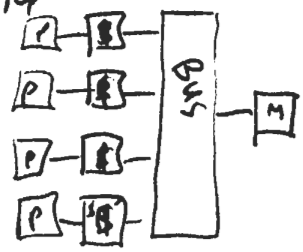
Competitive Snoopy Caching; & Directory-Based Caching

Oct 27, 2003
6.895 Bradley C. Kuszmaul

Today we analyse an algorithm that uses an MESI-like protocol.

Setup:




- Caches and memory can see all messages
- Only one ($ or M) can send a message at a time
- Processes (P) talk to caches ($) which talk to bus.

Locality of reference: ~~Spatial locality of a program is~~

Spatial locality is the property that if a processor reads location v, it is likely to operate on v+1 or other locations near v.

Programs exhibit spatial locality => transmit blocks of size p.

Here

Definitions sidebar
p = block size

if b=1, fetch costs 2 cycles
send address →
send reply ←

if b larger, it takes b+1 cycles, saving 2x for reasonably large b.

Blocks stored at >1 processor sometimes. Reduces read cost

To update a replicated block requires communication
- Either invalidate the other copies } which is right?
- or update them

Two common strategies
Exclusive Write: Invalidate everything else
Pack-rat : Update everything else

Worst-case behavior of exclusive-write: Here is a bad case

$P_0$ ~~writes~~ v ~~loops~~ repeat k times
$P_1 \cdots P_{n-1}$ reads v $P_0$ writes v
$P_1 \cdots P_{n-1}$ read v

reads => total cost of $k \cdot (n-1) \cdot p$

optimal ~~strategy update all (pack rat)~~
strategy for this sequence is pack rat

cost = $k + np$ (all reads cache hit + k writes)

$$\lim_{K\to\infty} \frac{K\cdot(n-1)\cdot p}{K+np} = (n-1)\cdot p$$

(Review:

$$\lim_{a\to\infty} \frac{a\cdot b}{a+c} = \lim_{a\to\infty} \frac{(a+c)b - cb}{a+c}$$

$$= \lim_{a\to\infty} \frac{(a+c)b}{a+c} - \lim_{a\to\infty} \frac{cb}{a+c}$$

$$= b - 0$$

$$= b$$ )

Bad (worst?) case analysis for packet:

Po waits
P1 waits
repeat w times
Po writes

Total cost $\geq w$

Optimal is ~~exclusive write, cost of 2p++~~
drop after waiting for cost of $2p$

$\lim_{w\to\infty} \frac{w}{2p}$ is unbounded.

---

Strategy: Competitive algorithm

Defn: an on-line algorithm A is c-competitive if
$\exists d$
$\forall$ sequences of inputs
optimal $\geq \frac{1}{c}\cdot \text{cost}(A) + d$ ($\leq$ OPTIMAL_OFFLINE_ALGORITHM)
for that sequence.
$c + d$ constant or independent of the sequence

Defn: Strongly c-competitive => no better c.

Another algorithm: Goodman's algorithm

On first write, Update everything else

On second write, invalidate everything else

Analysis: ? But it is fishy to do something once.
Why not twice?

STRATEGY: COMPETITIVE ALGORITHM

Defn: An ~~on~~ online ~~algorithm~~ $A$, is <u>c-competitive</u> (or <u>competitive</u>) iff

$\exists c$ $\forall$ sequences of inputs

~~cost~~ cost$(A) \leq$ ~~opt~~ cost(OPTIMUM OFFLINE ALG.

Defn: An algorithm is an <u>online</u> algorithm if ~~it must make its~~
its behavior depends only on the inputs

~~Defn: An Algo~~

Consider an algorithm that takes a sequence of inputs & produces a sequence of outputs

Defn Such an algo is on-line if it can produce the $i$th output having seen only the first $i$ inputs, otherwise offline.

~~Intuition: It is offline if it looks at all inputs to determine~~ ~~Offline "looks into the future."~~

Intuition: OFFLINE ALGORITHMS LOOK INTO THE FUTURE.

STRATEGY: COMPETIVE ALGORITHMS

DEF'N. AN ON-LINE ALGORITHM A, IS c-COMPETIVE (OR "COMPETIVE") IFF $\exists d$ $\forall$ sequences of inputs

$$\text{cost}(A) \leq c \cdot \text{cost}(\text{OPTIMAL OFFLINE ALGORITHM}) + d$$

note: $c, d$ independent of sequence.

Defn: STRONGLY c-competive if no better $c$ can be found.

Example of 2 competitive algorithm: (Due to Larry Rudolph)

Ski rental. Skis cost $10 to rent, $100 to buy.

What should I do?

- ~~Always~~
- Buy skis? ⇒ could go once & spend 10× too much
- Rent always? ⇒ could spend a fortune $.

2-competitive algorithm:

Rent ~~until~~ ~~10 times~~ 10 times, then buy.

Analysis: If I ski ~~≤~~ ≤ 10 times, the optimal alg is rent, which I did.

If I ski ≥ 10 times, the optimal alg is buy.

~~I spent $(10+100) instead of $100~~

I spent $200 instead of $100, a factor of 2 off.

---

COMPETITIVE SNOOPY CACHING, by Karlin, Manasse, Rudolph, Sleator
(No online copy to be found)

Examines many kinds of caches.

~~The~~

I'll focus on DIRECT-MAPPED SNOOPY CACHE

- Direct mapped. i.e. ∃ a function $h$ mapping ~~blocks to~~ block #'s to a single slot in each cache.
- caches snoop on variables.

~~Then if processor i updates location~~ V

~~broadcasts~~ in ~~its cache, & broadcasts~~

|

On read, if not in cache must fetch a whole cache line of $p$ bytes

On write, if we communicate, we send the new ~~if DIRTY~~ value for V over the bus. This costs 1 cycle.

Definitions sidebar

$[v]$ = block # of address

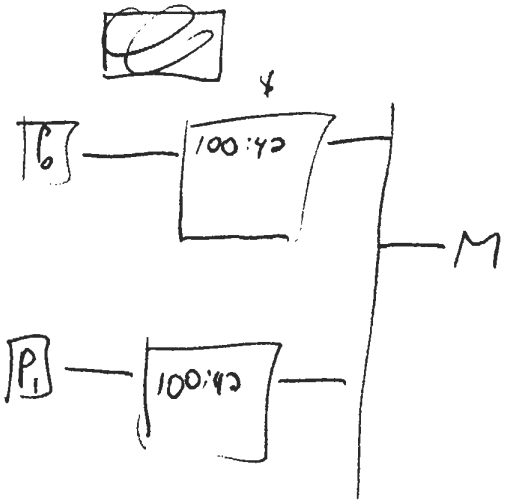
e.g. $[v] = \lfloor \frac{v}{p} \rfloor$ (V/P in C)

$h([v])$ = cache slot #

e.g. $h(B) = B$ ~~%~~ mod (Cache Size)

$h(B) = B$ % CACHESIZE in C

Only freedom: when should a processor drop a block from its cache




$P_0$ writes to 100 over and over.

$P_1$ may decide to drop the block so $P_0$ ~~does not~~

The only way for data to get into a cache is if the processor reads or writes it.

Once it is there, it must remain up-to-date.

---

Algorithm: DIRECT-MAPPED-SNOOPY-CACHING (dsc)

~~A~~
A data structure

$w_i[B]$ an array, i is processor #
B is ~~block~~ ~~slot #~~ block #

Invariant:

$w_i[B] == 0$ IFF Block B is ~~in~~ ∉ cache i.

C a bookkeeping integer

Algorithm:

TO READ v on processor i:
  let B := [v]
  if $w_i(B) == 0$ then GETBLOCK(i, B)
  else ~~$w_i(B)$ ++i; (actually can be incremented to any value up to~~
    $w_i(B) := \min(p, w_i(B)+1)$ (or ANY value in between)
  provide B from cache.

To write v from processor i:

```
let B := [v]
IF w_i(B) == 0 then GETBLOCK(i, B)
else w_i(B) := P

if ∃ j s.t. j ≠ i and w_j(B) ≠ 0 then
    // in some other cache //
    Broadcast the update on the bus.
    C++;
    w_j(B)--;
    IF (w_j(B) == 0) then drop block B from cache j.
else
    Update local copy (now B is dirty)
```

Procedure GETBLOCK(i, B):

```
if ∃ B' s.t. h(B) = h(B') and w_i(B) ≠ 0
    // B collides with B'. Drop B'. //
    IF B' is dirty then
        WRITE B' back to memory
        C += P;
    w_i(B') := 0
    Drop (i, B')
Fetch block B into cache
C += P;
w_i(B) := P
```

Two choices: increment can be by > 1
Choice of j during write

**Theorem:** Algorithm DSC is ~~strongly~~ 2-competitive.
That is, for any sequence of reads + writes, & any (online or offline) algorithm A

$$COST_{dsc}(\sigma) \leq 2\ COST_A(\sigma) + k$$

K depends only on the initial cache states, and $k = 0$ if all caches are initially empty

We use a potential function $\Phi$

$\Phi(t)$ is a function that depends on the cache states of dsc + A after processing the first $t$ steps of $\sigma$.

$$\Phi(t) = \sum_{(i,s) \in S_A} (\omega_i(B) - 2p) + \sum_{(i,s) \notin S_A} -\omega_i(B)$$

where $S_A$ is the set of pairs $(i,B)$ s.t. A has block B in cache i at step t.
note $\Phi(t) \leq 0$

We'll prove inductively

$$COST_{dsc}(t) - 2\ COST_A(t) \leq \Phi(t) - \Phi(0)$$

for $t = 0$, both sides are 0.

The inductive step is to show

$$\Delta\ COST_{dsc} - 2\ \Delta\ COST_A \leq \Delta\Phi$$

~~Strategy: given $\sigma$ construct easier sequence $\tau$~~

~~$\tau$ is constructed from $\sigma$ - s~~

~~for each request in $\sigma$~~

Strategy: Break up operations into

| | |
|---|---|
| Fetchblock$(i,\beta)$ | Block $\beta$ added to cache $i$, cost $= \rho$ |
| Drop $(i,\beta)$ | Block $\beta$ is dropped from $i$. ($\beta$ not dirty) cost $=0$ |
| WRITEBACK$(i,\beta)$ | Block $\beta$ is made clean. $\beta$ is broadcast to $\rho$ cost $=\rho$ |
| Supply $(i,v)$ | Variable $v$ is supplied to proc $i$ by its cache. block $[v]$ must be in cache. cost $=0$ |
| Update local$(i,v)$ | Variable $v$ is updated in cache $i$: $[v]$ must be unique to $i$. $(v)$ becomes dirty cost $=0$ |
| Update Global$(i,v)$ | Variable $v$ is updated in cache $i$ + broadcast. $[v]$ must be clean and in cache $i$ cost $=1$ |

~~Need~~

~~Algorithm does performs, for a~~

Any algorithm must perform those ops; for a read, end with supply$(i,v)$ for a write, end with update.

~~consider~~ $\tau$ a sequence of ~~these subops~~.

$\tau =$ subops ~~done by~~ A/ on ~~the~~ ~~first request in $\sigma$, not including~~ to supply or update subops done by ds on ~~first req~~

A: Fetch block( ); Drop( ); supply( ); ...

dsc: fetch block( ); supply( );

Idea: like these supplies + updates up so we have a new sequence

$\gamma$ = (
- A fetch block
- A drop
- B fetch block
- dsc+A supply block
- ...

} put A actions in front
} then dsc actions
} then the common actions

---

Now define a potential function: depends on cache state of dsc + A after processing the first $t$ steps of $\gamma$

$$\Phi(t) = \sum_{(i,B) \in S_A} (\omega_i(B) - 2p) + \sum_{(i,B)} -\omega_i(B)$$

- $\sum_{(i,B)}$: A has B in cache $i$ at step $t$
- $\sum_{(i,B)}$: A doesn't have B in cache $i$ at step $t$.

note: $\Phi(t) \leq 0$

Prove by induction

$$cost_{dsc}(t) - 2\, cost_A(t) \leq \Phi(t) - \Phi(0)$$

Theorem will follow with $k = -\Phi(0)$ since $\Phi(t) \leq 0$

BASE CASE: $t=0$, both sides 0

INDUCTIVE CASE:

SHOW

$$\Delta \text{cost}_{dsc} - 2\Delta \text{cost}_A \leq \Delta\phi$$

CASE ANALYSIS:

IF step $i$ is "A does fetchBlock$(i,B)$"

$\Delta C_A = p$

Must show $\Delta\phi \geq -2p$

Before this step ~~(A,B)~~ - Block $B$ not in cache $i$ in $A$.

Afterwards, it is in.

Therefore

$$\Delta\phi = w_i(B) - 2p - (-w_i(B))$$
$$= 2w_i(B) - 2p$$
$$\geq -2p.$$

Step $i$ is "A drops$(i,B)$"

$\Delta C_A = 0$

Must show $\Delta\phi \geq 0$

Before $B$ in cache $i$, after $B$ not in cache $i$.

Check is $2p - 2w_i(B) \geq 0$

"A writeBack$(i,B)$"

$\Delta C_A = p$

must show $\Delta\phi \geq -2p$

but $\Delta\phi = 0$.

"dsc Fetchblock (i,D)

$\Delta_{dsc} = \rho$

must show $\Delta\phi \geq \rho$

$w_i(B)$ changes from 0 to $\rho$.

Because we did A's first, A must have B in cache i now.

so ~~$\Delta\phi$~~ $\Delta\phi = \rho$

⋮

"Goth supply (i,v)" ~~to i~~

Cost to Goth = 0

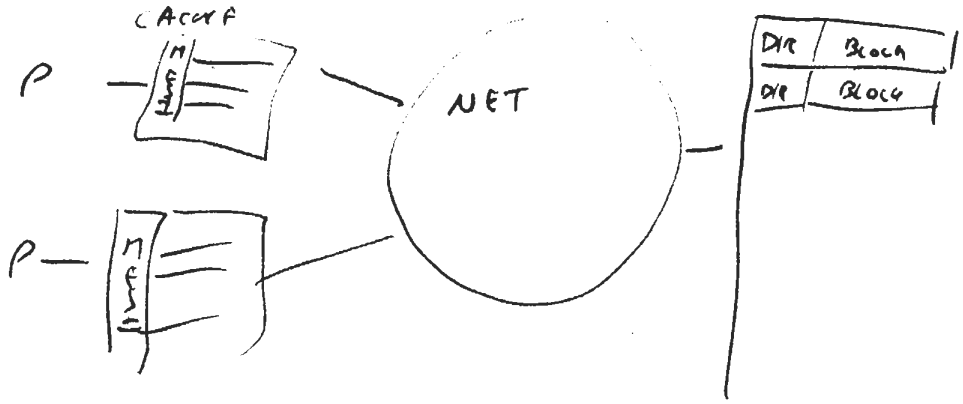
~~$\Delta w_i(B)$~~ $\Delta w_i(B) \geq 0$ since it gets it from

B in i so $\Delta\phi \geq 0$.

# Directory-Based Cache

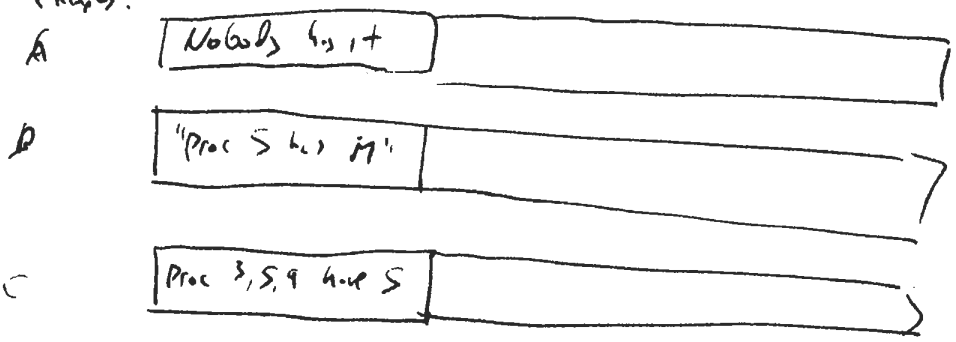
Problem with MESI: doesn't scale

Solution: Add info to every ~~cache~~ block in memory




Idea: leave info at memory to identify who needs to be updated when cache must be invalidated or updated

Examples:




To obtain E access for proc 2

case A: Fetch from mem, set M to "2 has it in M"

case B: send message to proc 5
proc 5 sends block to mem,
then case A

case C: send messages to 3, 5, 9 to invalidate. (in parallel)
~~then send~~ + send data to 2
3, 5, 9 send to 2
2 then proceeds to update.

NO ANALYSIS!