Memory System Implementation

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Code for the lecture is available on the course website under the code tab
The simplest implementation

Direct-Mapped Cache

- Tag
- Index
- Offset
- Cache line number
- req address in bytes

Tag | Index | Offset
---|---|---

Cache line number

- V
- t
- k
- b

Cache Line

Data Word

Hit

assume

- b = 6
- k = 9
- t = 17

April 5, 2018

Conflict Misses

In a direct map cache, a cache-line can be stored in only one specific cache slot. Thus, if the cache slot corresponding to the load address is occupied (even if there are other free slots) that cache line has to be thrown out.

Can we do better?

Yes, a set-associative cache.
n-way Set-associative Cache

Typically $n$ is 4 to 8. High degree of associativity increases hardware complexity and access latency

High degree of associativity increases hardware complexity and access latency.
Loads

Search the cache for the processor generated (byte) address

Found in cache
a.k.a. **hit**
Return a copy of the word from 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 return word to processor
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

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

Typically one uses either Write-back-Allocate-on-miss policy or Write-through-No-allocate-on-miss policy
Replacement Policy

To bring in a new cache line, usually another cache line has to be thrown out. Which one?

- Direct mapped cache: No choice
- N-way set associative cache: Choice of policy
  - One that is not dirty, i.e., has not been modified. In I-cache all lines are clean;
  - If there is still a choice of more than one then Least Recently Used (LRU)? Most Recently Used? Random?

How much is performance affected by the choice?

Difficult to know without benchmarks and quantitative measurements.
Blocking vs. Non-Blocking cache

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

Non-blocking cache:
- Multiple outstanding misses
- Cache can continue to process requests while waiting for memory to respond to misses
Memory Systems: Two Implementations

The whole memory is implemented using SRAM. The memory has a small SRAM cache which is backed by much bigger DRAM memory.

Processor accesses are for words while DRAM accesses are for lines.

`mkDRAM` and `mkSRAM` primitives are given:

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

To avoid type clutter we assume that DRAM has 64Byte lines and uses line addresses.
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

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...
module mkSMemory(Memory);
SRAM#(LogNumWords, Word) sram <- mkSRAM;

function Bit#(LogNumWords) getSRAMIndex(Word addr) =
    truncate(addr >> 2);

method Action req(MemReq r);
    sram.req(getSRAMIndex(r.addr), r.op, r.data);
endmethod

method ActionValue#(Word) resp();
    let d <- sram.resp;
    return d;
endmethod
endmodule

typedef 30 LogNumWords;
will create a memory of $2^{30}$ 32-bit words

Notice, this module is just a wrapper for the instantiated SRAM
and does not do much beyond trimming address bits
Now we will implement a cache

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

Backend memory is updated only when a dirty line is evicted from the cache
Memory System with Caches

Cache Interface

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

processor-side methods:
- fixed size words and (byte) addresses
back-side methods:
- fixed size cache lines and line-addresses

```vhdl
module mkProc(Empty);
  DRAM#(logNumLines) dram <- mkDRAM;
  Cache#(CacheLogNumLines) cache <- mkBlockingCache;
  ...
```
Cache Organization

- way-cache 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
Blocking cache

module mkBlockingCache(Cache#(LogNumCacheLines));
    WayCache#(LogNumCacheLines) wayCache <- mkWayCache();

    FIFO#(Word)  hitQ <- mkSizedFIFO(1);
    Reg#(MemReq) currReq <- mkRegU();
    Reg#(ReqStatus) state <- mkReg(Ready);
    FIFO#(LReq)  lineReqQ <- mkSizedFIFO(1);
    FIFO#(Line)  lineRespQ <- mkSizedFIFO(1);

    rule waitWayCacheResponse ...
    rule waitDRAMResponse ...

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

endmodule

State Elements

Ready -> WaitWayCacheResp ->
       (if miss SendReq -> WaitDRAMResp) -> Ready
Blocking cache methods

method Action req(MemReq r) if (state == Ready);
   currReq <= r; wayCache.req(r);
   state <= WaitWayCacheResp;
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

We will first design the way-cache and then examine the rules in the blocking cache
way-cache Interface

interface WayCache#(numeric type logNumCacheLines);
  method Action req(MemReq r);
  method ActionValue#(WayCacheResp) resp();
  method Action update(CacheIdx idx, TaggedLine newline);
endinterface

typedef union tagged {
  Word LdHit;
  void StHit;
  TaggedLine Miss ;
} WayCacheResp deriving(Eq,FShow);

typedef struct {
  Line line; CacheStatus status; CacheTag tag; } TaggedLine
deriving(Eq,FShow);
module.mkWayCache(WayCache#(LogNumCacheLines));
     SRAM#(LogNumCacheLines, Line) dataArray <- mkSRAM;
     SRAM#(LogNumCacheLines, CacheTag) tagArray <- mkSRAM;
     SRAM#(LogNumCacheLines, CacheStatus) statusArray <- mkSRAM;
     Reg#(MemReq) currReq <- mkRegU; // shadow of currReq

method.Action req(MemReq r);
    currReq <= r; let idx = getIdx(r.addr);
    tagArray.req(idx, Ld, ?); dataArray.req(idx, Ld, ?);
    statusArray.req(idx, Ld, ?);
endmethod

method.ActionValue#(WayCacheResp) resp; // next slide
method.Action update(CacheIdx idx, TaggedLine tline);
    tagArray.req(idx, St, tline.tag);
    dataArray.req(idx, St, tline.line);
    statusArray.req(idx, St, tline.status);
endmethod
endmodule
way-cache: response method

```haskell
method ActionValue#(WayCacheResp) resp;
    CacheTag tag <- tagArray.resp;
    CacheStatus status <- statusArray.resp;
    Line line <- dataArray.resp;
    // get currTag, idx and wOffset from currReq.addr
    // do tag match
    if (currTag == tag && status != Invalid) begin // hit
        if (currReq.op == Ld) begin // Ld Hit
            return (tagged LdHit line[wOffset]);
        end else begin // Store hit
            line[wOffset] = currReq.data;
            dataArray.req(idx, St, line);
            statusArray.req(idx, St, Dirty);
            return (tagged StHit);
        end
    end else // Miss Ld or St
    return (tagged Miss TaggedLine
            {line:line, status:status, tag:tag});
endmethod
```
Blocking cache rules

**rule waitWayCacheResponse**

```haskell
rule waitWayCacheResponse (state == WaitWayCacheResp);
  let x <- wayCache.resp;
  case (x) matches
    tagged LdHit .v : begin
      hitQ.enq(v);
      state <= Ready;
    end
    tagged StHit : state <= Ready;
    tagged Miss .oldTaggedLine: begin
      // extract status, tag and line from oldTaggedLine
      let lineAddr = {tag, getIdx(currReq.addr)};
      if (status == Dirty) begin
        // writeback required
        lineReqQ.enq(LReq{op:St, lAddr:lineAddr, lData:line});
        state <= SendReq;
      end else begin
        // no writeback required
        lineReqQ.enq(LReq{op:Ld, lAddr:lineAddr, lData:?});
        state <= WaitDramResp;
      end
    endcase
endrule
```

Is this guard needed?
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]);
    wayCache.update(idx, TaggedLine {line: line, status: Clean, tag: tag});
end else begin // St
    line[wOffset] = currReq.data;
    wayCache.update(idx, TaggedLine {line: line, status: Dirty, tag: tag});
end

state <= Ready;
endrule
```

Revisit the update rule (slide 18)
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*
Lab6: 2-way Set-Associative Cache

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