The Aries Decentralized Abstract Machine Architecture and Streams

Andrew "bunnie" Huang
Ben Vandiver

Agenda

- Architectural Motivation
- Salient Architectural Features
  - Many other architectural features and issues left out for another talk
    - Capabilities
    - Migration
    - Network
    - Transactions
    - Fault tolerance
    - Introspection
    - Thread scheduling
    - ISA encoding hacks
- Streams in the context of ADAM (Ben)

Context

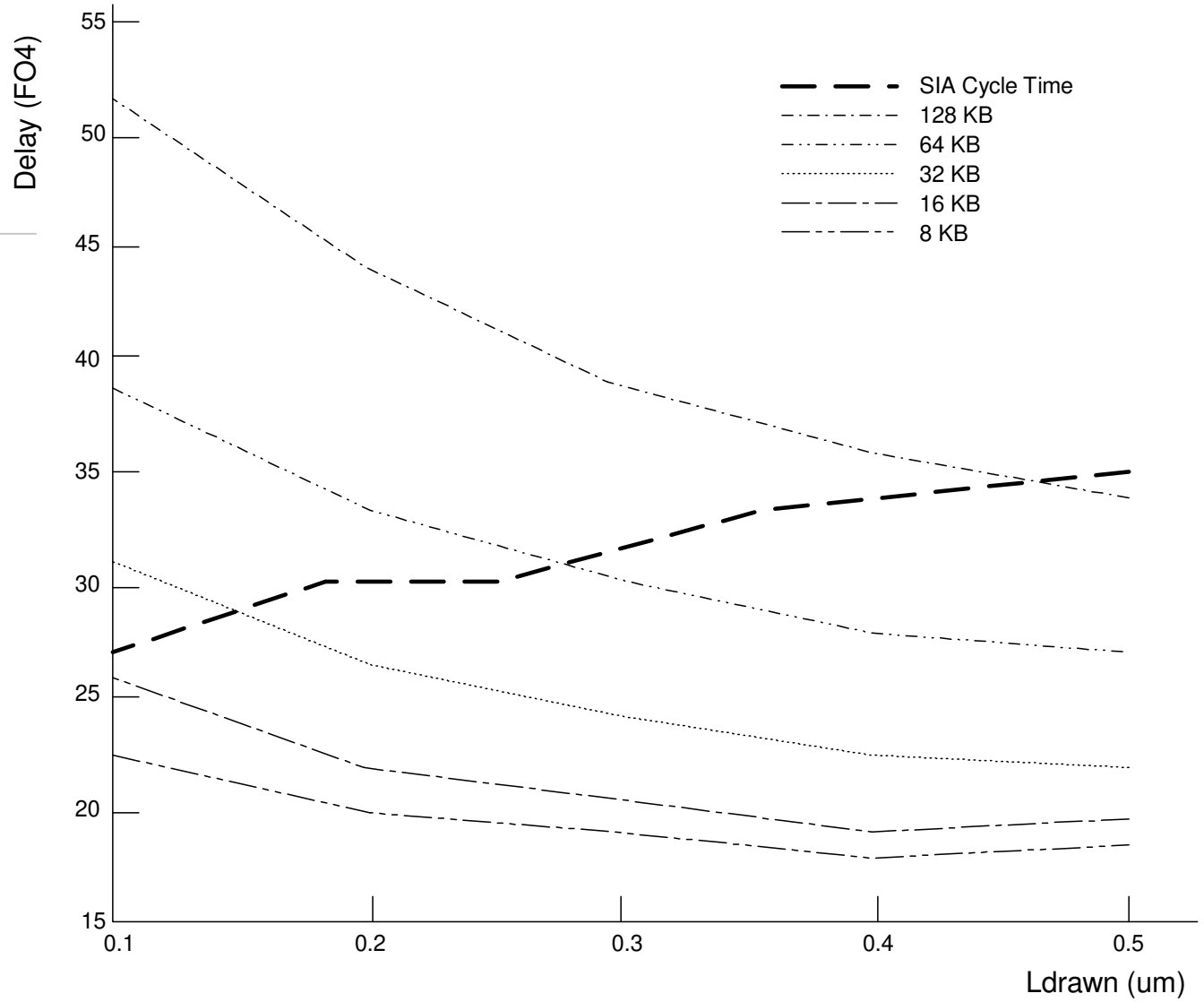
- Focus: Technology issues
  - Memory latency gap
  - Complexity
  - Programmability
- Context: Big Iron
  - Building-sized computer
  - Millions of nodes
  - 100's of Megawatts of power
  - Some techniques will apply to desktops, embedded processors thanks to device scaling

Motivation

- Issues
  - **Performance**
    - **Low Latency**
    - **High Performance Synchronization**
  - Reliability / Yield / Fault tolerance
  - Obsolescence (Cost of Ownership)
  - Complexity Management

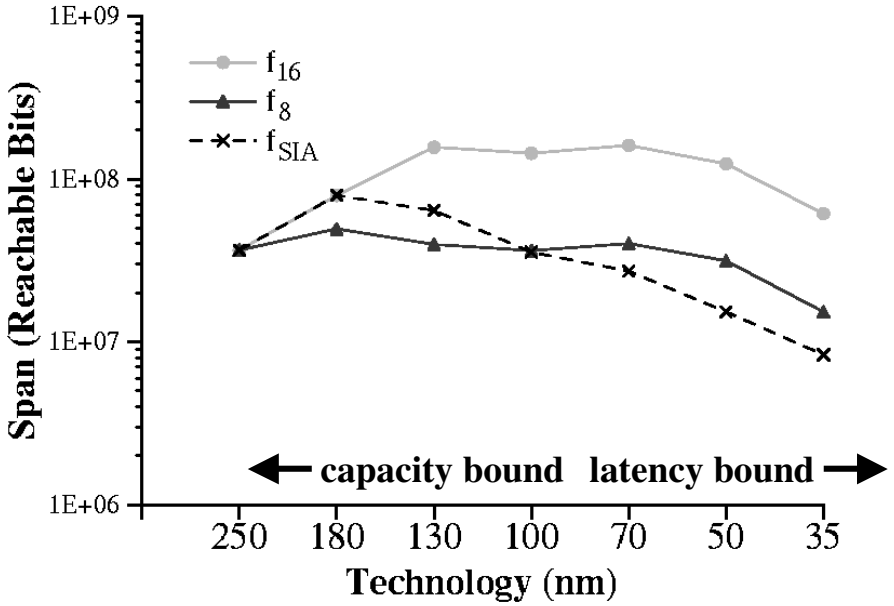
Latency

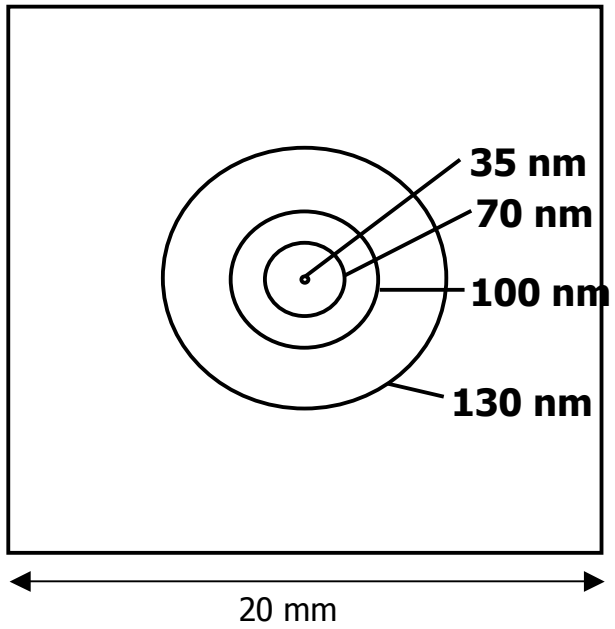
- Hard lower bound by physics
  - Good system design operates at this limit
  - Assume network components and processor components are tweaked out
- Assuming we are at the lower bound, what can we do?
  - Latency hiding through caches
    - Runs out of steam in the next decade



From McFarland, Grant: CMOS Technology Scaling and Its Impact on Cache Delay (Stanford PhD thesis, 1997)

Wire Scaling: Reduces Available Silicon Area





- Total chip area in close proximity is decreasing with technology and clock scaling
- At 35nm, less than 0.5% reachable at 8FO4 clocks

(from Last Classical Computer ISAT study, 2001)

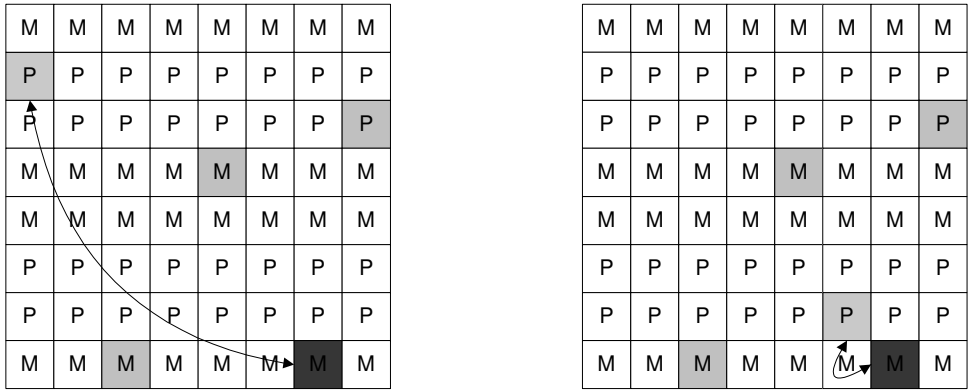
You Can't Fake What You Don't Have

- As technology scales:
  - Latency differential between cache and memory access is getting worse
  - Cache performance is *not* getting better
  - Microarchitecture is also subject to the same technology scaling issues that hurt caches
- Also:
  - Some latency cannot be hidden
    - Inter-process synchronization requires full round-trips between nodes

Synchronization on Distributed Parallel Machines

- Performance-limiting issue
  - Conventional memory locks have poor performance
    - Non-cacheable access that may require multiple round-trip accesses
    - Sharing of synchronized and unsynchronized namespaces can cause performance problems
      - Locking, aliasing
    - Lack of synchronization namespace on large machines can be limiting
    - Good programmer support of special hardware features needed as well

Latency Reduction



- Alternative solution
  - Want a spatially distributed processor that can self-organize to minimize latency
  - Efficiently migrate data and code to improve run-time performance
  - Static analysis for better initial placement helps too

Motivation

- Issues
  - Performance
    - Low Latency
    - Synchronization in a distributed machine
  - **Reliability / Yield / Fault tolerance**
  - Obsolescence (Cost of Ownership)
  - Complexity Management

Reliability, Fault Tolerance, and Manufacturing Yield

- Large systems experience high failure rates
  - Uptimes measured in hours
  - Fault prediction, detection, isolation and recovery without interruption of service is mandatory
- Manufacturing is getting difficult
  - Perfection and verification is uneconomical at 1 billion unique devices/die
  - Mask sets @ 0.1u will cost ~$1 million
  - Architect for yield and mask cost
    - Processors with regular structures
    - Partial failure on a die results in performance degradation, not the trash can

Motivation

- Issues
  - Performance
    - Low Latency
    - Synchronization in a distributed machine
  - Reliability / Yield / Fault tolerance
  - **Obsolescence (Cost of Ownership)**
  - Complexity Management

Obsolescence

- Your processor implementation is obsoleted in two years
  - Exact replacements impossible to find or manufacture in ten years
  - Want incremental, "hot" upgrades
    - Like the "Ship of Theseus"
- Want seamless scalability
  - Programs should be able to run on one or one million processors
  - Want easy portability from developer's desk to the big iron

Motivation

- Issues
  - Performance
    - Low Latency
    - Synchronization in a distributed machine
  - Reliability / Yield / Fault tolerance
  - Obsolescence (Cost of Ownership)
  - **Complexity Management**

Complexity

- 1 billion transistors on a chip is coming
  - "Flat" designs (no hierarchy) are increasingly tough to verify, place, and route
    - Design resources (time, manpower) are limited
  - RISC cores used to be 50k transistor items
    - 80 dB of microarchitecture complexity for 20 dB of performance gain!
    - 20 dB ~ whisper, 80 dB ~ vacuum cleaner
  - Want simple, repeated processor structures
    - Only way to do it at 20 million transistors/grad student

A Conundrum

- Want machine implementation independence
  - Decouple applications from the details of hardware
    - Location of data and code
    - Failed nodes
    - Newer hardware
- Want high performance
  - Implementation visibility desired
    - Take advantage of special features
    - Leverage application knowledge of data layout and communications patterns to boost performance

Virtual Machines

- Java has proven the concept
  - Reasonable performance with platform independence
  - Not performance-oriented, moderated MT
- High-performance VM targeted at parallel computing environments
  - Standard VM model is a simple RISC processor
  - Explicit architectural support for low-overhead synchronization and multithreading
  - Hygienic development environment
    - GC – compaction, migration, latency reduction
    - Type checking and support for capabilities in hardware

Solution: ADAM

- Abstract machine that provides or enables:
  - Rich synchronization namespace with low synchronization overhead
  - Decoupled mechanism for dealing with synchronization and memory access
  - Low-overhead mechanism for spatial awareness and latency reduction
  - Non-homogenous node structure
  - Object-oriented programmer-friendly development environment
- *All of the above with good single-threaded code performance and simple hardware implementation*

Queues

- Our basic synchronization, temporary storage, and communication unit
  - Block on empty
  - Block on full
  - Serialization of data in FIFO order
  - Remap queue tails for comm.
- Available as architectural primitives
  - Takes the place of registers in the RF
  - Allows compiler to deal with queues
  - Programmers only see abstracted FIFO & streaming structures

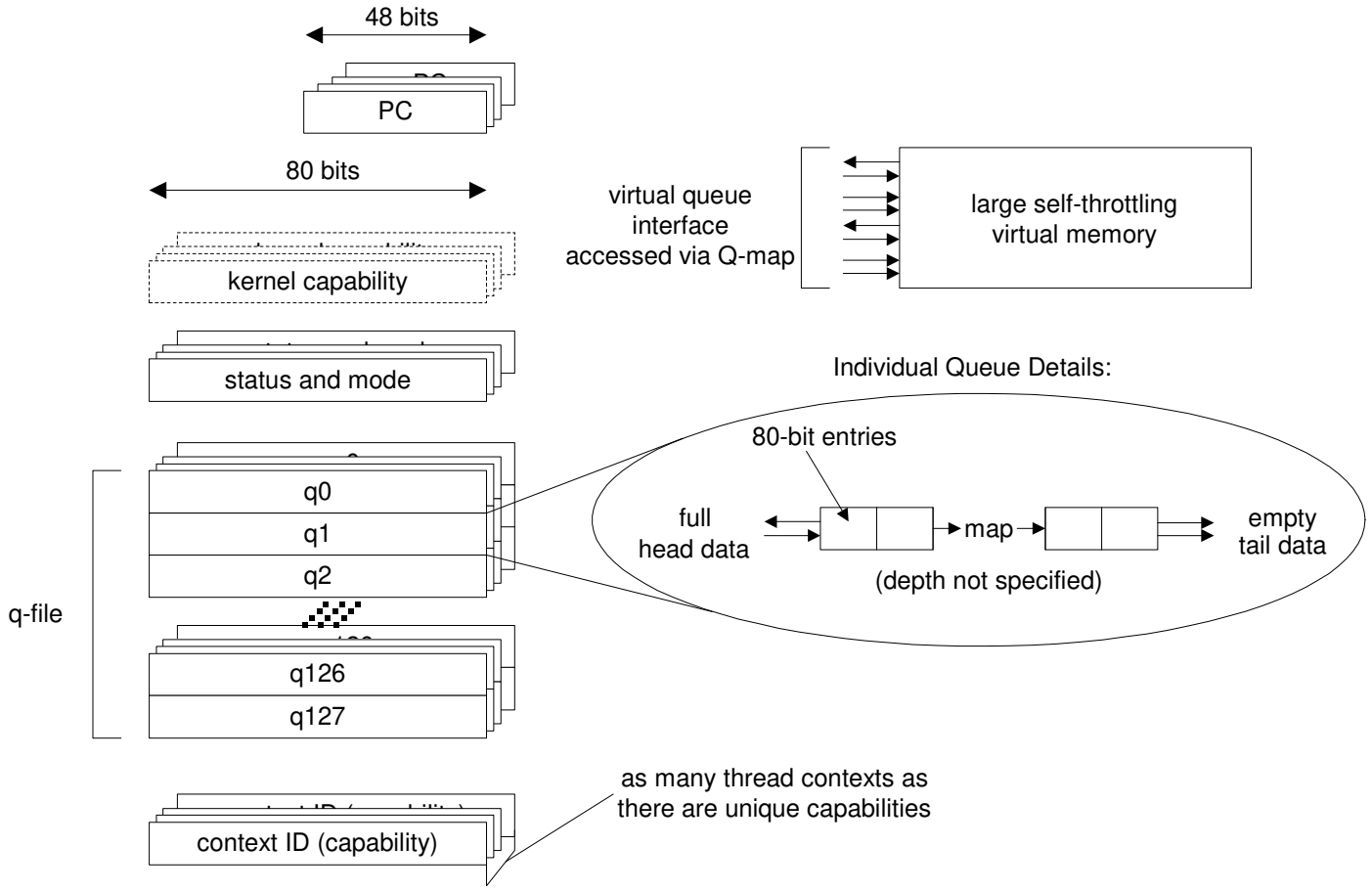
Queue-files

- Register files are mostly wire
  - Small fraction is active storage area
  - Use extra area under the wire to implement a queue
  - Wire in empty/full bits to a pipeline stall signal
    - Swap to another thread on stall

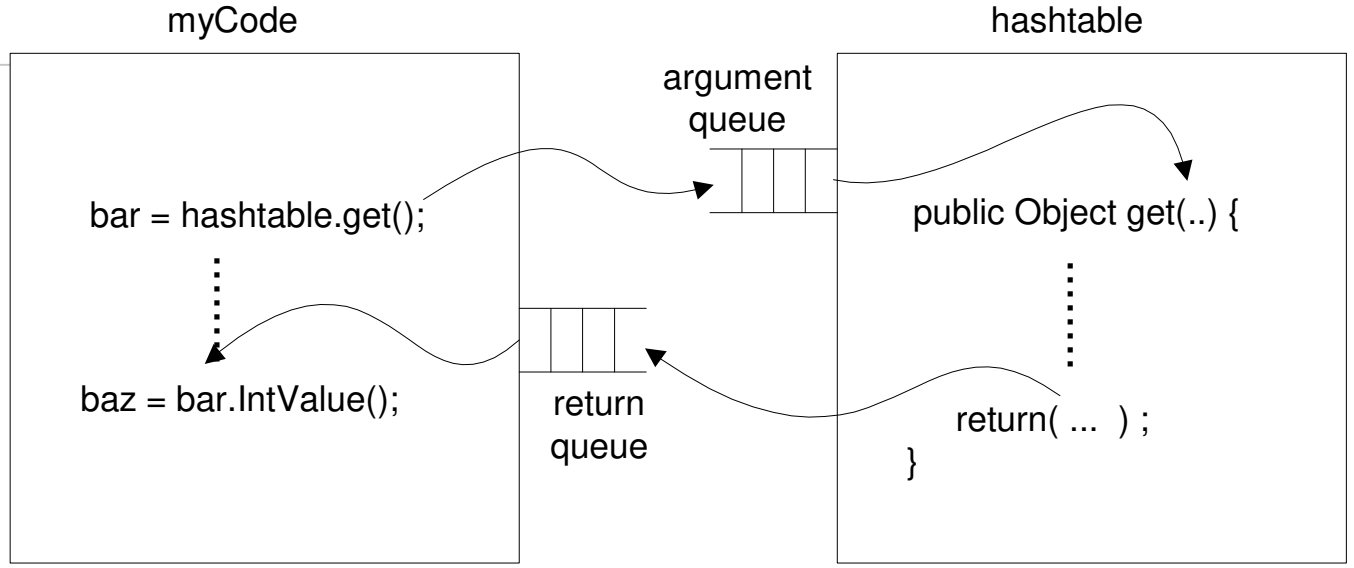
Programmer's Model

- Massively multithreaded programming model
  - Thread ID = storage pointer (capability)
- A single thread context looks similar to common RISC machine
  - 128 queues, PC, status & exception
  - ISA looks like a cross between Alpha and Java
  - Overview of ISA/datatypes/capabilities in another talk
- Inter-thread comms via queue mappings
- No loads or stores
  - Queue based comms to memory

Programmer's Model



Queue Mapping



- Map the tail of any queue to the head of a queue in another context
  - Inter-context communication through a pair of queues
  - Execution proceeds until result value is required
- Contexts do not have to be on the same node!

Queue-Based Memory Interface

- Memory accessed through queue maps
  - Multiple concurrent non-blocking loads, stores can be issued
- Enables intelligent memory
  - The other side of the queue can be anything
  - Instead of just addresses, commands can be sent to the memory subsystem
    - Pointer chasing, array indexing, hashing, etc.

Instruction Order Decoupling

- E/F bits on queues enables cheesy OOE
  - Window a set of operations and dispatch based on E/F bits
  - Semantics of execution guaranteed to be preserved despite OOE
  - Current implementation efforts do not include this feature

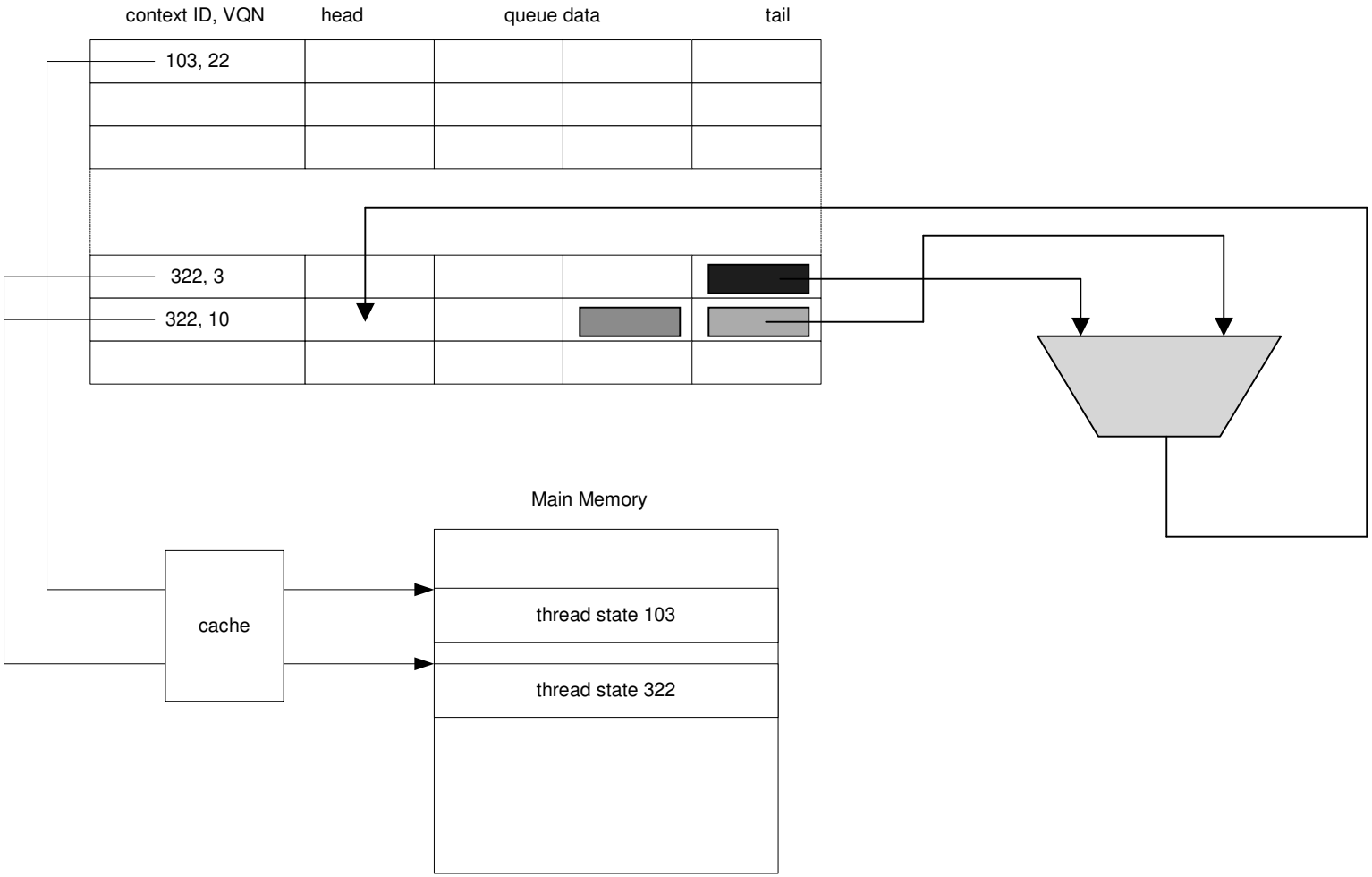
Massive Multithreading

- Pros
  - On stall in context, swap to alternate thread to keep utilization high
  - Programs written in MM style can more easily leverage massive parallelism
- Cons
  - Fast context swaps require hardware context management for up to 2^{80} contexts!
  - MM has yet to find good programmer support

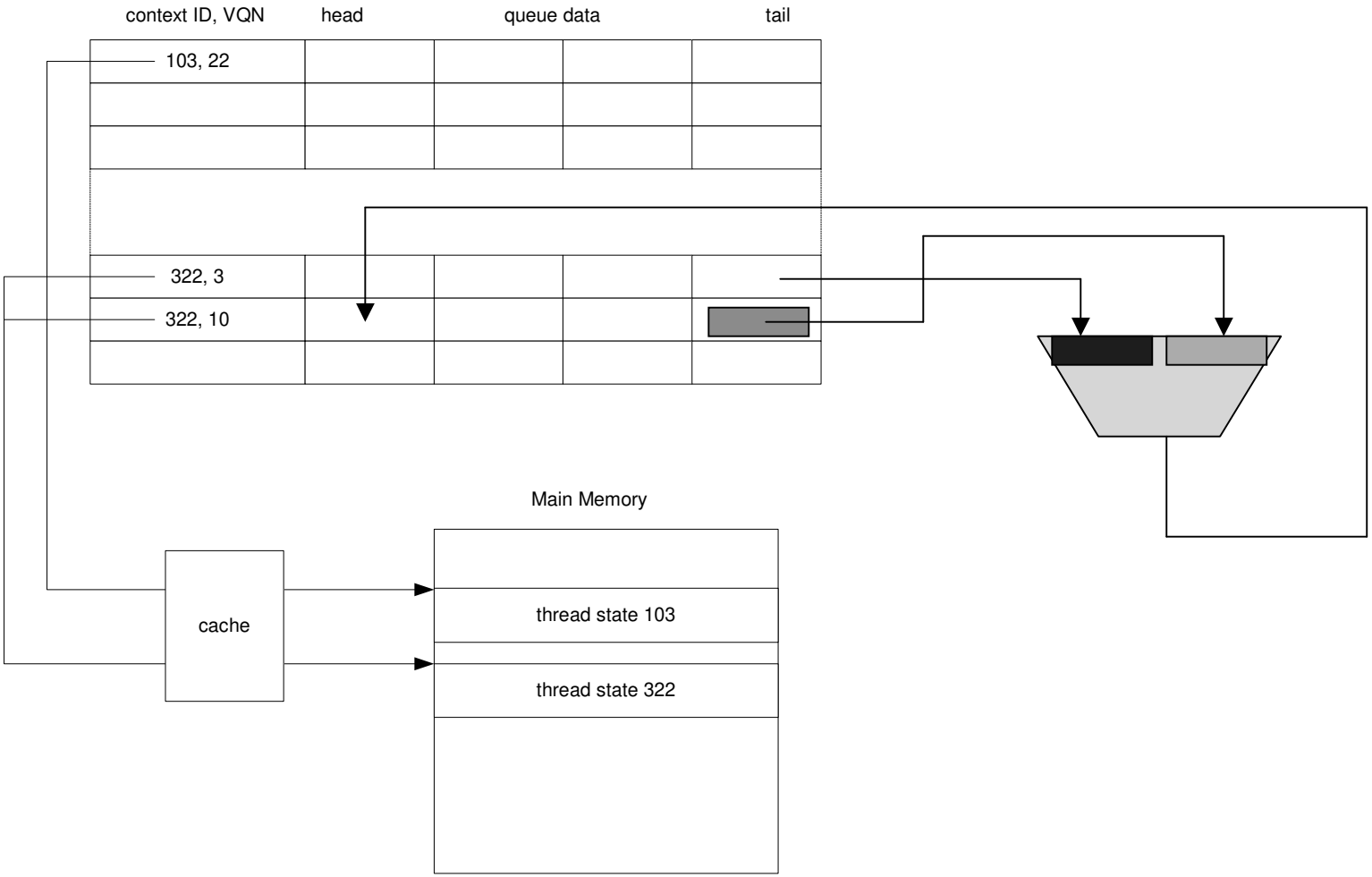
Multi-context Q-Files and Named Context State

- Infrastructure for massively MT machine
- Similar to named-state register file (Nuth & Dally, HPCA '95)
- Give each context a unique ID
- Make the queue file responsible for managing context state
  - Increases QF utilization w/out impacting code performance
  - Gives a very rich namespace of synchronization elements
  - Stripe namespace across processors for fast name resolution of inter-context comm

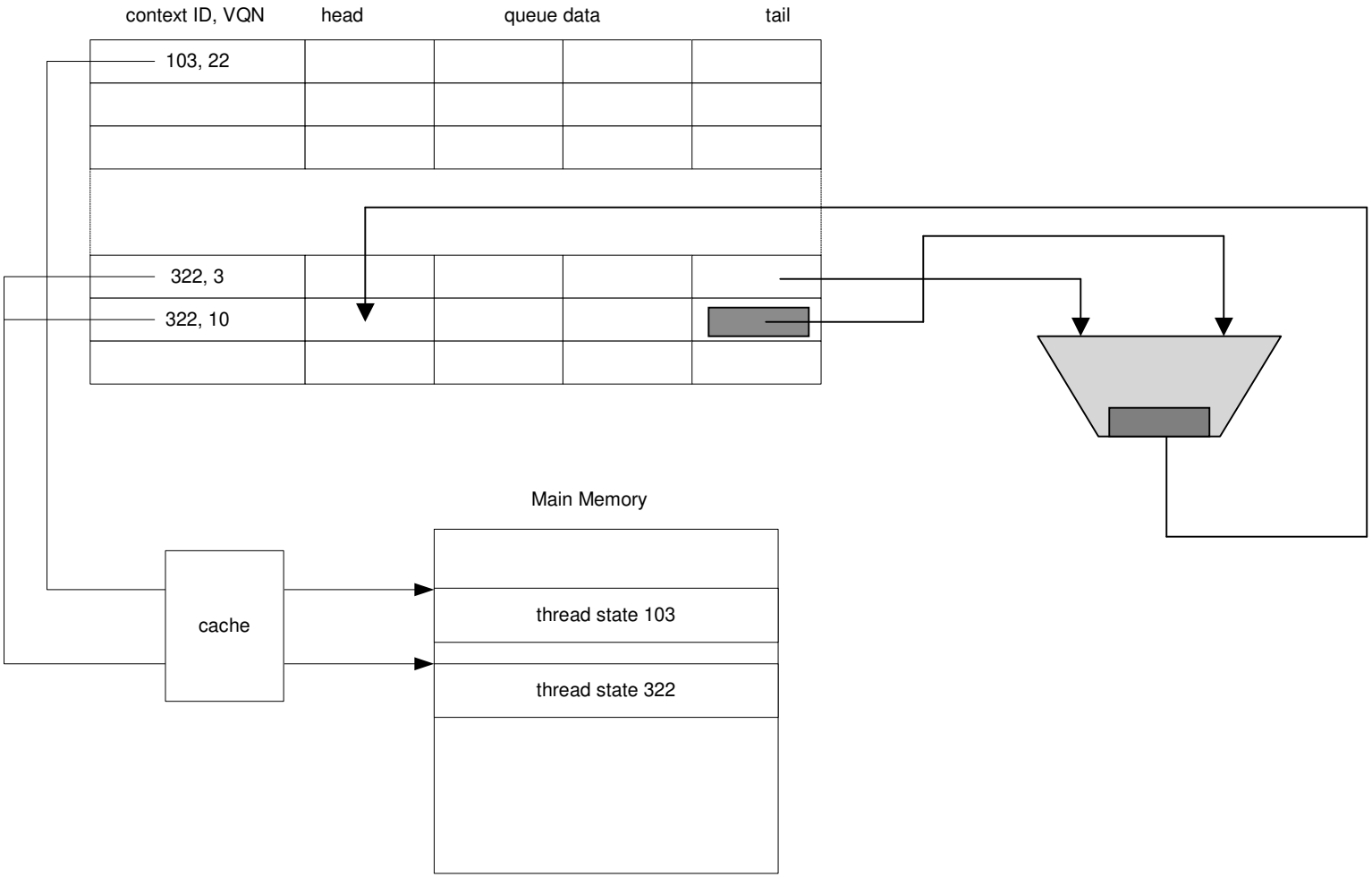
Normal Operation



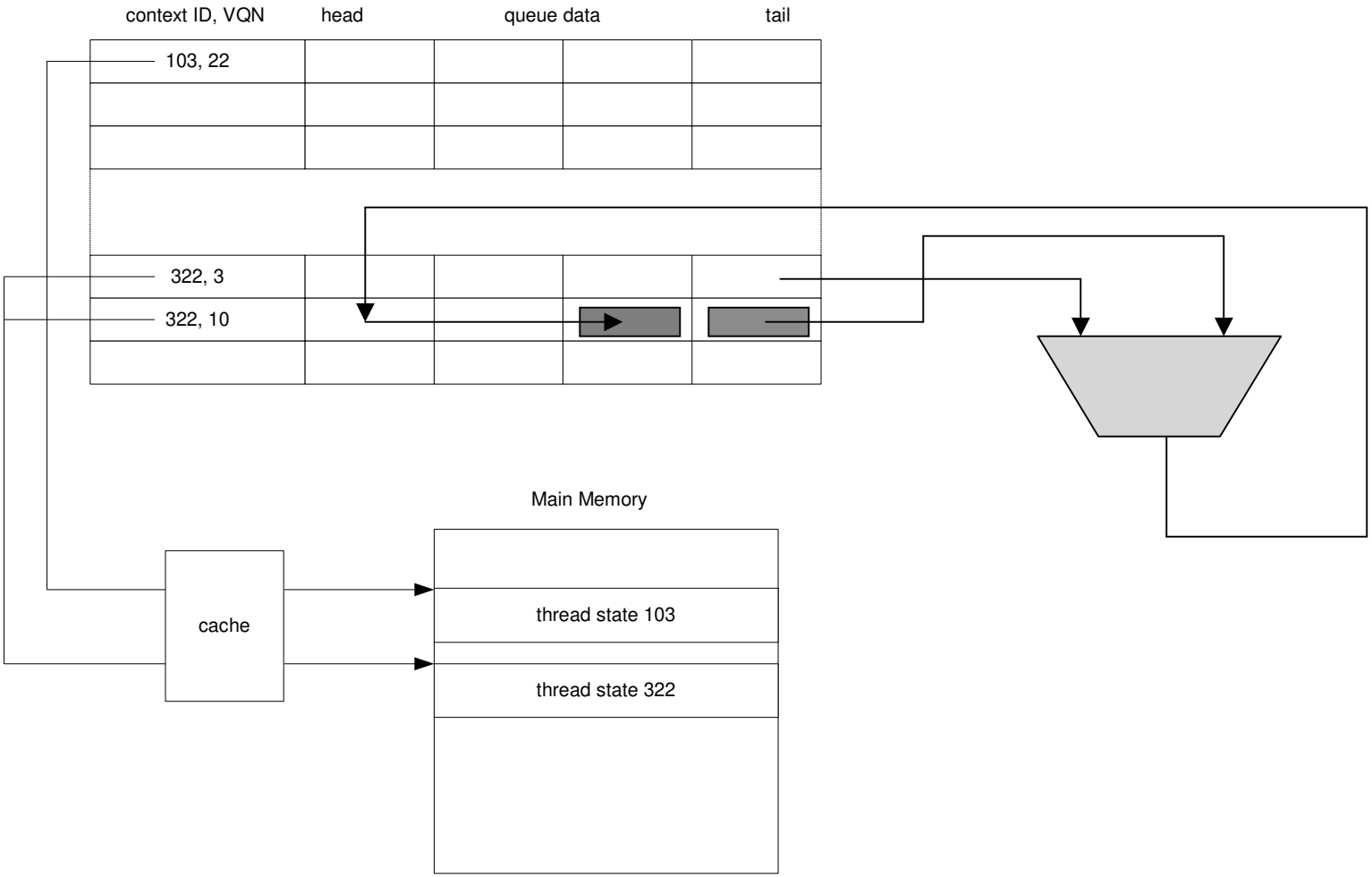
Normal Operation



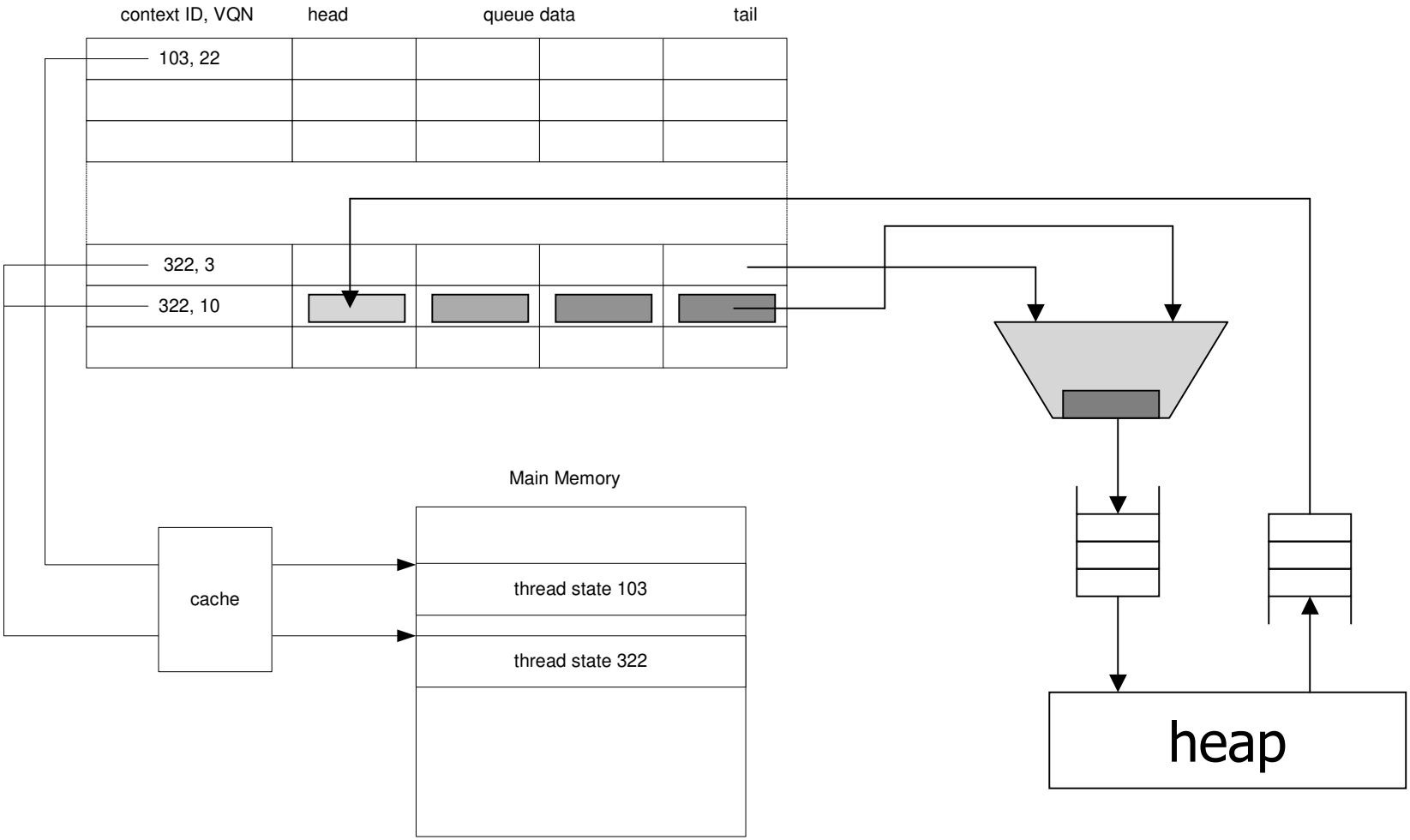
Normal Operation



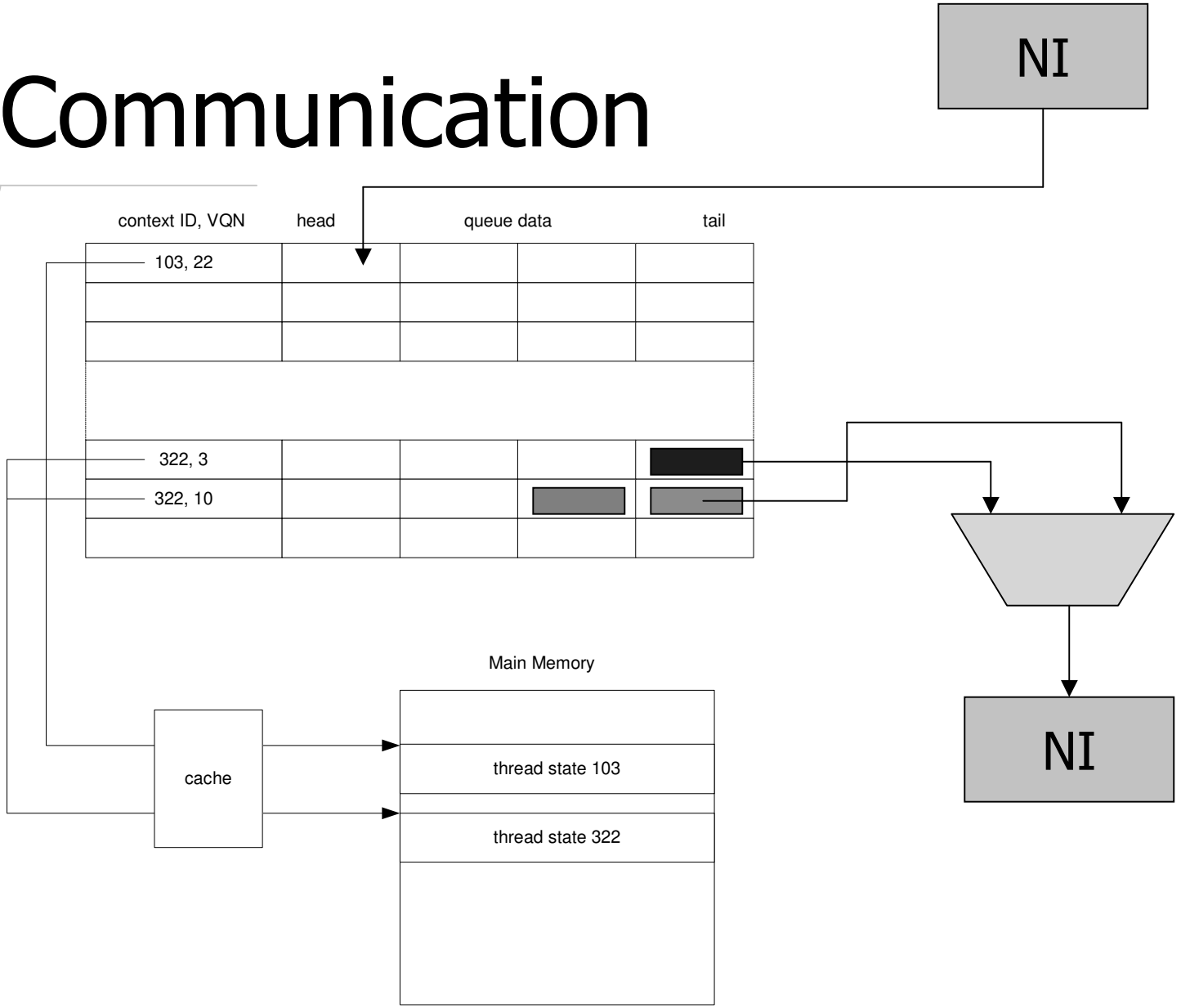
Normal Operation



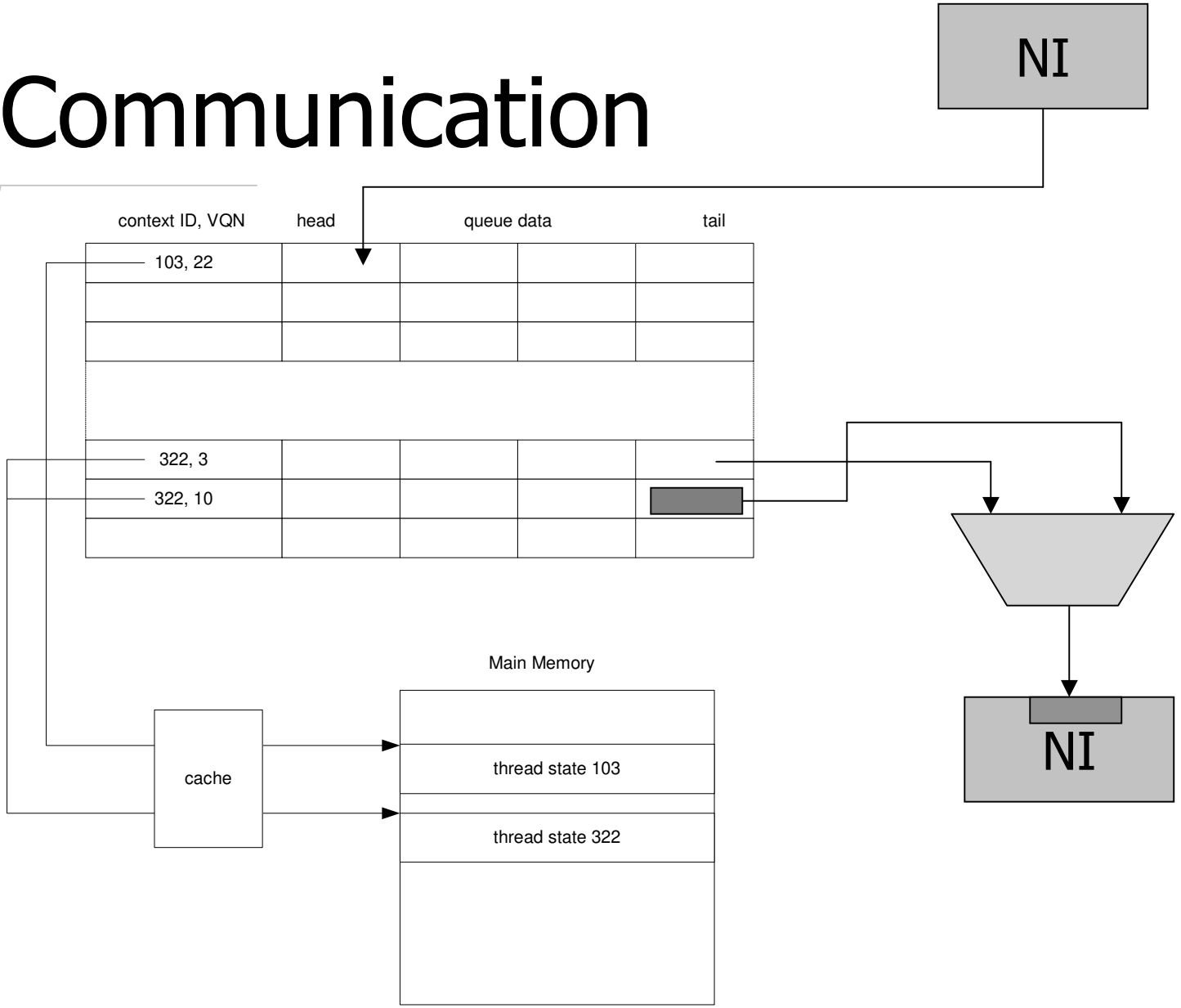
Queue Overflow



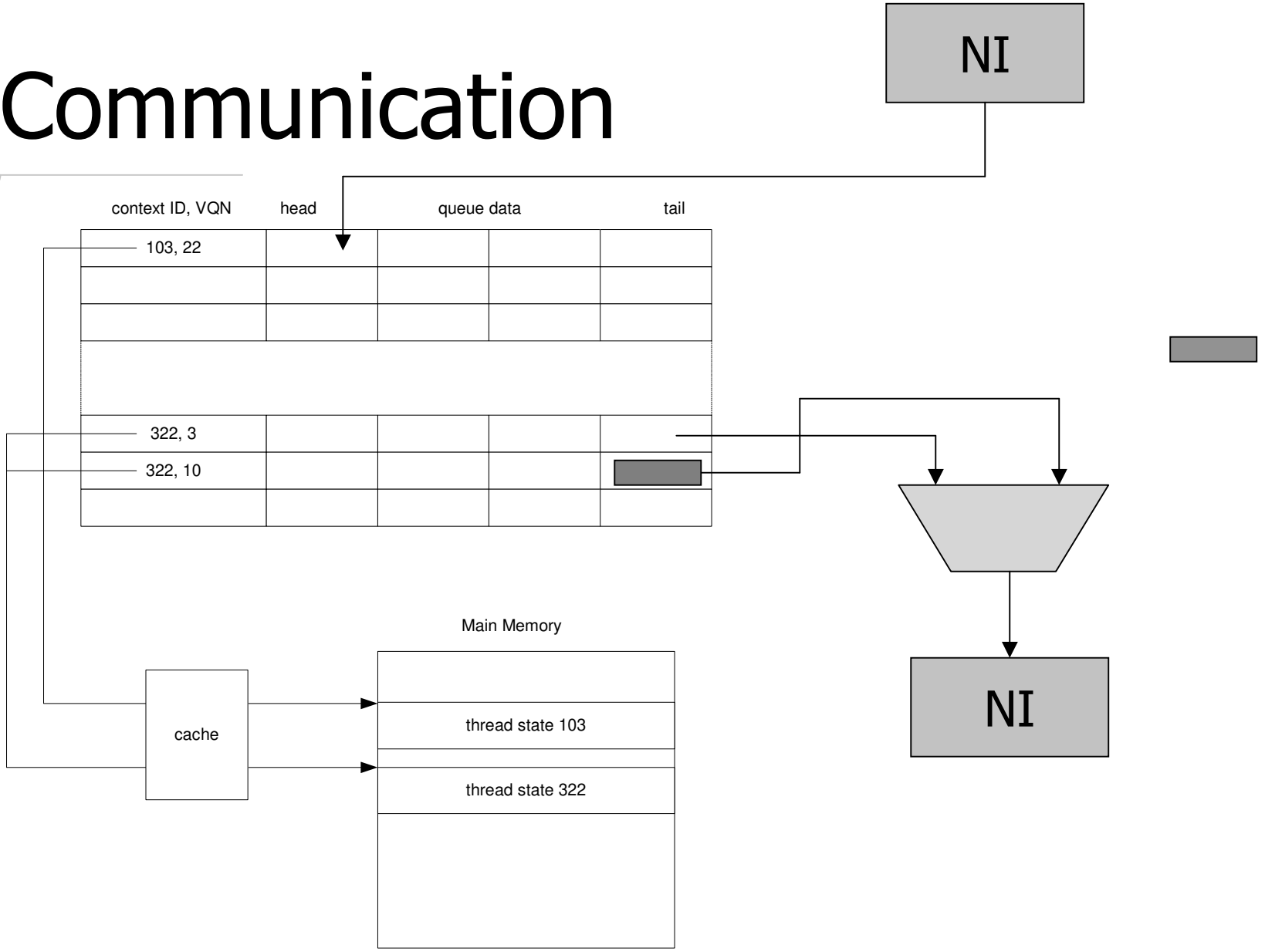
Communication



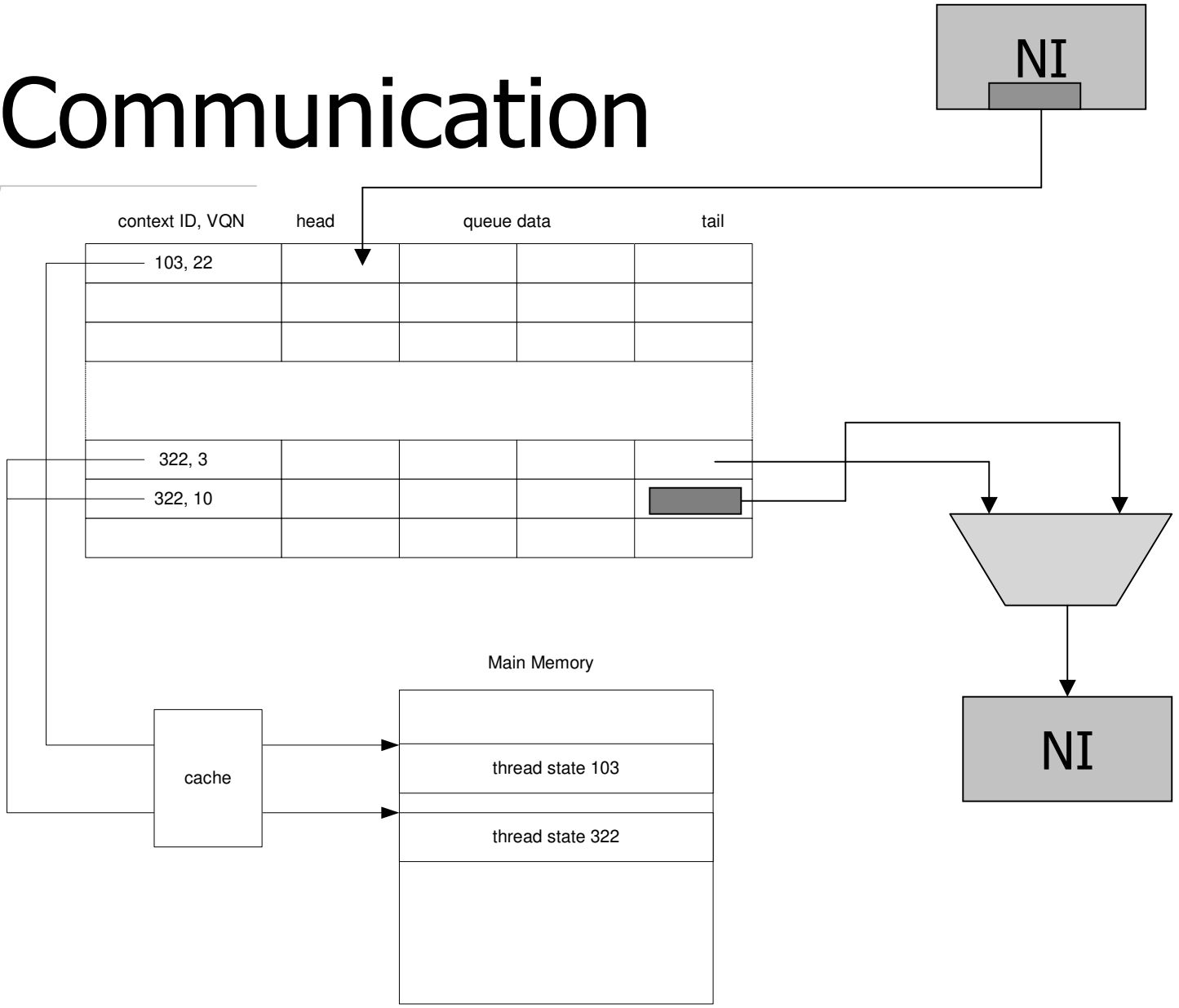
Communication



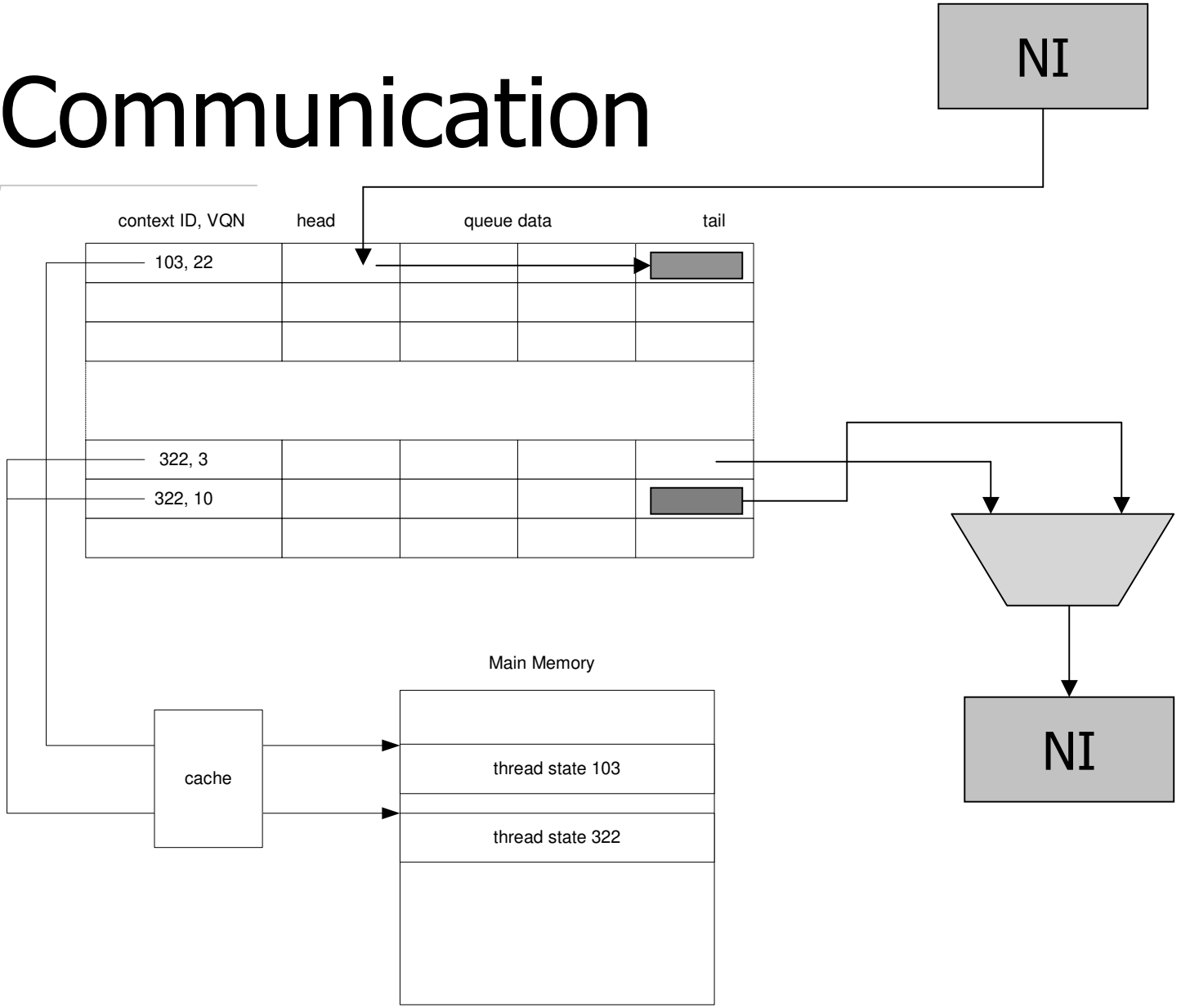
Communication



Communication



Communication



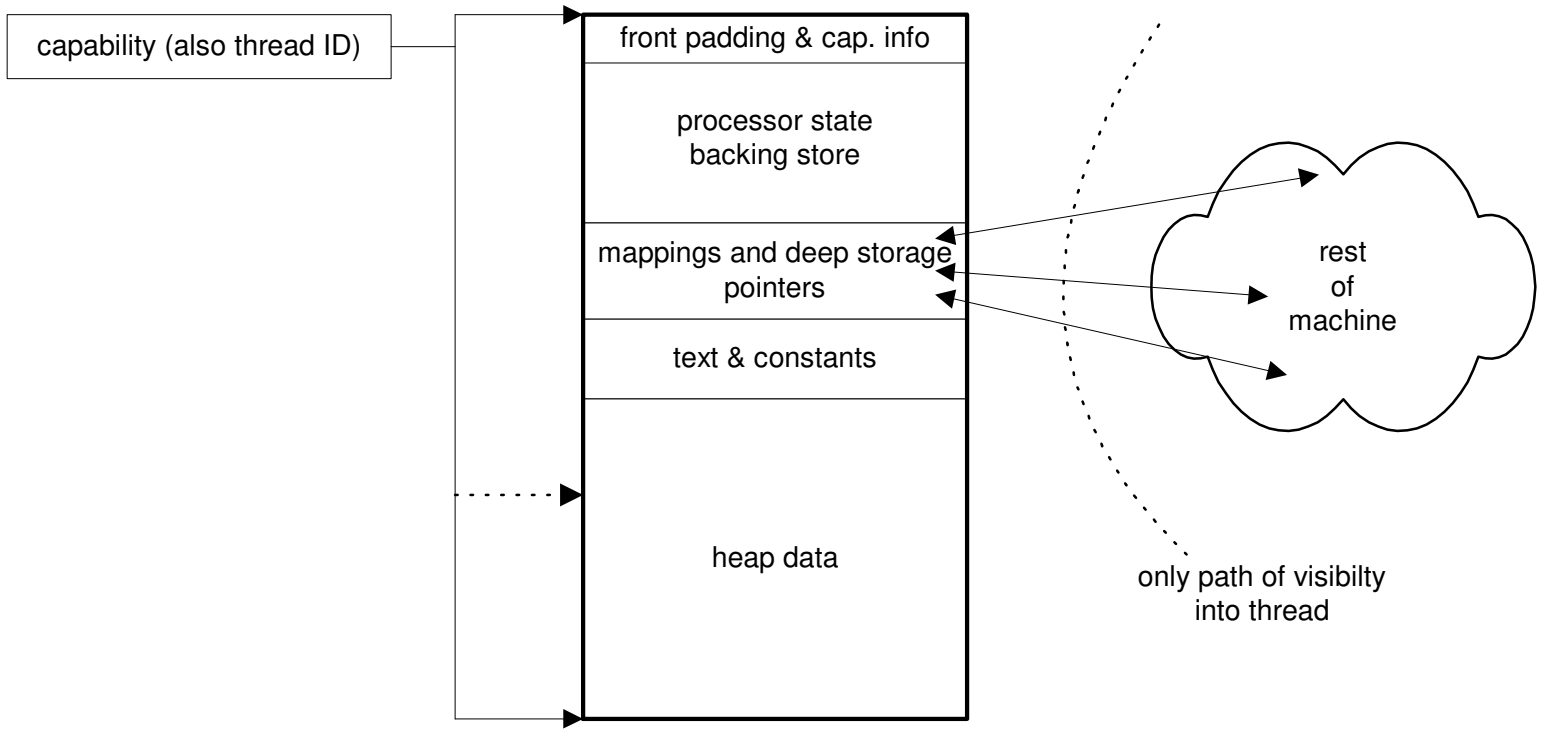
Practical Concerns

- Empty queues are not stored
  - Noted with a single bit instead
- Garbage collector runs on data stored in queue-space as well
  - Terminated threads have their name-space and storage-space recycled with time

Mobile Threads

- Each method invocation gets its own context ID
  - Thread-per-method
  - Calling convention only uses queues
  - Processor state and method data are stored together in the heap within a single capability
  - Enables fast migration
    - No large stack frames to move
    - Only queue maps to the thread have to be changed, or forwarding pointers left behind
    - Overhead equivalent BW of loading 2-3 L2 cache lines (vs. P4) beyond "standard" thread data

Mobile Threads



Migration and Placement

- Can be user-driven via language extensions
- Can be compiler driven via annotated code
- Can be automatic
  - Use the garbage collector mechanism!
    - GC is informed of hot spots and traffic patterns via hardware monitors
    - Named-state QF enables heap-based mechanisms to alter data and thread placement on the machine
    - GC also does adaptive load balancing based on node implementation performance
    - GC migrates data out of failing nodes

ADAM Properties

- Lightweight multi-threading
- Q cache instead of register file
- Capability-based memory

Couatl

- Based on 6.035's Espresso
- Object-Oriented
- Stripped down Java
- Freedom to change semantics and add new constructs

Simple Techniques

```
x = 4;
y = x + 3;
if (y == 7) {
   x = 7;
}
y = x + 8;
return y;
```

```
MOVEC 4, q1
ADDC @q1,3,q2
SEQ q2, q3
BRZ q3, after
MOVEC 7, @q1
after: ADDC q1, 8, q2
MOVE q2, q0
```

Necessary Analyses

- Live Variable
  - Objective: if *X* is live after read, use copy (@)
  - Standard backwards analysis
- Data Presence
  - Objective: if *X* is full before write, use clobber
  - Forward analysis
    - def(x) -> full(x)
    - use(x,dequeue) -> empty(x)

Procedure Calls

- Calling Convention:
  - Caller Side
    - Fork new thread
    - Map into new thread's q1
    - Enqueue return point (thread id, queue number)
    - Enqueue arguments
- Callee Side
    - Return sends data to return point, no control flow
    - Not an error to lack a return statement
- Semantics
  - Side effects are not guaranteed to have occurred until after return value used.

Memory Interface

- Map queue to memory for load or store

  MML q5,q6
  MOVE q2, q5
  MOVEC 0, q5
  where q2 contains capability, result in q6
- Decouples address computation from result retrieval
- Using memory queues looks like using normal queues

Object-Oriented Programming

- Data + Procedures that act on the data = Objects
- Good for locality
- Have object generate method threads "nearby" to keep computation local

Objects

- Object is a thread
- Handle is the thread id
- Call it a "server" thread
  - holds capability for object state
  - responds to method requests
  - since all requests go through object, could track / adapt to requests

Object Server Code

```
class Program {
  int double(int x) {
    return x * 2;
  }

  void main() {
    int c,d;

    c = 3;
    d = double(c);
    callout println(d);
  }
}
```

```
Program_init:
        MAPSQ q0, q1              ; sources of method invoke requests to q1
        ALLOCATEC 1, q2           ; allocate space for object ivars
        MMS q6, q7                ; open store queue
        MOVE @q2, q6              ; store to object root
        MOVEC 0, q6               ; store into reserved first position
        PROCID q7                 ; store object ref
        EEQ q6                    ; make sure it's saved
Program_dispatch:
        SGTC @q0, 2, q3           ; test top bound
        BRNZ q3, Program_error
        SHLC q0, 1, q3            ; multiply offset by two
        BREL q3                   ; jump to appropriate fork
        FORK Program_getrep, q4       ; autogenerated getrep method
        BR Program_dispatch2
        FORK method_Program_double, q4  ; method double
        BR Program_dispatch2
        FORK method_Program_main, q4    ; method main
Program_dispatch2:
        MAPQ q5, q1, q4           ; map to new thread
        MOVE q1, q5               ; send caller
        MOVE @q2, q5              ; send obj root
        PROCID q5                 ; send obj ref "this"
        EEQ q5                    ; make sure it's sent
        UNMAPQ q5                 ; disconnect
        BR Program_dispatch       ; go back to top
```

Method Calls

- Caller
  - Send method id to object
  - receive thread id in q0 (synchronous receive queue)
  - proceed with call like before
    - enqueue return point (thread id, queue number)
    - enqueue arguments

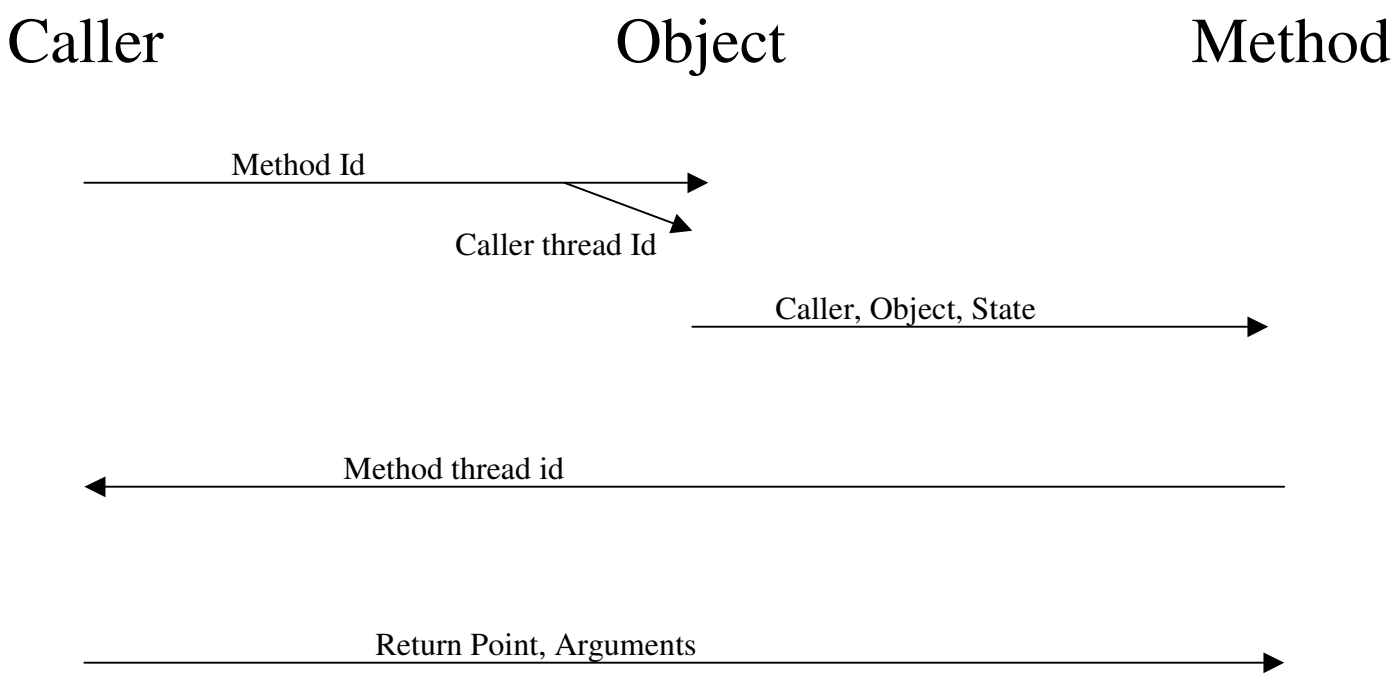
Method Calls

- Object Server
  - listen to q1, with sources mapped to q2
  - switch on received method id
  - fork new thread running method code
  - send caller id (from q2) to method
  - send “this” (server thread id) to method
  - send obj state capability to method

Method Calls

- Method
  - receive caller thread id
  - map into q0 of caller
  - send method thread id
  - receive object and object state capability from object server
  - receive return point and arguments from caller

Method Calls



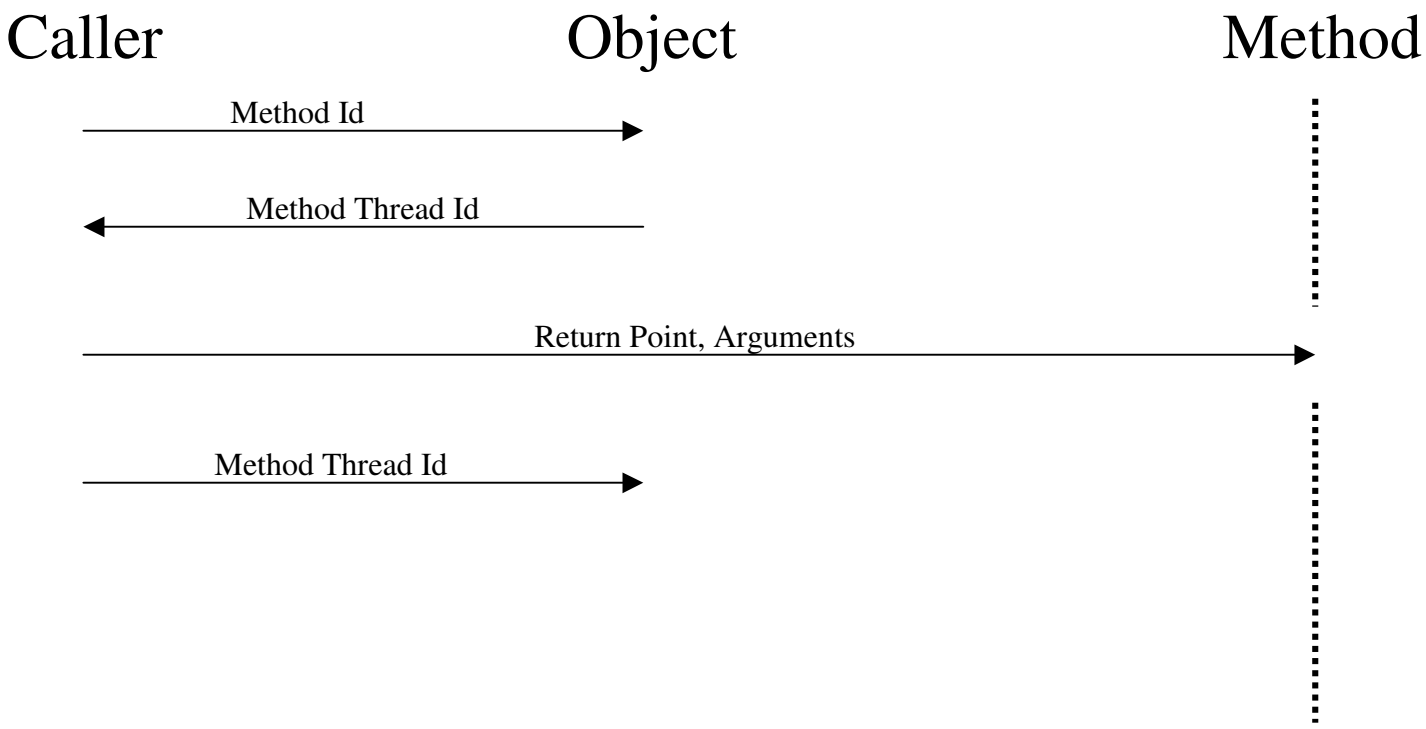
```
class Program {
  int double(int x) {
    return x * 2;
  }

  void main() {
    int c,d;

    c = 3;
    d = double(c);
    callout println(d);
  }
}
```

```
method_Program_double:
        MAPQ q5, q0, q1          ; open connection back to caller's dropQ
        PROCID q5                ; send method id
        MOVE q1, q2              ; save obj root
        MOVE q1, q4              ; save This
        MML q7, q8               ; open load queue
        MMS q10, q11             ; open store queue
        EEQ q5                   ; ensure msg sent
        MOVE q1, q3              ; thread to return to
        MOVE q1, q6              ; q in thread to return to
        MAPQI q5, @q6, @q3       ; map back
        MOVE q1, q13             ; move x to alloc'd q
block24:
        MULC q13, 2, q5          ; return x * 2
        HALT 5, 17               ; end of method
method_Program_main:
        MAPQ q5, q0, q1          ; open connection back to caller's dropQ
        PROCID q5                ; send method id
        MOVE q1, q2              ; save obj root
        MOVE q1, q4              ; save This
        MML q7, q8               ; open load queue
        MMS q10, q11             ; open store queue
        EEQ q5                   ; ensure msg sent
        MOVE q1, q3              ; thread to return to
        MOVE q1, q6              ; q in thread to return to
        MAPQI q5, @q6, @q3       ; map back
block26:
        MOVEC 3, q13             ; c = 3
        MAPQ q12, q0, @q4        ; d = this.double(...)
        MOVEC 1, q12
        MAPQ q12, q1, q0
        PROCID q12               ; return value to this thread
        MOVEC 14, q12
        MOVE q13, q12            ; args[0] = c
        EEQ q12
        PRINTQ q14               ; print variable: d
        HALT 5, 17               ; end of method
```

Synchronized Methods

- Objective: ensure only one copy of method (per object) executing at a time
- Solution: “durasive” method
- Object forks thread for method exactly once.
- Method code loops to top after completion
- Caller dequeues method id from object, sends arguments, then sends it back

Durasive Method



Streaming Constructs

- Idea: expose queue-based communication to the programmer
- Key Ideas
  - Simple Syntax
  - Allow use of abstraction

Streams

- A directed graph of computational modules
- Termination
  - Like to know when streaming computation completes, possibly with a result
- Inertness
  - Portions of the graph which aren't contributing shouldn't use computational resources

Stream

```
stream (closure) {
  stream-var-decls;
  inputs -> { code } -> outputs;
}
```

- A meander is a computational element in a stream.
- The closure contains variables copied to all meanders.
- Each meander consists of a set of input stream variables, a stream code block, and a set of output stream variables.
- A variable must appear as an output of exactly one meander.

Streams

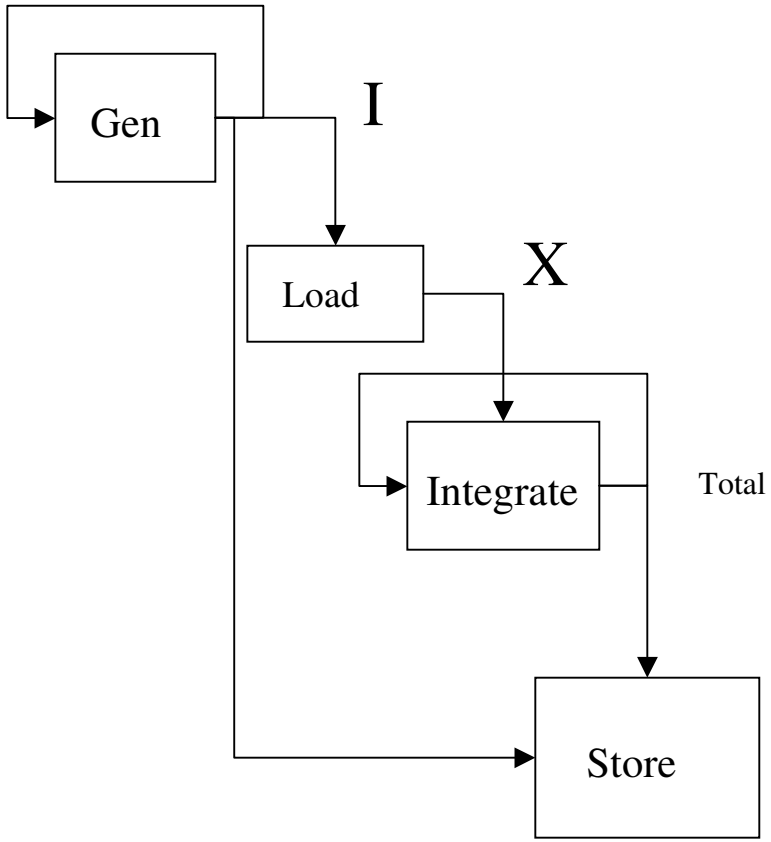
- Stream variables work differently
  - Each read consumes the value
    - y = x * x
      If x is a stream variable, multiplies consecutive values of x.
    - To avoid creating lots of temporaries explicitly: uses (*variables*) { …code… }
      Captures the current values of all variables, consuming that value if the variable is a stream variable.

Stream Example

```
int source[64];
Iterator it;
it = new Iterator().init(0,63);
stream (source,it) {
  int i;
  -> { i = it.next(); assert i; } -> i;
  i -> {
    uses (i) {
      source[i] = 0;
      if (i == 63) { return; }
    }
  } -> return;
}
```

- assert enqueues value for all receivers
- exhibits termination
- exhibits inertness due to Iterator implementation

Stream Example




```
class Program {
  void main() {
    int array[10];
    int dest[10];
    int i;

    // init array
    i = 0;
    while (i < 10) {
      array[i] = i;
      i = i + 1;
    }

    stream (array,dest) {
      int i,x,total;
      i(0) -> {
        int temp;
        temp = i;
        if (temp < 9) { i = temp + 1; assert i; }
      } -> i;
      i -> { x = array[i]; assert x; } -> x;
      x,total(0) -> { total = total + x; assert total; } -> total;
      total,i -> {
        int temp;
        temp = i;
        dest[temp] = total;
        if (temp == 9) { return; }
      } -> return;
    }

    i = 0;
    while (i < 10) {
      callout println(dest[i]);
      i = i + 1;
    }
  }
}
```

Stream Implementation

- Stream manager forks all meanders
- All stream variables allocated to queues beforehand
- Stream manager distributes thread ids of all immediately downstream meanders to each meander
- Static connections

Streaming Methods

- Name meanders to build layer of abstraction
- streaming foo(input int x, output int total)
- Stream variables are attached to declared inputs and outputs.
- Chosen verilog syntax

Streaming Method Example

```
class Program {
  int cap;

  void main() {
    cap = 10;
    stream () {
      int x;
      this.produceInts(.i(x));
      x -> { callout println(x); } -> ;
    }
    callout prints("Done!\n");
  }

  streaming produceInts(output int i) {
    int state,cap;
    cap = this.cap;
    state = 0;
    while (state < cap) {
      i = state; assert i;
      state = state + 1;
    }
  }
}
```

Future Directions

- Named Streams
  - Like Streaming Methods except body is like stream construct not sequential code.
  - Treat subgraph as a meander
  - Next level of abstraction

Future Directions

- Data Parallel operations
  - replicating meanders
- Dynamic and/or durasive streams
- Graphical programming interface for streams

Future Directions

Using Queue Depth

```
main:   PRINTS "Printing fib of "
        MOVEC 5, q2             ; fib(5)
        PRINTQ @q2               ; tell tester
        SLTC @q2, 2, q3         ; base case?
        BRNZ q3, base_fib       ; deal separately
        MOVEC 0, q4             ; init "last"
        MOVEC 1, q5             ; init "current"
        SUBC q2, 2, q2           ; offset to get index correct
fib:    ADD q4, @q5, q5         ; enQ(current, last + @current)
        MOVE q5, q4              ; move old current to last
        SUBC q2, 1, q2           ; n = n - 1;
        SGTC @q2, 0, q3         ; test termination
        BRNZ q3, fib             ; if not, go to top
done:   PRINTS "Result "        ; output result
        PRINTQ q5
        HALT 4, 76
base_fib:
        MOVE q2, q5
         BR done
```

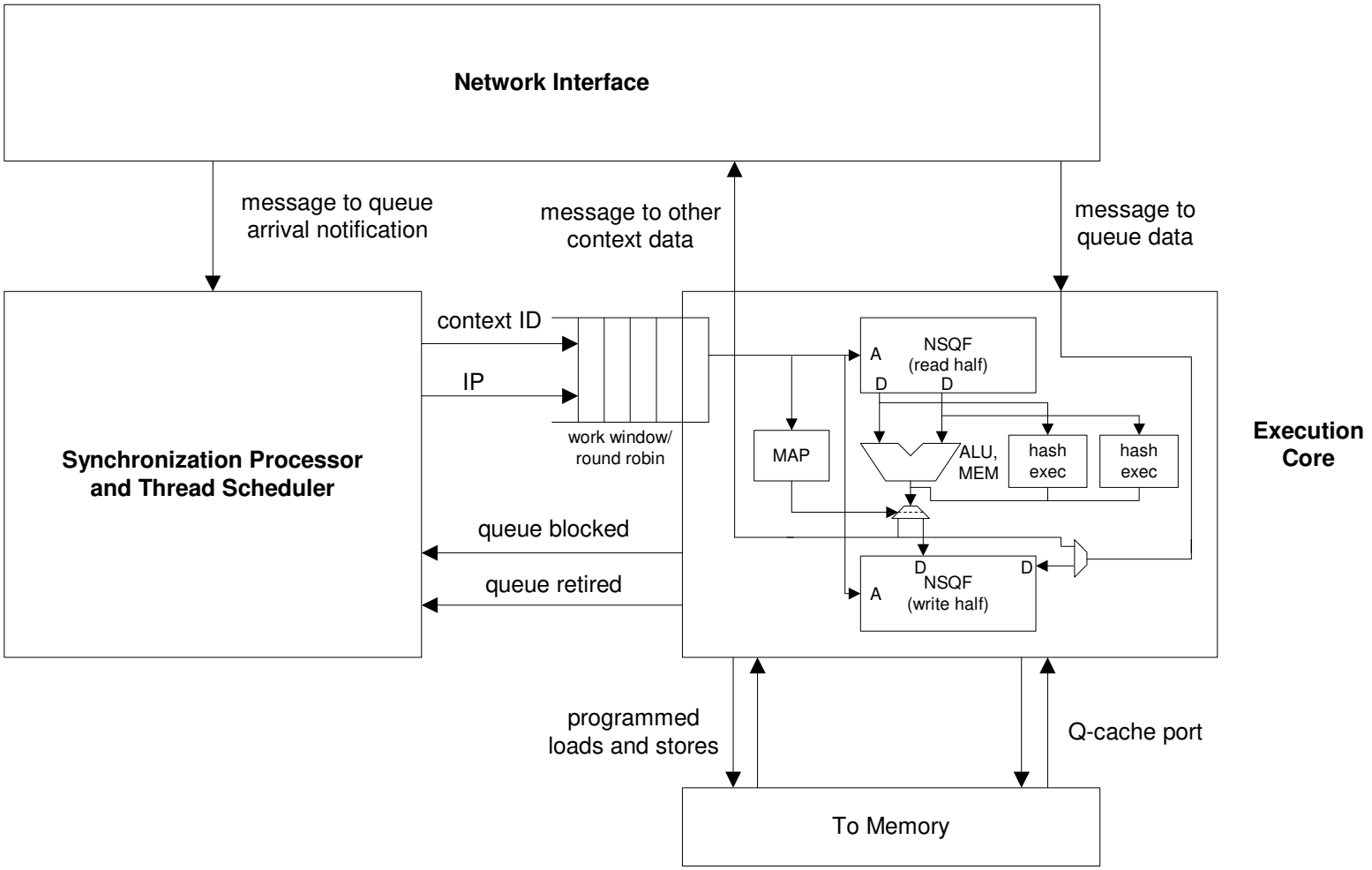
```
int Qfib(int n) {
  int last,current;

  if (n < 2) {
    return n;
  } else {
    last = 0;
    current = 1;
    while (n > 0) {
      current = last + current;  // force enQ of this
      last = current;
      n = n - 1;
    }
    return current;
  }
}
```

Conclusions

- Different, not hard, to compile to
- Allows compiler to pass more information about the program to the hardware
- Multi-word synchronization missing; compiler's biggest irritant
- Allocation of threads and memory to processors

Backup

Node Implementation



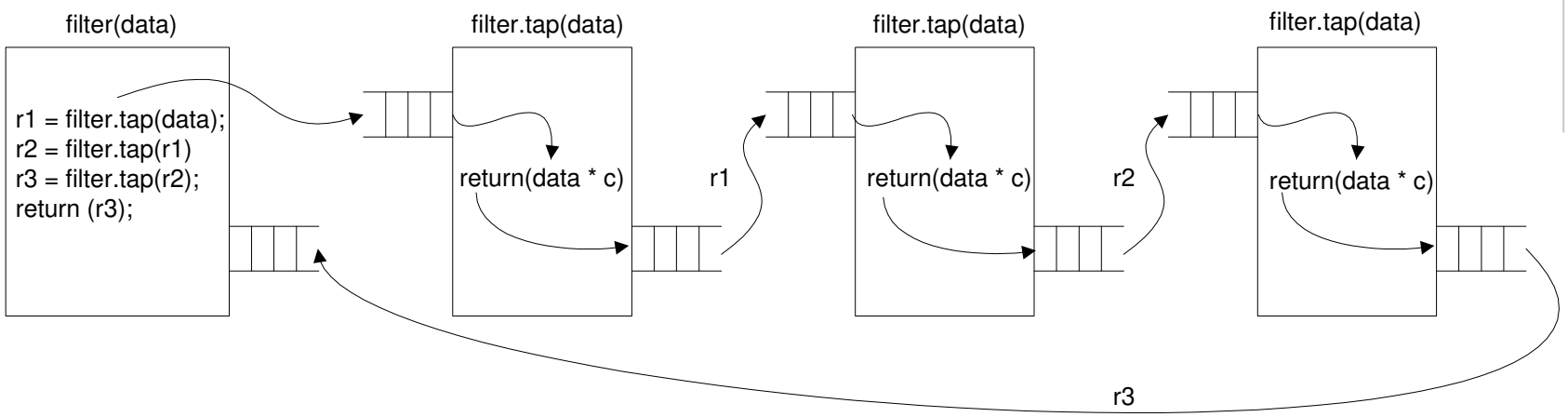
Compilation

- Q-Machine compilers can leverage existing technology
  - Queues are like the nodes of a dataflow graph, computed by existing compiler front ends
  - Compilers that use SSA notation are uniquely suited to target this architecture
  - Copy/clobber operations available on queues
  - Procedure calls are simple
  - Deadlock is dreadfully easy on this machine
    - Hardware-supported transaction and rollback mechanism can help solve this
  - "Software pipelining" techniques can be used to enhance performance, esp. on DSP code

Uniform Machine Model and Scalability

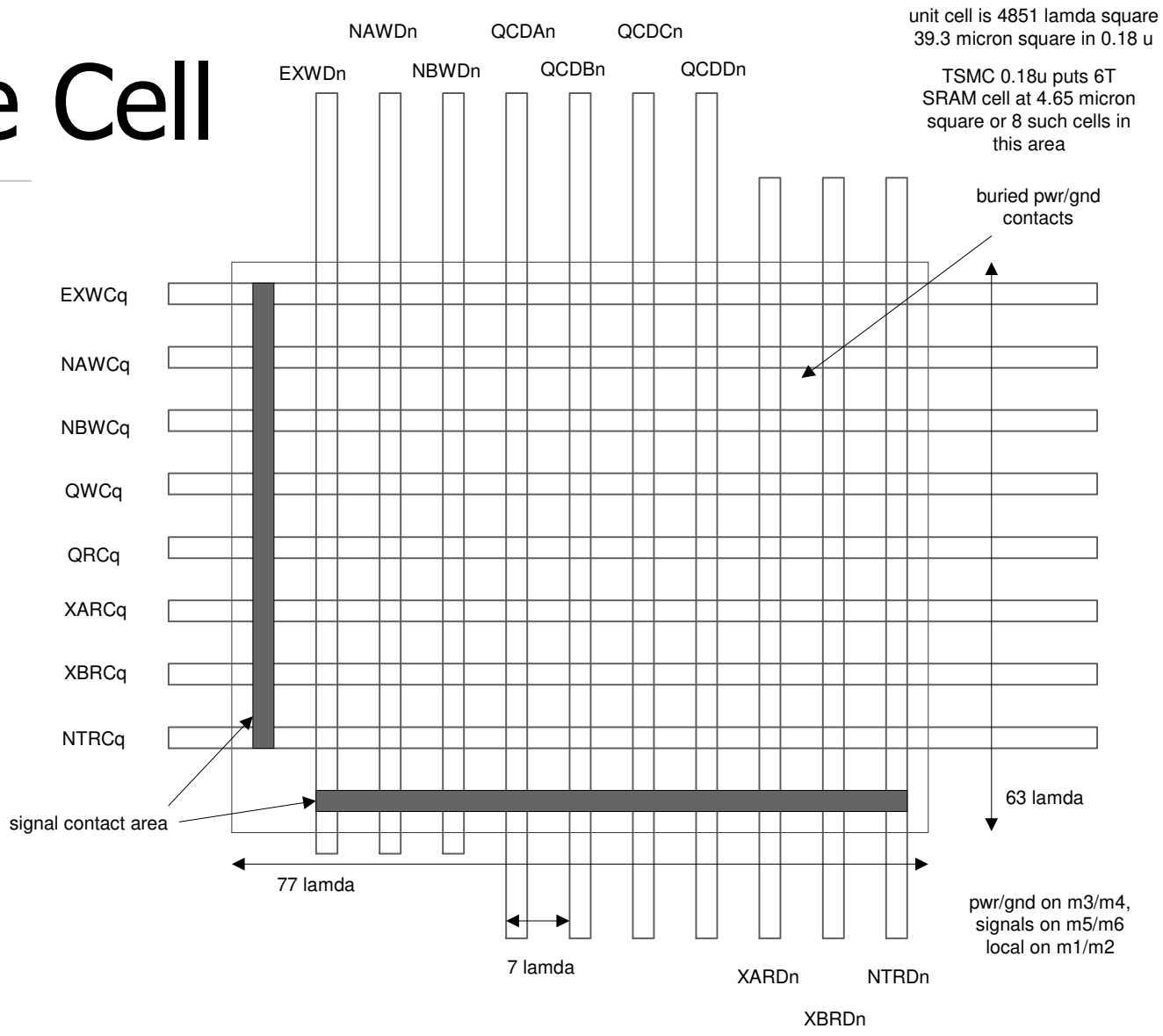
- Location independence of code and data enables scalability
  - Applications run on a wide range of machines without recompilation
  - A single application runs faster on a larger machine
  - Can partition the machine to run multiple jobs slower

Software Pipelining



- Simple "filter" example

Q-File Cell


NAWDn
QCDAn
QCDCn
EXWDn
NBWDn
QCDBn
QCDDn
unit cell is 4851 lamda square
39.3 micron square in 0.18 u
TSMC 0.18u puts 6T SRAM cell at 4.65 micron square or 8 such cells in this area
buried pwr/gnd contacts
EXWCq
NAWCq
NBWCq
QWCq
QRCq
XARCq
XBRCq
NTRCq
signal contact area
63 lamda
77 lamda
pwr/gnd on m3/m4, signals on m5/m6 local on m1/m2
7 lamda
XARDn
NTRDn
XBRDn