.
- Atomic DResp:
  - Locked until all control AND data responses received and processed.

Mutex in Gem5

- Array of mutexes
  - Implemented in C++
  - Varying amounts of mutexes
  - Connected as a SimObject to L1 Cache
- Pros
  - Reduces state transitions
  - Race free
- Cons
  - Latency of mutex lock and unlock
  - Serialization of some requests

SLICC Implementation

- SLICC is the language used by gem5 to implement a cache coherence state machine

Layout of a SLICC transition:

```
          Current State,              Event,       Next State
                ↓                       ↓              ↓
transition(S,                        Store,          M){
           Grab_mutex;
           …
}
```

Actions to be taken

PushS

Basic Principles of PushS:

  - While controlling a lock you can perform any coherence updates you want
  - The list of sharers on an address are a good indication of the list of sharers on the address after it has been modified

How it Works

- When a cache block in Shared(S) transitions to Modified(M) keep the old sharers list
- When that same block is read, M->S, use the old sharers list to push the data into the caches of all the old sharers and update their coherence states

PushS

<1,1,1,0,1> (C0-C2 are sharing)

<1,0,0,0,0> (C0 is modified) $\longrightarrow$ keep the old sharers list <C0,C1,C2>

(C2 tries to read the block and acquires the lock)

1. Use the old sharers list to push the data into <C1,C2>
2. After the last cache has completed the mutex is unlocked

<1,1,1,0,1>

PushS

Pro:

- If there is a lot of sharing between caches PushS can quickly restore data to all caches after one cache has modified the data

Con:

- This is still a prediction and therefore could evict useful data in the cache with useless data

Neural Network Design

1. Interested in training rather than Inference
2. Tiling the neural network processing
   a. Maximize cache residency
   b. For feed-forward and backpropagation
3. Will have the most performance impact on convolution layer due to sharing of parameters

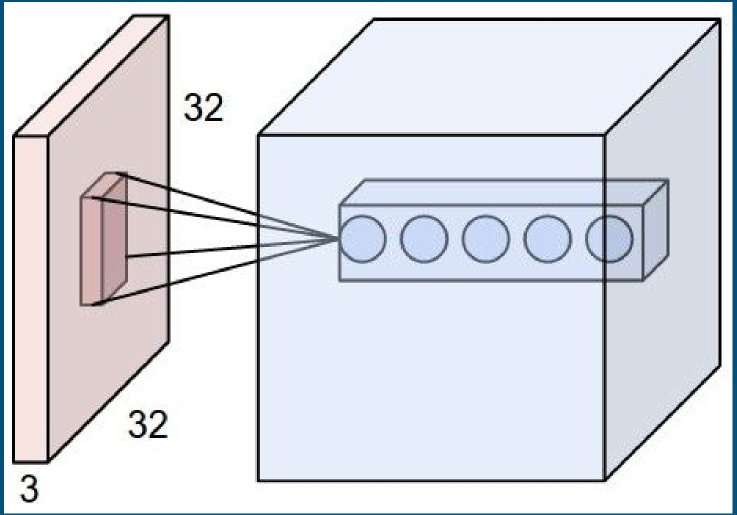


Image from: http://cs231n.github.io/convolutional-networks/

Limitation

- Full blown networks like AlexNet require environments like TensorFlow or Caffe
  - High time cost
- Implemented a 3 layer MLP neural network instead and manipulated the loop structure
  - Only need a small amount to capture the memory traffic of the system
  - Implemented in C with OpenMP

Further Work

1. MOESI Implementation
2. Add additional optimizations - ShiftF
3. Export TensorFlow python to a C++ file
    a. Hard to parallelize
    b. Less freedom to tile
4. Implement more neural network types like LSTMs/RNNs and compare the performance

Summary

- Removal of race and split transitions by use of mutexes
- Use of PushS to offset mutex overhead
  - Neural Networks can gain large benefits from this
- All this implemented in the gem5 environment with C++ SimObjects and SLICC

Questions?

References

http://cs231n.github.io/convolutional-networks/

http://www.cs.bham.ac.uk/~jxb/INC/nn.html

http://pharm.ece.wisc.edu/papers/hpca2011-vantrease.pdf