Productive Design of Extensible Cache Coherence Protocols

Henry Cook, Jonathan Bachrach, Krste Asanovic
Par Lab Summer Retreat, Santa Cruz
June 1, 2011
Background

- Cache coherence is important
  - Not just functionality, but performance and energy
  - Major implications for programming models

- Cache coherence is difficult
  - To implement and verify a single protocol
  - To explore design space of multiple protocols

- Hardware design in general is not productive
  - Often lacking modularity, extensibility, composability
Motivating Hypotheses

- Hypothesis: We can write protocols using succinct, declarative descriptions, and generate effective hardware implementations
  - Produce verified implementations from verified specifications
  - Experiment with more designs

- Hypothesis: Customization of protocol behavior is important for energy efficiency
  - On a per-motif or per-specializer basis
  - Heterogeneity in memory hierarchy
Chisel

- **Constructing Hardware In a Scala Embedded Language**

- Embed a hardware-description language in Scala, using Scala’s extension facilities
  - A hardware module is just a data structure in Scala
  - Different backends can generate different types of output (C, Verilog) from same Chisel representation

- Full power of Scala for writing hardware generators
  - Object-Oriented: Factory objects, traits, overloading etc
  - Functional: Higher-order funcs, anonymous funcs, currying
  - Compiles to JVM: Good performance, Java interoperability
Chisel Vision

- Apply the best of SW design practices to HW design
  - Write reusable modules
  - Capture design patterns as generators
  - Declarative design and search

- Write it the way you do on the whiteboard
Coherence on the Whiteboard

- Three views
- First view: coherence protocol as abstract state machine
  - Node types
  - States
  - Invariants
Coherence on the Whiteboard

- Second view: coherence protocol as set of sequences of request/reply messages
  - Set of all sequences
  - Order of messages in each sequence
  - Messages, payloads
Third view: coherence protocol as rule tables

- Given state and input, emit messages and update state
- Set of rules for each node type
Complexity Concerns

- We can verify this protocol’s rules, but are there additional sources of complexity?
  - Turning message sequences into transactions
  - Making multi-step, intra-node behavior atomic
Are we still on the whiteboard?

- Complexity at the inter-node message sequence level
  - Interleave messages re: particular block
  - Add transient/busy states to protocol
  - Handle races
  - Provide write serializability and atomicity
  - Avoid deadlock, livelock, starvation
  - Address externalities: type of network used, amount of message buffering available
Are we still on the whiteboard?

- Complexity at the intra-node atomicity level
  - Arbitrating for finite number of SRAM ports
  - Dequeuing and buffering requests
  - Enqueuing requests and responses
  - Filling and draining MSHRs
  - Multi-cycle ops with potentially conflicting updates lead to additional transient states
- Any modification to deal with the above (or area/timing constraints) could render original verification work useless
Current Focus

- Address intra-node complexity using **BlueChisel**, a declarative, embedded DSL built on top of Chisel

- Goal: Generate not only control logic for protocol-defined activity but also:
  - Arbitration logic for access to SRAM ports
  - Skid buffers and queuing logic
  - Logic implementing intermediate/transient protocol states
Inspiration: Bluespec

- High-level, functional HDL compiled to a term rewriting system and translated into HW
  - Natural way to describe many HW devices
  - Understandable, well-defined semantics
  - Conditional atomic execution of state updates, based on rules
    - Guarded atomic actions
  - Scheduler dynamically tries to fire as many as possible

- Limitations:
  - In general, guarded atomic actions are a productive abstraction in some cases, but not in others
  - Rules can only express actions that take single cycle
BlueChisel

- Core functionality: conditional evaluation of rules leads to atomic state updates

- For cache coherence, automatically generate:
  - Extra transient/implicit protocol states
  - Additional rules to govern multi-cycle operations
  - Fairness and rule priority with urgency annotations

- Extension built on top of Chisel, which we choose to apply where appropriate
Inputs: \( \text{rule ( cond ) \{ updates ... \} } \)

Ouputs:

<table>
<thead>
<tr>
<th>Old States</th>
<th>Rules</th>
<th>New States</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CAN FIRE</td>
<td>WILL FIRE</td>
</tr>
<tr>
<td></td>
<td>Muxing</td>
<td></td>
</tr>
</tbody>
</table>

**Rules**

- **Condition**
- **Action**

- **Condition**
- **Action**

- **Condition**
- **Action**
Hardware Integration

Directory controller protocol rule engine

Cache controller protocol rule engine

Diagram showing relationships between different components like Proc, rule engine, Interconnect, and others with arrows indicating flow or communication directions.
Design Flow

Abstract State Machines

Independent Message Sequences

Per-Node Rule Tables

Race-Free Coherence Transactions

Integrated Memory Hierarchy

Chisel

Per-Node Rule Engine

Chisel

BlueChisel

To formal verifier
Future Work: Extending Chisel

- Try to address inter-node, message sequence interleaving complexity
  - Generate sufficient transient/busy states
  - Compatible physical networks and buffering
- High-level composition of new transactions with existing protocols
  - “MESI” = “MI” + “E” + “S” + ?
- Patterns in message sequences
Future Work: Design Flow

Abstract State Machines

Independent Message Sequences

Per-Node Rule Tables

Race-Free Coherence Transactions

Integrated Memory Hierarchy

BlueChisel

Chisel

Per-Node Rule Engine

Old States

New States

Rules

Condition

Action

Scheduler

Muxing
Future Work: Protocols

- Protocol extensions that
  - Exploit SW knowledge via explicit SW->HW directives
  - Allow for heterogeneous memory hierarchy behavior

- Assignment of data to particular state
  - Clean this cache block

- Assignment of data to particular sub-protocol
  - Keep this block coherent using an update protocol

- Exemption of data from any protocol other than SW-directed actions
  - Only move this block in response to a DMA command
Future Work: Specialization

- Programs have been found to employ a set of common types of sharing behavior
  - Write-once, Private, Write-many, Result, Synchronization, Migratory, Producer/Consumer, Read-mostly, Streaming

- Sharing behavior is often known by expert or even application programmer
  - Utilize high-level info instead of reconstructing in HW

- SEJITS: emit optimized code from high level abstractions
  - Code that can control cache behavior according to pattern
  - Even define user-level protocols

- Bloom: Use consistency-analysis to inform decisions about which sub-protocols to employ
Conclusion

- We can write protocols using succinct, declarative descriptions, and generate effective hardware implementations
  - Declarative extension to Chisel based on ideas from Bluespec
  - Produce verified implementations from verified specifications
  - Experiment with more designs
- Explore space of protocols that use SW input to create heterogeneous behavior
Thanks
Related Work

- Murphi
  - DSL for finite-state analysis/model checking
- Teapot
  - DSL for user-space software coherence protocols
- SLICC
  - DSL for emitting SW modules for GEMS simulator
- Bluespec SystemVerilog
  - HDL based on guarded atomic actions
- Bloom
  - DSL for high-level consistency analysis of distributed parallel algorithms
Consistency

- For now, left up to enforcement at processor by compiler-issued ISA constructs (e.g. fences)

- In the future, would like to consider protocols that exploit very relaxed consistency models (e.g. location consistency)

- What patterns? What whiteboard design?
Inputs:

```java
rule ( cond ) { update ... }
```

Outputs:

- Rule engines, consisting of scheduler and muxing logic, that conditional modify state

CHISEL internals:

- During creation, log rule with component
- During elaboration, analyze domain and range of rules to create scheduler and mux code
Applying CHISEL to Intra-node Complexity

- What are the abstractions?
  - Data (states, messages, payloads)
  - Actions (send messages, update states)
  - Rules

- What are the reusable modules?
  - Collections of rules encapsulated in rule engines

- What are the design patterns?
  - Control logic for queues, arbiters, MSHRs
Flow of Information and Control

Applications

Pattern Compositions

PLL Knowledge

Specializations

PELL/Runtime Implementation

SW-Management Mechanisms

Manycore HW Features

Knowledge
- Allocation
- Placement
- Eviction
- Prefetch
- Synchronization
- Bulk Transfers

Predictability
- Isolation
- Reconfigurability

Opt-in
- Efficiency

Functional
- Correctness
Data object access patterns

- **Write-once**
  - Initialized but then only read
  - Best supported by replication (selected portions of large objects)
- **Private**
  - Need not be managed
  - On violation, demote/activate management?
- **Write-many**
  - Frequently modified by multiple threads between synch points
  - Use delayed-update protocol
- **Result**
  - Restricted subset of write-many, no reads until all writes complete
  - Lack of conflicts allows maximum utilization of delayed-update protocol
- **Synchronization**
  - Distributed locks, atomic operands
- **Migratory**
  - Read and written by single thread at a time, as object in critical sections of code
  - Associate w/ lock movement, look for signature pattern
- **Producer/Consumer**
  - Produced by one thread and consumed by fixed set of other threads
  - Eager object movement in update protocol
- **Read-mostly**
  - Replicate and update infrequently via broadcast
- **Streaming**
  - Read once (or few times), too large to keep in particular level for reuse
- **General read/write**
  - Default, rare
def train (val, data):
    def val_compare(x):
        return compare(x, val)
    z = map(val_compare, data)
Case Study 2: Stencils

- Producer/consumer
- Update protocol with proactive transmission of ghost cells to static neighbors
Case Study 3: nbody

- Migratory
- Migrate on read miss rather than replicate
Specialization: Update protocols

- Known at compile time (prod/con)
- Fixed per run (prod/con)
- Dynamic (migratory)
MI -> MESI