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

- **C**onstructing **H**ardware **I**n a **S**cala **E**mbedded **L**anguage

- 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

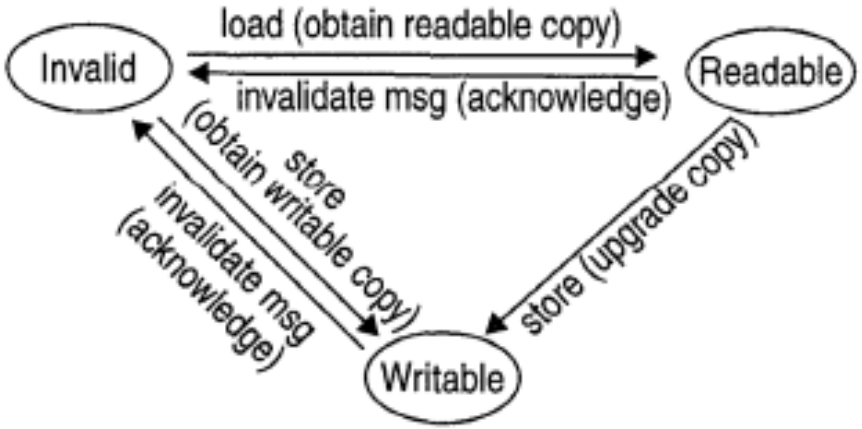
Chisel Vision

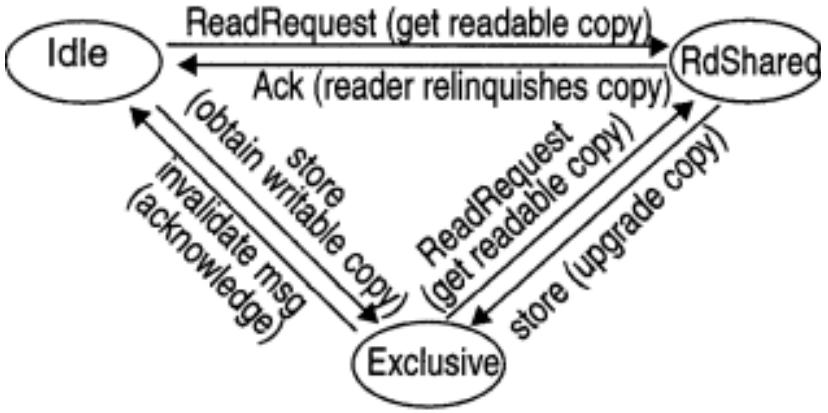
- Apply the best of SW design practices to HW design
  - Write reusable modules
  - Capture design patterns as generators
  - Declarative design and search

- Write it the way you do on the whiteboard

Coherence on the Whiteboard

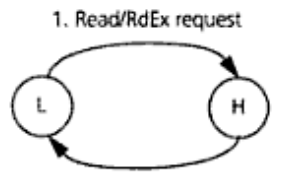
- Three views
- First view: coherence protocol as abstract state machine
  - Node types
  - States
  - Invariants

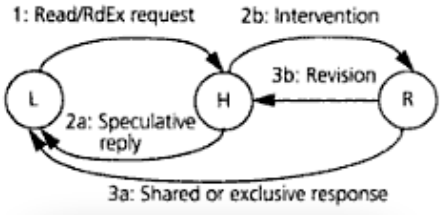


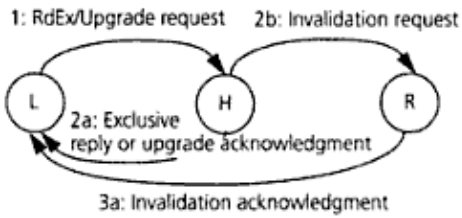


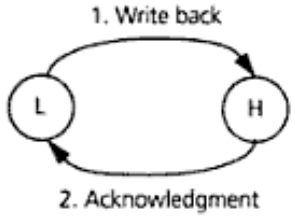
Coherence on the Whiteboard

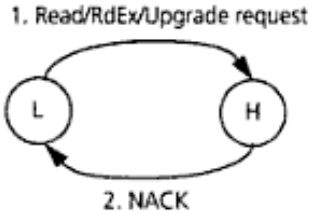
- Second view: coherence protocol as set of sequences of request/reply messages
  - Set of all sequences
  - Order of messages in each sequence
  - Messages, payloads

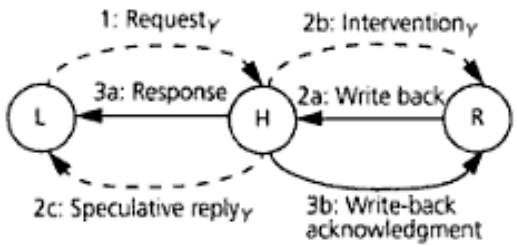












Coherence on the Whiteboard

- Third view: coherence protocol as rule tables
  - Given state and input, emit messages and update state
  - Set of rules for each node type

P2C request[4]	deferQ State	CState	Action	Next CState
Load(a)	$a \in$ deferQ	-	req → deferQ[1]	-
	$a \notin$ deferQ	Cell(a,v,Sh)	*retire*[2]	Cell(a,v,Sh)
	$a \notin$ deferQ	Cell(a,v,Ex)	*retire*[2]	Cell(a,v,Ex)
	$a \notin$ deferQ	Cell(a,-,Pen)	req → deferQ[1]	Cell(a,-,Pen)
	$a \notin$ deferQ	$a \notin$ cache	*if* cmissQ.isNotFull *then* ⟨ShReq, a, L⟩ → Mem, req → cmissQ *else* req → deferQ[1]	Cell(a,-,Pen) $a \notin$ cache
Store(a,v)	$a \in$ deferQ		req → deferQ[1]	
	$a \notin$ deferQ	Cell(a,-,Sh)	⟨Inv, a, H⟩ → Mem, Keep req	$a \notin$ cache
	$a \notin$ deferQ	Cell(a,-,Ex)	*retire*[2]	Cell(a,v,Ex)
	$a \notin$ deferQ	Cell(a,-,Pen)	req → deferQ[1]	Cell(a,-,Pen)
	$a \notin$ deferQ	$a \notin$ cache	*if* cmissQ.isNotFull *then* ⟨ExReq, a, L⟩ → Mem, req → cmissQ *else* req → deferQ[1]	Cell(a,-,Pen) $a \notin$ cache
voluntary rule	-	Cell(a,-,Sh)	⟨Inv, a, H⟩ → Mem[3]	$a \notin$ cache
	-	Cell(a,v,Ex)	⟨WBI, a, v, H⟩ → Mem[3]	$a \notin$ cache
	-	Cell(a,v,Ex)	⟨WB, a, v, H⟩ → Mem[3]	Cell(a,v,Sh)

[1] deferQ must not be full for this operation, otherwise, req will remain in the p2cQ
[2] `retire` means a response is sent to the requesting processor and the input request is deleted
[3] $c2mQ_H$ must not be full for this operation
[4] The rules for handling deferQ requests are almost identical and not shown

Figure 3: Rules for Handling P2C Requests at Cache-site

C2M Message	Priority	MState	MDIR	Action	Next MState	Next MDIR
ShReq(c,a)	Low	-	$\emptyset$[1]	⟨ShResp, a, Mem[a]⟩ → c, deq c2m Message	S	{c}
		S	$c \notin$ MDIR	⟨ShResp, a, Mem[a]⟩ → c, deq c2m Message	S	MDIR + {c}
		E	{c'}, $c' \neq c$	⟨WBReq, a⟩ → c	T	{c'}
ExReq(c,a)	Low	-	$\emptyset$[1]	⟨ExResp, a, Mem[a]⟩ → c, deq c2m Message	E	{c}
		S	$c \notin$ MDIR	$\forall c' \in$ MDIR. ⟨InvReq, a⟩ → c',	T	MDIR
		E	{c'}, $c' \neq c$	⟨WBIReq, a⟩ → c'	T	{c'}
Inv(c,a)	High	*mstate*	$c \in$ MDIR	deq c2m Message	*mstate*	MDIR - {c}
WBI(c,a,v)	High	T | E	{c}	Mem[a]:=v, deq c2m Message	S	$\emptyset$
WB(c,a,v)	High	T | E	{c}	Mem[a]:=v, deq c2m Message	S	{c}

[1] any state with MDIR = $\emptyset$ is treated as S with $\emptyset$

Figure 4: Rules for Handling Cache Messages at Memory-site

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

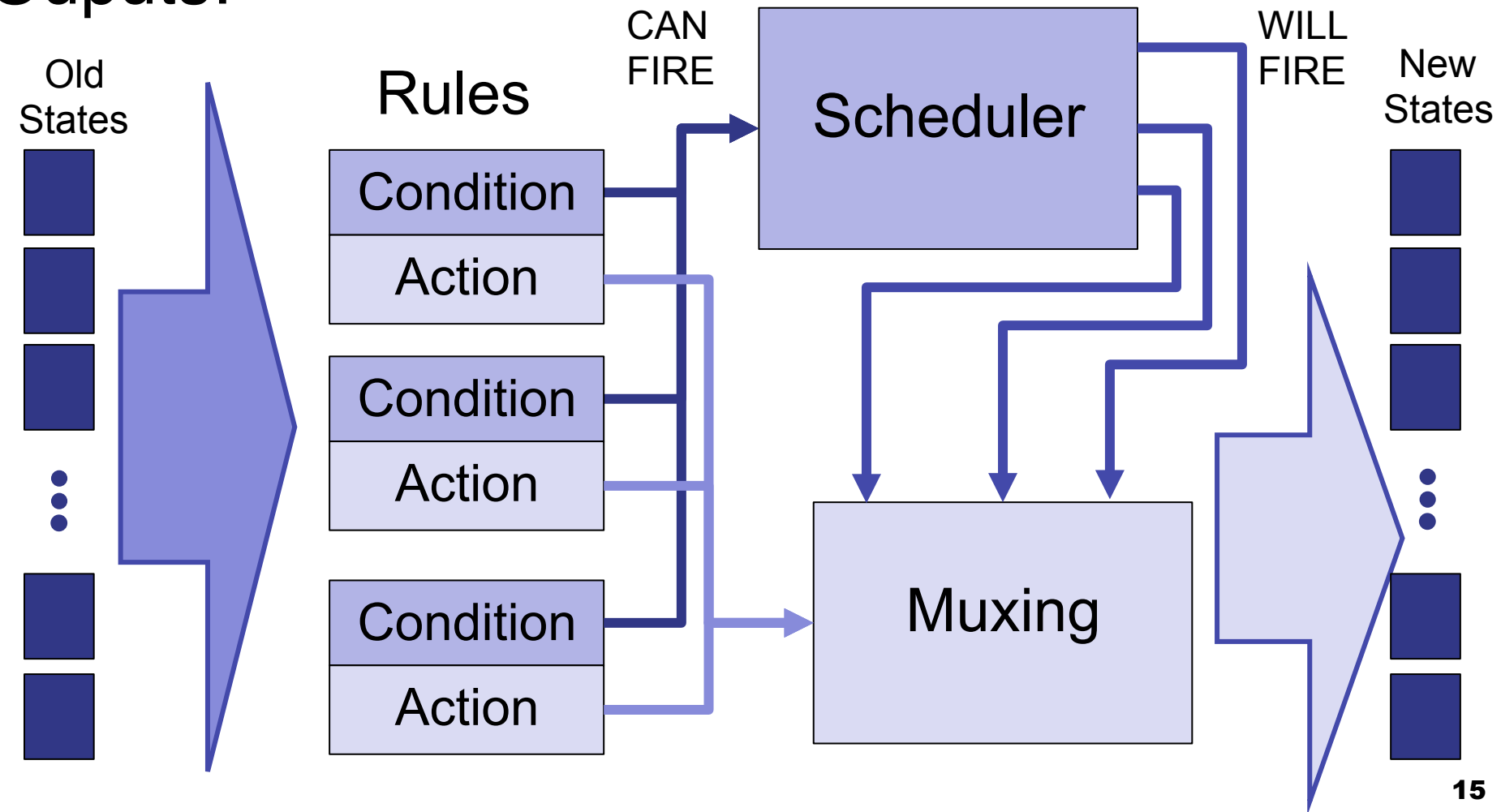
BlueChisel

- 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

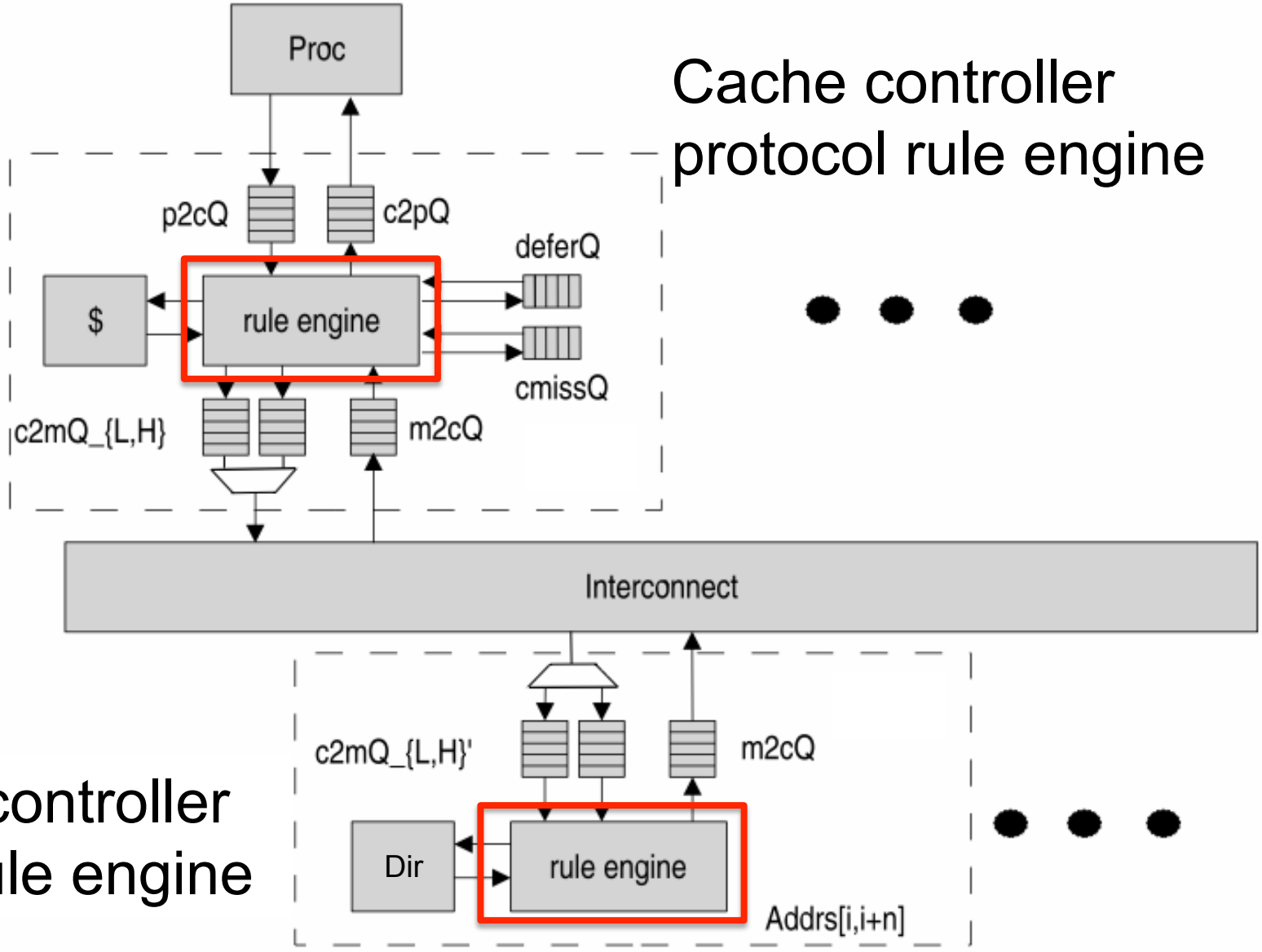
BlueCHISEL Inputs and Outputs

Inputs: `rule ( cond ) { updates … }`

Ouputs:



Hardware Integration

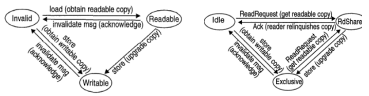


Cache controller protocol rule engine

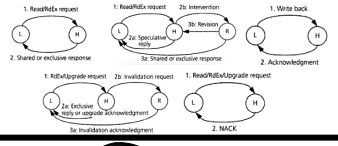
Directory controller protocol rule engine

Design Flow

Abstract State Machines



Independent Message Sequences



Per-Node Rule Tables

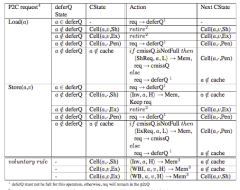


Figure 3: Rules for Handling P2C Requests at Cache-site

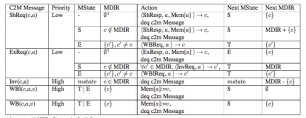
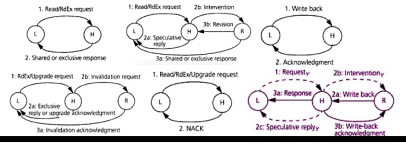


Figure 4: Rules for Handling Cache Messages at Memory-site

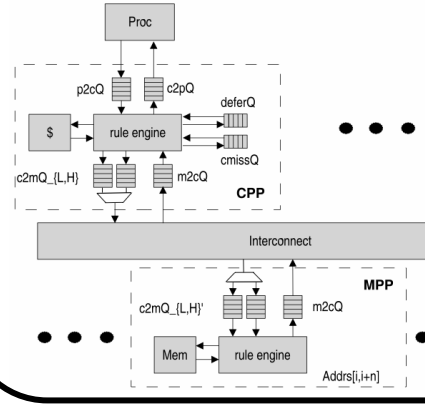
To formal verifier

Race-Free Coherence Transactions



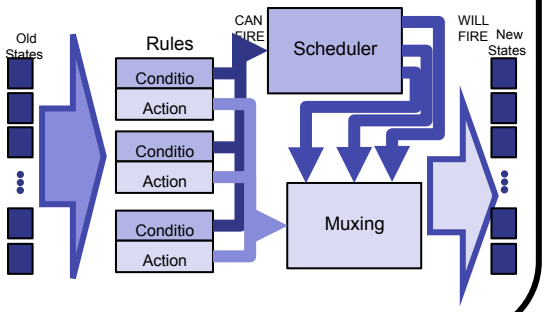
BlueChisel

Integrated Memory Hierarchy



Chisel

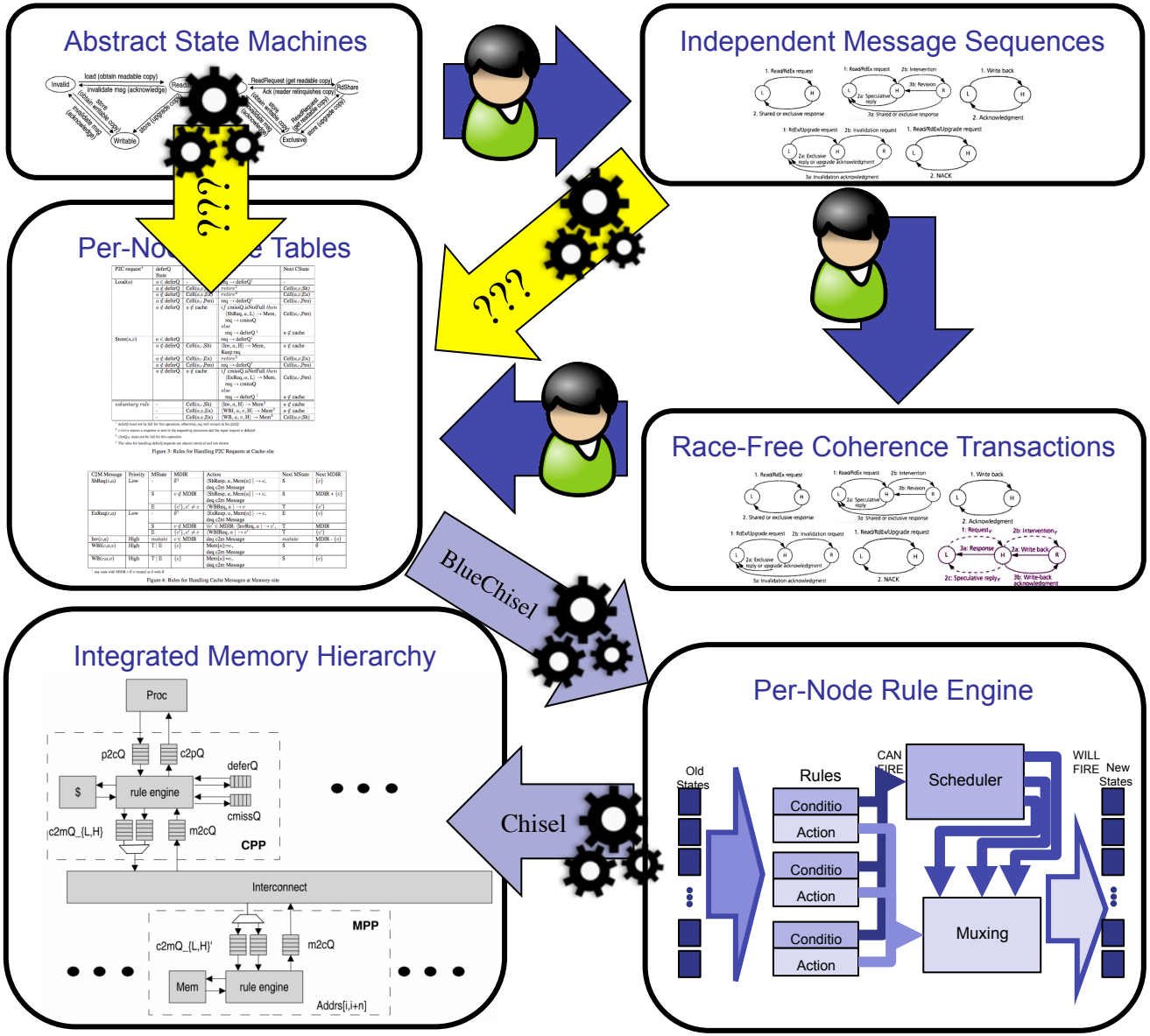
Per-Node Rule Engine



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



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

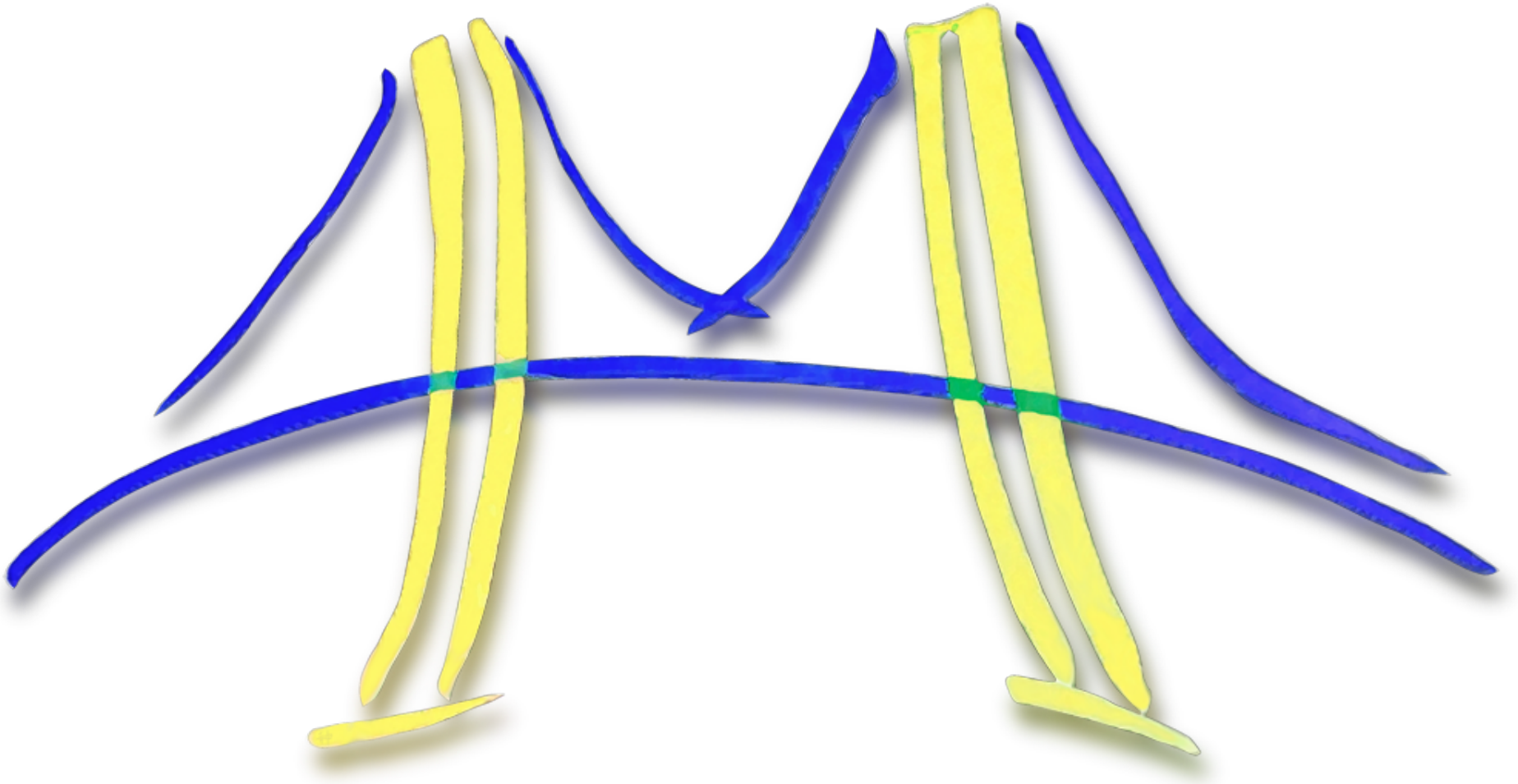
Future Work: Specialization

- 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?

BlueChisel

- Inputs:

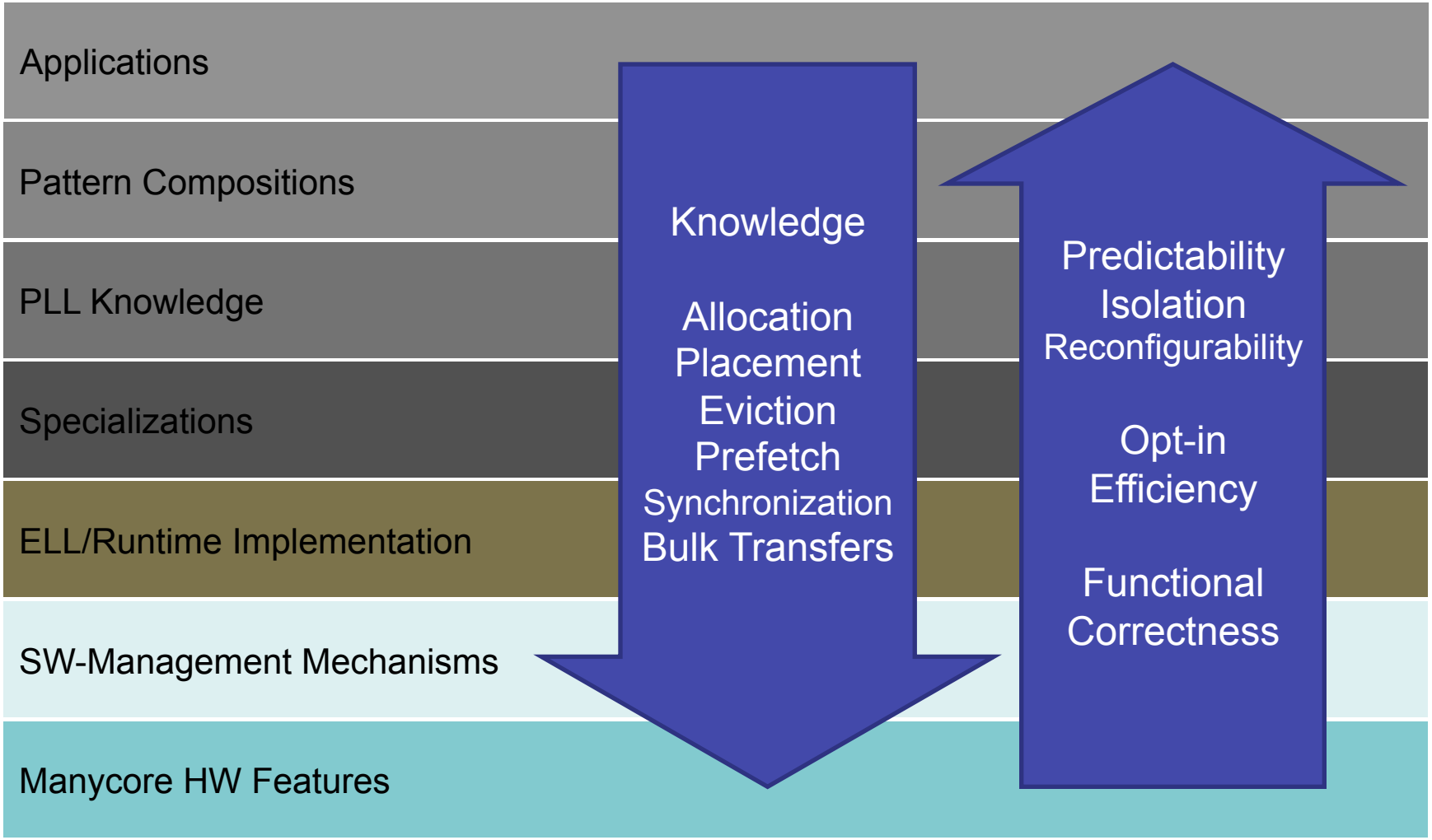
```
rule ( cond ) { update … }
```

- 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

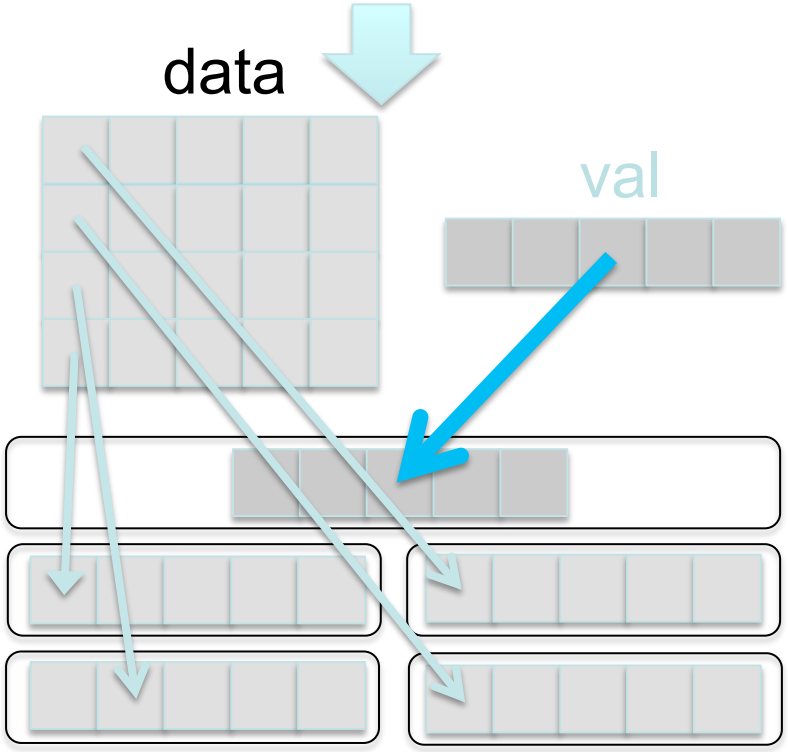


Data object access patterns

- 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

Case Study 1: SVM Training


```
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

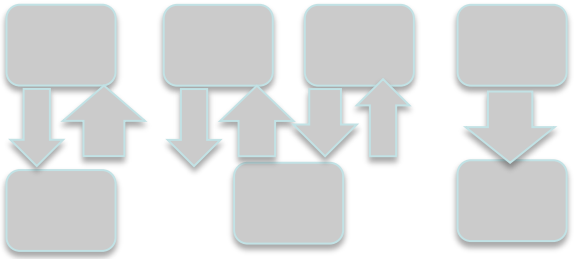
Case Study 3: nbody

- Migratory
- Migrate on read miss rather than replicate

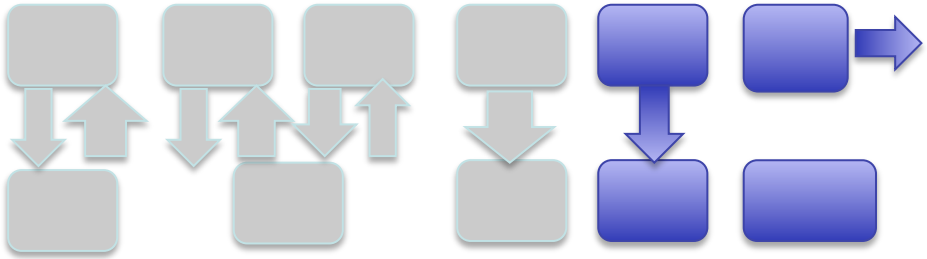
Specialization: Update protocols

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

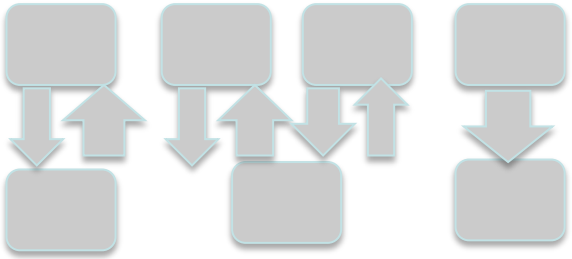
MI -> MESI



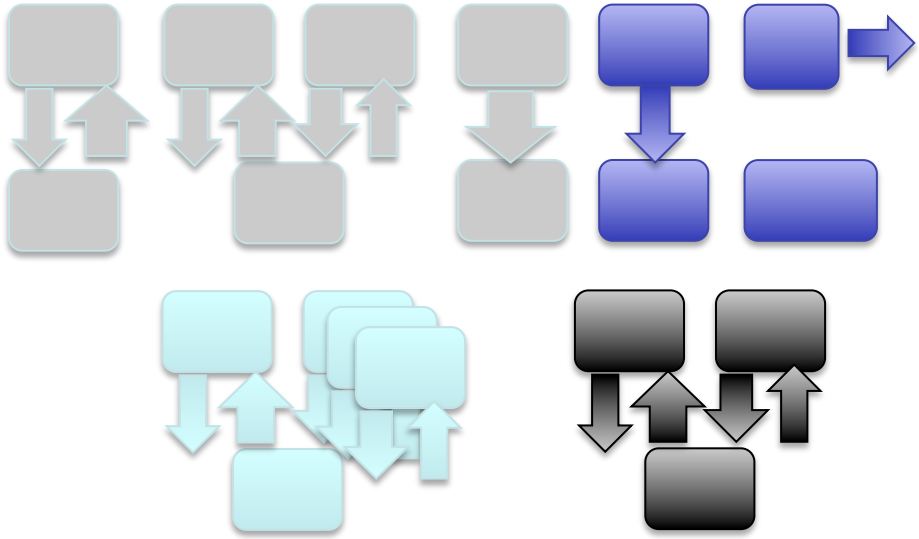
MI



MEI



MSI



MESI