Memory Systems: Design and Implementation

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Reminders:
Quiz 2 Review tonight 7:30-9PM in 6-120
Quiz 2 Thursday 7:30-9:30PM in 50-340
Internal Cache Organization

Direct-Mapped Cache

Size of data per line: $2^b$ bytes
$2^{b-2}$ words

req address in bytes

Cache size $2^k$ lines

The last two bits of b are dropped because word is 4-byte aligned
n-way Set-associative Cache

Typically n is 4 to 8. High degree of associativity increases hardware complexity and access latency; assume n=2.

A cache line can be stored in either cache-array unit ⇒ fewer conflict misses.
Stores

- On a write hit
  - **Write-back policy**: write only to cache and update the next level memory when the line is evicted
  - **Write-through policy**: write to both the cache and the next level memory

For **Write-back** need status to be one of: **Invalid**, **Clean**, or **Dirty**

- On a write miss
  - **Allocate** – because of multi-word lines we first fetch the line, and then update a word in it
  - **No allocate** – cache is not affected, the Store is forwarded to memory

*We will use **Write-back** and **miss-allocate**.*
Processing Loads and Stores

Search the cache for the processor generated (byte) address

- Found in cache a.k.a. hit
  - Load: Return a copy of the word from the cache-line
  - Store: Update the relevant word in the cache-line

- Not in cache a.k.a. miss
  - Bring the missing cache-line from Main Memory
    May require writing back a cache-line to create space

Which line do we replace?

Update cache, and for a Load return word to processor
Replacement Policy

- **Direct Mapped Cache:**
  - No choice (each address maps to exactly one cache line)

- **N-way Set-associative cache:**
  - N choices for line to replace
  - LRU: evict line from least recently used way
  - Other policies exist but LRU is most common

For a 2-way set associative cache, can implement LRU with 1 bit of state per cache line.
Cache performance

- Cache designs have many parameters:
  - Cache size in bytes
  - Line size, i.e., the number of words in a line
  - Number of ways, the degree of associativity
  - Replacement policy

- A typical method of evaluating performance is by calculating the number of cache hits for a given set of cache parameters and a given set of memory reference sequences
  - Memory reference sequences are generated by simulating program execution
  - Number of hits, though fixed for a given memory reference pattern and cache design parameters, is extremely tedious to calculate (so it is done using a cache simulator)

  You will see examples in tomorrow’s recitations
Blocking vs. Non-Blocking cache

- **Blocking cache**
  - At most one outstanding miss
  - Cache must wait for memory to respond
  - Cache does not accept processor requests in the meantime

- **Non-blocking cache**
  - Multiple outstanding misses
  - Cache can continue to process requests while waiting for memory to respond to misses

We will discuss the implementation of blocking caches
Now we will implement a cache

- One-way, Direct-mapped
- Write-back
- Write-miss allocate
- blocking cache

Back-end memory (DRAM) is updated only when a line is evicted from the cache

Cache is updated on Store miss

Cache processes one request at a time
Memory, SRAM and DRAM interfaces

Interfaces assume fixed sizes for memory, DRAM, line, and addresses.

```verilog
interface Memory;
   method Action req(MemReq req);
   method ActionValue#(Word) resp;
endinterface

interface DRAM;
   method Action req(LReq req);
   method ActionValue#(Line) resp;
endinterface

interface SRAM#(numeric type indexSz, type dataT);
   method Action req(Bit#(indexSz) idx, MemOP op, dataT data);
   method ActionValue#(dataT) resp;
endinterface
```

Size of SRAM = $2^{\text{indexSz}}$ data elements

```verilog
typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq...
typedef struct {MemOp op; LAddr laddr; Line line;} LReq...
```

no response for Stores

Interfaces assume fixed sizes for memory, DRAM, line, and addresses.

```verilog
interface Memory;
   method Action req(MemReq req);
   method ActionValue#(Word) resp;
endinterface

interface DRAM;
   method Action req(LReq req);
   method ActionValue#(Line) resp;
endinterface

interface SRAM#(numeric type indexSz, type dataT);
   method Action req(Bit#(indexSz) idx, MemOP op, dataT data);
   method ActionValue#(dataT) resp;
endinterface
```

Size of SRAM = $2^{\text{indexSz}}$ data elements

```verilog
typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq...
typedef struct {MemOp op; LAddr laddr; Line line;} LReq...
```
Cached Memory Systems

Processor accesses are for words while DRAM accesses are for lines

mkDRAM and mkSRAM primitives are given:

```
DRAM dram <- mkDRAM;
SRAM#(LogNumEntities, dataT) sram <- mkSRAM;
```

To avoid type clutter we assume that DRAM has 64Byte (16 word) lines and uses line addresses
Cache Interface

interface Cache#(numeric type logNLines);
    method Action req(MemReq req);
    method ActionValue#(Word) resp();
    method ActionValue#(LReq) lineReq;
    method Action lineResp(Line r);
endinterface

Notice, the cache size does not appear in any of its interface methods, i.e., users do not have to know the cache size

module mkMemory(Memory);
    DRAM dram <- mkDRAM;
    Cache#(LogNLines) cache <- mkBlockingCache;
    ...

Cache Organization

- cache-array unit encapsulates data, tag and status arrays, which are all made from SRAM
- Need queues to communicate with the back-end memory
- hitQ holds the responses until the processor picks them up
- state and current req registers hold the request and its status while it is being processed
Suppose a request gets a hit in the cache-array unit
  - Load hit returns a word
  - Store hit returns nothing (void)

In case of a miss, the line slot must have been occupied; all the data in the missed slot is returned so that it can be written to the back-end memory if necessary.

When the correct data becomes available from the back end memory, the cache-array line is updated.
interface CAU#(numeric type logNLines);
    method Action req(MemReq r);
    method ActionValue#(CAUResp) resp();
    method Action update(CacheIndex index, TaggedLine newline);
endinterface

typedef struct{HitMissType hitMiss; Word ldValue;
              TaggedLine taggedLine;} CAUResp;
typedef enum{LdHit, StHit, Miss} hitMiss;
typedef struct{Line line; CacheStatus status; CacheTag tag;
              } TaggedLine;
module mkCAU(CAU#(LogNLines));
    Instantiate SRAMs for dataArray, tagArray, statusArray;
    Reg#(CAUStatus) status <- mkReg(Ready);
    Reg#(MemReq) currReq <- mkRegU; //shadow of outer currReq
    method Action req(MemReq r);
        initiate reads to tagArray, dataArray, and statusArray;
        store request r in currReq
    endmethod
    method ActionValue#(CAUResp) resp;
        Wait for responses for earlier requests
        Get currTag, idx, lineOffset from currReq.addr and do tag match
        In case of a Ld hit, return the word; St hit, update the word;
        In case of a miss, return the data, tag and status;
    endmethod
    method Action update(CacheIndex index, TaggedLine newline);
        update the SRAM arrays at index
    endmethod
endmodule
Blocking cache

module mkBlockingCache(Cache#(LogNLines));
    CAU#(LogNLines) cau <- mkCAU();

    FIFO#(Word)    hitQ <- mkFIFO;
    Reg#(MemReq)   currReq <- mkRegU;
    Reg#(ReqStatus) state <- mkReg(Ready);
    FIFO#(LReq)    lineReqQ <- mkFIFO;
    FIFO#(Line)    lineRespQ <- mkFIFO;
endmodule

Rules to process a request including cache misses
Ready -> WaitCAUResp ->
    (if miss SendReq -> WaitDramResp) -> Ready

method Action  req(MemReq req) ... 
method ActionValue#(Word)  resp() ... 
method ActionValue#(LReq)  lineReq ... 
method Action  lineResp(Line r) ... 
endmodule
Blocking cache methods

method Action req(MemReq r) if (state == Ready);
    currReq <= r; cau.req(r);
    state <= WaitCAUResp
endmethod

method ActionValue#(Word) resp;
    hitQ.deq(); return hitQ.first;
endmethod

method ActionValue#(LReq) lineReq();
    lineReqQ.deq(); return lineReqQ.first();
endmethod

method Action lineResp (Line r);
    lineRespQ.enq(r);
endmethod
Blocking cache rules

rule waitCAUResponse;

let x <- cau.resp;

case (x.hitMiss)

<table>
<thead>
<tr>
<th>Case</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>LdHit</td>
<td>begin Word v = x.ldValue; hitQ.enq(v); state &lt;= Ready; end</td>
</tr>
<tr>
<td>StHit</td>
<td>state &lt;= Ready;</td>
</tr>
</tbody>
</table>
| Miss       | begin let oldTaggedLine = x.taggedLine; extract cstatus, evictLaddr, line from oldTaggedLine
|            | if (cstatus == Dirty) begin // writeback required
|            | lineReqQ.enq(LReq{op:St, laddr:evictLaddr, line:line}); state <= SendReq; |
|            | end else begin // no writeback required
|            | extract newLaddr from currReq                                           |
|            | lineReqQ.enq(LReq{op:Ld, laddr:newLaddr, line: ?}); state <= WaitDramResp; |
| end        |                                                                         |
| endcase    |                                                                         |
| endrule    |                                                                         |

Lab7
Blocking cache rules

rule waitDramResponse

rule waitDramResponse(state == WaitDramResp);
let line = lineRespQ.first(); lineRespQ.deq();
get idx, tag, wOffset from currReq.addr;
if (currReq.op == Ld) begin
hitQ.enq(line[wOffset]);
cau.update(idx,
    TaggedLine {line: line, status: Clean, tag: tag});
end else begin // St
line[wOffset] = currReq.data;
cau.update(idx,
    TaggedLine {line: line, status: Dirty, tag: tag});
end
state <= Ready;
endrule
Hit and miss performance

- **Hit**
  - Directly related to the latency of L1
  - 1-cycle latency with appropriate hitQ design

- **Miss**
  - No evacuation: DRAM load latency + 2 X SRAM latency
  - Evacuation: DRAM store latency + DRAM load latency + 2 X SRAM latency

Adding a few extra cycles in the miss case does not have a big impact on performance
Lab 7: 2-way Set-Associative Cache

- Need to instantiate cache-array unit twice
- Tag matching has to be done for each cache-array unit
- We can get a hit in at most one cache
- Miss detection requires checking both cache-array units; a victim for throwing out has to be selected
- Once the victim has been selected Cache-miss processing is the same
- cache-array unit should be used as is
Take Home problem

- Describe a situation where 1-Way Direct Map cache would get a miss but a 2-Way cache would not.