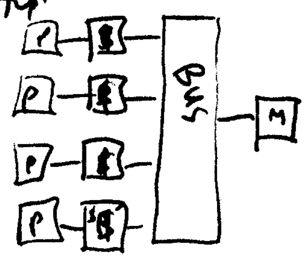
Oct 27, 2003
6.895 Bradley C. Kuszmaul

COMPETITIVE SNOOPY CACHING; ~~or~~ DIRECTORY-BASED CACHING

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



- ~~Processors~~ 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 in~~

Spatial locality is the property that if a processor reads/writes location v, it is likely to operate on $v+1$ or other locations near v.
Programs exhibit spatial locality ⇒ transmit blocks of size p.
~~Hence~~

> Definitions side bar
> 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 someday. Reduces read cost
To update a replicated block requires communication
- Either invalidate the other copies
- or update them

} which is right?

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

Worst-case behavior of exclusive-write: Here is a bad case case
P_0 writes v ~~then~~ repeat k times
$P_1 \cdots P_{n-1}$ reads v
P_0 writes v
$P_1 \cdots P_{n-1}$ reads v

read ⇒ 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 read cost, update k times

$$\lim_{K\to\infty} \frac{K\cdot(n-1)\cdot p}{K+np} = (n-1)\cdot p \qquad \text{~~for large K~~}$$

(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 co~~

Bad (worst?) case analysis for protocol:

P_0 reads

P_1 reads

repeat w times

P_0 writes

Total cost $\geq w$

Optimal is ~~exclusive write, a cost of 2p + 1~~

drop after writing for a 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,

$\exists d$ ~~cost (A) on a sequence~~

$\forall$ sequences of inputs

optimal $\geq$ c · Cost (A) + d) ≤ OPTIMAL_OFFLINE_ALGORITHM

for that sequence.

c + d constant are independent of the seq

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

Def'n: An ~~oh~~ online algorithm A, is c-competitive (or competitive) iff

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

$$\text{cost}(A) \le \cancel{\text{opt}}\ \text{cost}(\text{OPTIMAL OFFLINE ALG.})$$

Def'n: An algorithm is an online algorithm if ~~it must make its~~ its behavior depends only on the inputs

~~Def'n: An Algor~~

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

Def'n Such an algo is on-line if it can produce the ith output ~~with~~, having seen only the first i inputs, otherwise offline.

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

Intuition: OFFLINE ALGORITHMS LOOK INTO THE FUTURE.

STRATEGY: COMPETITIVE ALGORITHMS

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

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

note: c, d independent of sequence.

Def'n: STRONGLY c-competitive if no better c can be found.

Example of ~~a~~ competitive algorithm: (Due to Larry Rudolph)

~~Skiing~~ Ski rental. skis cost \$10 to rent, \$100 to buy.

What should I do?

~~Always~~

- Buy skis? ⇒ could go once & spend 10X too much
- Rent always? ⇒ could spend unbounded \$.

2-competitive algorithm:

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

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

If I ski $k > 10$ times, the optimal alg is buy.

~~I spent \$(10k+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.

~~I'll~~

I'll focus on Direct-Mapped Snoopy Cache

- Direct mapped. i.e. $\exists$ a function h mapping ~~blocks to a~~ block #'s to a single slot in each cache.

Definitions sidebar

$[v]$ = block # of address

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

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

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

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

- caches snoop on variables.

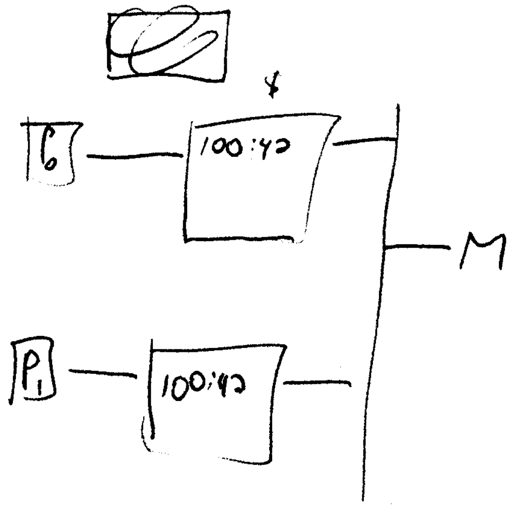
~~That is Processor i updates location~~ v

~~[illegible]~~ in it ~~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.

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



P_0 writes to 100 over + over.

P_1 may decide to drop the block so P_0 ~~can do less~~

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 $B \notin$ 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
~~$w_i(B)$++; (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 processer i:

let $B := [v]$

if ~~w~~ $w_i(B) == 0$ then GETBLOCK(i, B)

else $w_i(B) := p$

if $\exists j$ s.t. $j \neq i$ and $w_j(B) \neq 0$ then

~~/* some~~ in some other cache */

~~up~~

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 $\exists B'$ s. $h(B) = h(B')$ and ~~w~~ $w_i(B) \neq 0$

/* B collides with B'. drop B'. */

if B' is dirty then

WRITE B' back to mem

$c += p$;

~~w~~

$w_i(B') := 0$

Drop (i, B')

Fetch block B into cache

$c += p$;

~~w~~ $w_i(B) := p$

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

$$COST_{dsc}(\sigma) \le 2 \; COST_A(\sigma) + K$$

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

We use a potential function Φ

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

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

where S_A is the set of pairs (i, B) s.t. A has block B in cache i at step t.

note $\Phi(t) \le 0$

We'll prove inductively

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

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

The inductive step is to show

$$\Delta \; COST_{dsc} - 2 \; \Delta \; COST_A \le \Delta \Phi$$

~~Strategy: given σ construct cheaper sequence τ~~

~~τ is the constructed from σ~~ - s

for ~~each request in σ~~

Strategy: Break up operations into

Fetchblock(i, β)	Block β added to cache i, cost $= p$
Drop (i, β)	Block β is dropped from i. (β not dirty) cost $= 0$
WRITEBACK(i, β)	Block β is made clean. β is broadcast on bus cost $= p$
Supply (i, v)	Variable v is supplied to proc i by its cache. block $[v]$ must be in cache. cost $= 0$
Updatelocal(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 $= 1$

~~Alg~~

~~Algorithm does performs, for a~~

Any algorithm must perform these ops; for a read, end with supply(i, v)

for a write, end with update.

consider τ a sequence of ~~these subops~~.

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

subops done by dsc on ~~first request~~

~~Alg~~

A: Fetchblock( ); Drop( ); supply( ); ...

dsc: fetchblock( ); supply( );

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

$$\sigma = \left(\begin{array}{ll} A & \text{fetchblock} \\ A & \text{drop} \\ dsc & \text{fetchblock} \\ dsc + A & \text{supply block} \\ & \vdots \end{array} \right. \begin{array}{l} \left.\right\} \text{put A actions in first} \\ \left.\right\} \text{then dsc actions} \\ \left.\right\} \text{then the common action} \end{array}$$

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

~~$\Phi(t) = \sum_{(i,B) \in S_A}$~~

$$\Phi(t) = \sum_{\substack{(i,B) \\ A \text{ has } B \\ \text{in cache } i \\ \text{at step } t}} (w_i(B) - 2p) + \sum_{\substack{(i,B) \\ A \text{ doesn't} \\ \text{have } B \text{ in} \\ \text{cache } i \\ \text{at step } t.}} -w_i(B)$$

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

Prove by induction

$$\text{cost}_{dsc}(t) - 2\,\text{cost}_A(t) \le \Phi(t) - \Phi(0)$$

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

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

INDUCTIVE BASE:

SHOW

$$\Delta\, cost_{dsc} - 2\Delta\, cost_A \leq \Delta\phi$$

CASE ANALYSIS:

if step i is "A does fetchblock(i, B)"

$\Delta C_A = p$

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

Before this action ~~(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.

Change 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, b)

$\Delta_{dsc} = P$

must show $\Delta\phi \geq P$

$w_i(B)$ changes from 0 to P.

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

so ~~ϕ increases~~ $\Delta\phi = P$

⋮

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

cost to both $= 0$

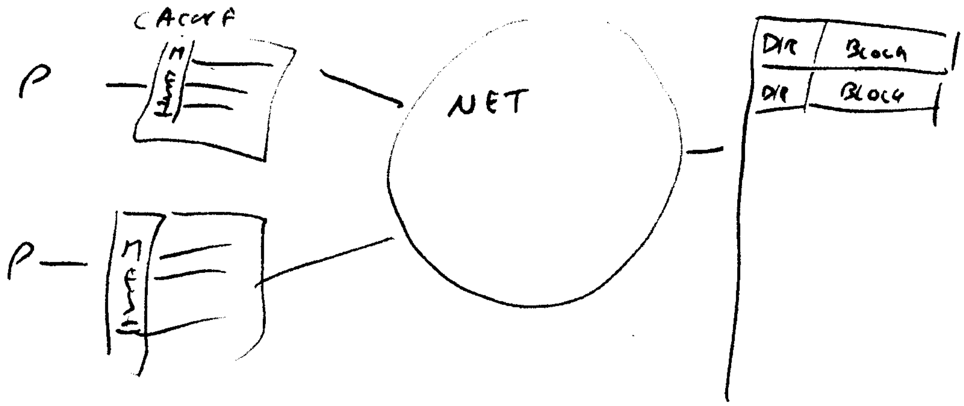
~~Δ~~ $\Delta w_i(B) \geq 0$ since it gets it top

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 update

Examples:

A | Nobody has it

B | "Proc 5 has M"

C | Proc 3,5,9 have S

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 back to mem,
then case A

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

$\boxed{\text{NO ANALYSIS!}}$