6.895 Theory of Parallel Systems	Lecture 14

Competitive Snoopy Caching

Lecturer: Bradley C. Kuszmaul *Scribe: Advait D. Karande and Sriram Saroop*

Lecture Summary

1. *Setup for snoopy caching algorithms*
   This section briefly describes the arrangement of the processors, caches, memory and their interaction by means of the bus.
2. *Two strategies to ensure correctness while updating replicated blocks*
   This section introduces the two commonly used strategies, pack rat and exclusive write, which are employed to ensure correctness while writing a replicated block. We also compare the two algorithms, and briefly introduce the algorithm suggested by Goodman.
3. *Introduction to Competitive Algorithms*
   We define the terms - online algorithms and c-competitive algorithms. This sections also illustrates competitive algorithms by means of an example (due to Larry Rudolph).
4. *Competitive Snoopy Caching*
   In this section, we introduce competitive algorithms for snoopy cache multiprocessor systems and provide one such algorithm for direct-mapped caches.

1 Setup for snoopy caching algorithms

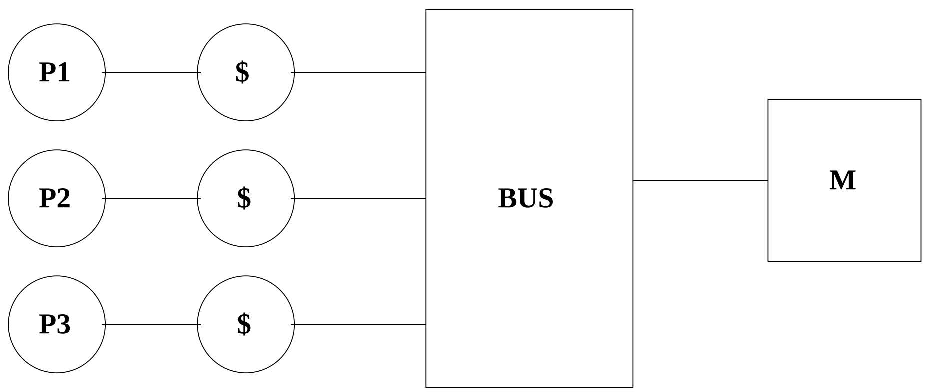

Figure 1: Setup for competitive snoopy caching

The competitive snoopy caching algorithm uses a MESI like protocol. The set up of the system is illustrated in Figure 1. We have processors each of which has a cache. The caches and the memory are interconnected through a bus. We observe the following characteristics in the system:

- Caches and Memory can see all the messages in the bus.
- Only one entity, either cache($) or memory(M) can send a message at a time.
- Processors(P) talk to the caches($) which in turn talk to the bus.

1.1 Why do caches work ?

Localities of Reference:

- *Temporal Locality* : It is the property that if a processor refers(reads/writes) to a memory location once, it is likely to refer to the same memory location again in the near future.
- *Spatial Locality* : It is the property that if a processor accesses(reads/writes) a location v, it likely to access locations that are near v , like v-1, v+1, v-2, v+2 etc.

It has been observed that programs do exhibit temporal locality and spatial locality. Since they exhibit spatial locality, caches are organized as blocks of say size p. Hence, whenever a processor requests for a single memory location, it receives the memory location and p-1 neighboring locations through the bus.

Let us compare the cost in terms of the number of cycles for different values of p. We assume that it takes one cycle for every request to be sent and one cycle for every byte of data that is got back. We consider two cases wherein 1 byte and 128 bytes are requested respectively.

- *1 byte requested*: Let us first consider a request for 1 byte. If p=1, it takes an unit cycle to send the request and an unit cycle for the memory location(1 byte) to come back. For larger p (p>1), it takes an unit cycle to send the request and p cycles for the memory location(1 byte) and its p-1 neighbors to come back. Hence, it takes a total of p+1 cycles.
- *128 bytes requested*: We proceed to consider another case wherein 128 bytes of data are requested. In the case when p=1, the processor sends a request for each byte, which takes an unit cycle each time, and receives 1 byte for every request, which again takes an unit cycle each time. Hence, for 128 bytes, it takes 128 request unit cycles and 128 bytes of data response unit cycles, amounting to a total of 256 cycles. This corresponds to an *uncached read.* In the case when p=128, it takes 1 request unit cycle, because of which a block of p=128 bytes is returned. Hence this takes 129 cycles. However, since the rest of the 127 bytes are already cached, no more cycles are needed. This corresponds to a *cached read.*

Therefore, for a reasonably large cache block size, we notice an improvement by a factor of 2 in the number of cycles needed.

2 Two strategies to ensure correctness while updating replicated blocks

2.1 An introduction to the Pack Rat and Exclusive Write algorithms

Blocks are sometimes stored at more than one processor, since we use caching to store the blocks at each processor's local cache. Hence whenever such a replicated block is being updated, we need to ensure that we communicate this change to other processors that have this replicated block. There are two commonly used strategies that address this problem.

- *Pack Rat* :

  If the processor knows that someone else has a copy of the block, it goes ahead to write onto the bus, and also *updates everybody* who has a copy. The processor incurs an unit cycle for writing the memory location, and since there is no reply, that is the only cost it incurs. Thus, a pack rat operation requires just 1 cycle.

- *Exclusive Write* :

  The processor which is writing to a memory location invalidates all processors that hold a copy of the block to which the memory location belongs to. This means that the exclusive write operation incurs only 1 unit cycle, as the only cost incurred is the communication cost for invalidating the cache-blocks of all the other processors.

2.2 Good cases for pack rat and exclusive write

Let us consider a couple of good cases for each algorithm.

- *Good case for exclusive-write* : Suppose a given processor wants to perform k consecutive writes, then exclusive-write would incur a lesser cost (just 1 cycle) since all other processors have been invalidated for the first write itself. Whereas pack-rat would have to update all other processor-caches for every write (incurring a cost of k cycles) which evidently is more expensive.
- *Good case for pack-rat* : Suppose that another processor wants to read a memory location after a processor has performed a write. In this case, exclusive-write would incur p+1 cycles for the read, as this processor's block in the cache would have been invalidated by the previous processor's write on the replicated block. Thus, it needs to read the entire cache block of p bytes. Whereas pack-rat would incur 0 cycles, as the previous processor would have updated its cache block.

2.3 A bad case for exclusive write

repeat k times
  P_0 writes to location v
  P_1,...,P_{n-1} read location v

The total number of read cycles needed for this task in *exclusive-write* is *k(n-1)p*. This is because the n-1 processors have to read the cache block that contains location v in every iteration as that cache block is invalidated in each iteration.

However, the total read cost incurred by *pack-rat* is only *k+np*, since the n processors have to load their caches only initially (cost = np), and the updates cost k cycles over all iterations. There are no further read costs incurred since the cache blocks of the processors are updated each time P_0 writes to location v. For large k, $np \ll k$. Considering the ratio of the performance of exclusive-write to that of pack-rat as k $\rightarrow \infty$, we get :

$$\lim_{k\to\infty} \frac{k(n-1)p}{k+np} = \lim_{k\to\infty} \frac{(n-1)p}{1+(np/k)} = (n-1)p \tag{1}$$

Hence in this case, exclusive-write is worse than pack-rat by a factor of (n-1)p.

2.4 A bad case for pack rat

P_0 reads location v
P_1 reads location v
repeat w times
  P_0 writes to location v

The first two reads by P_0 and P_1 each incur p+1 cycles in both exclusive-write and in pack-rat as their cache blocks have to be loaded initially. The w writes by P_0 incurs a cost of w cycles in pack-rat because in every iteration P_0 has to update all other replicated cache blocks which contain v. Whereas it incurs only one cycle in exclusive-write since the invalidation takes up only one cycle and does not have be repeated for each iteration.

Hence, the total cost for *exclusive-write* is (p+1)+(p+1)+1= 2p+3.

The total cost for *pack-rat* is (p+1)+(p+1)+w= w+2p+2.

Hence for large w,

$$\lim_{w \to \infty} \frac{w + 2p + 2}{2p + 3} \to \infty \tag{2}$$

This is an unbounded limit on the ratio of the performances.

2.5 Goodman's algorithm

Goodman's algorithm [1] alternates between the two strategies.

```
On first write, update everything else
On second write, invalidate everything else
```

It sounds fine for the first bad case that affected exclusive-write in Section 2.3 because Goodman's algorithm acts like pack-rat for the first write. But it is still unclear if this is an optimal strategy.

3 Introduction to Competitive Algorithms

Consider an algorithm that takes an input sequence, σ and produces an output sequence τ.

Definition 1 (Online algorithm) *An algorithm is online if it can produce the i^{th} output, τ_i having seen only the first i inputs: $\sigma_1, \sigma_2, ..., \sigma_i$. Otherwise, the algorithm is considered to be offline.*

The *intuition* is that online algorithms cannot look into the future.

Definition 2 (Competitive algorithm) *An online algorithm, A is c-competitive (or competitive) if and only if $\exists$ d $\forall$ sequences σ, such that: cost(A) $\leq$ c.cost(optimal offline algorithm)+d , where c and d are independent of the sequence, σ.*

Definition 3 (Strongly c-competitive algorithm) *A c-competitive algorithm is strongly c-competitive for a given c such that no better c can be found to satisfy the definition for c-competitiveness.*

3.1 An example of a competitive algorithm (Due to Larry Rudolph)

Let us consider the ski-rental problem. Suppose that skis cost \$10 to rent and \$100 to buy. What would the optimal strategy be?

We can construct a 2-competitive algorithm in the following manner. The strategy would be to rent until we spend the purchase price and then proceed to buy the ski. This would mean that we would rent the ski 10 times and only after that do we proceed to buy the ski. Obviously, the best strategy would be to rent the ski if we plan to go skiing less than 10 times. The algorithm that we constructed follows this as well. In the case when we plan to go for skiing more than 10 times, we would be spending 200\$, whereas the optimal offline strategy had we known the future would have been to purchase the ski beforehand for 100\$. Therefore, in the worst case our algorithm is at most a factor of 2 off the optimal offline strategy. Hence, our algorithm is 2-competitive.

4 Competitive Snoopy Caching

In a snoopy cache multiprocessor system, each processor has a cache memory in which it can store blocks of data. (Refer to Figure 1) The *snoopy* cache of each processor actively monitors the activity on the bus and in its processor and dynamically determines which blocks of data to keep and which ones to discard. This strategy is called *block retention strategy.*

Karlin et al. [2] introduced competitive snoopy caching algorithms for various cache architectures. A cache architecture is defined by several parameters such as, whether the cache is direct-mapped or associative, whether it snoops on single variables or on blocks of data and so on. We will focus on the competitive algorithm for *direct-mapped* caches.

The characteristics of direct-mapped snoopy cache model are as follows.

- Direct-mapped: Caches are characterized by a hash function $h(B)$, which gives a unique cache-line in which block B would reside. An example of a simple hash function would be $h(B) = b0 \textit{ modulo } c$, where b0 is the block number of B and c is the cache size in terms of the number of cache lines. In other words, each memory block maps to a unique cache-line in the cache. This is in contrast to *k-way set associative* or *fully associative* caches, wherein a memory location may be mapped to multiple cache-lines.
- Each read or write request of a processor for a memory variable is satisfied only after its cache has the block containing the variable.
- Caches snoop only on variables and not on the blocks.

We now provide a setup for analyzing the Direct-Mapped Snoopy Caching algorithm (**DSC**).

We let p be the size of each cache-line in bytes, $[v]$ denote the the block containing variable (i.e. memory location) v. Usually, $[v]$ would be calculated as $\lfloor \frac{v}{p} \rfloor$.

Processors perform a sequence of memory transactions, each of which can be of one of the two types:

- $\text{READ}_i(v)$: Processor i requests the value of variable v.
- $\text{WRITE}_i(v)$: Processor i requests that a new value be written to variable v.

In response to a request by processor i, its cache performs one or more actions from the following domain. Associated with each action is the cost, which reflects the time spent or bus cycles used in performing the actions.

$Fetchblock(i, B)$:	Block B is added into cache i. Since it requires fetching p bytes over the bus, it costs p.
$Writeback(i, B)$:	Block B is sent over the bus to the main memory. Again, this requires sending p bytes over the bus and costs p.
$Drop(i, B)$:	The cache-line containing block B is made empty. The action costs 0.
$Supply(i, v)$:	When block $[v]$ is present in cache i, it supplies variable v to processor i. The action costs 0.
$Updatelocal(i, v)$:	When block $[v]$ in cache i is the only cached copy of the block, variable v is updated in the cache. The action costs 0.
$Updateglobal(i, v)$:	When block $[v]$ is in cache i, variable v is updated in the cache and is broadcast on the bus. This action requires sending unit data over the bus and costs 1.
$Finishwrite(i)$:	It occurs at the end of a write request. Its cost is 0.

4.1 Direct-mapped snoopy caching algorithm (dsc)

In the direct-mapped snoopy cache model described above, the only freedom a cache has has is in deciding when it should drop a block. It makes this choice using the *dsc* algorithm.

The main data structure used in this algorithm is an array of counts $W_i[B]$, which is used to decide when to drop block B from cache i. The values in $W_i[B]$ range from 0 to p. The algorithm is presented below.

To read variable v on processor i [$\text{READ}_i(v)$]:

B := [v]
if $W_i[B] == 0$

then GETBLOCK(i, B);
else $W_i[B]$:= min(p, $W_i[B] + 1$);
Supply(i, v);

To write variable v from processor i [WRITE$_i(v)$]:

B := [v];
if $W_i[B] == 0$
then $GETBLOCK(i, B)$;
else $W_i[B]{:=}p$;
if $\exists j \neq i$ such that $W_j[B] \neq 0$
then $Updateglobal(i, v)$;
$W_j[B] := W_j[B] - 1$;
if $W_j[B] == 0$
then Drop(j, B);
else $Updatelocal(i, v)$;

Procedure GETBLOCK(i, B)
if $\exists B'$ such that $h_i(B') = h_i(B) \wedge W_i[B'] \neq 0$
then $\triangleright$ B collides with B', so drop B'
if $W_i[B'] == 0$
then $Writeback(i, B)$;
$W_i[B'] := 0$;
$Drop(i, B')$;
$Fetchblock(i, B)$;
$W_i[B] := p$;
end GETBLOCK;

Let's look at some of the characteristics of this algorithm. It maintains two invariants.

- $W_i[B] = 0$ if and only if block B is not in cache i.
- If i is the last processor to modify a dirty block B , then $W_i[B] = p$.

A block is fetched into a cache only when the corresponding processor performs a read or write to a variable in the block and the block is not in the cache. After fetching block B into cache i, its count for that processor ($W_i[B]$) is set to p. It is dropped from cache i, when the count $W_i[B]$ reaches zero. In case of a read operation, if the required block is found in the cache, then its count is incremented by 1 up to a maximum of p. Similarly, on a cache hit during a write operation, the count for the corresponding block $W_i[B]$ is set to p. Thus, a cache hit on read or write operation increases the likelihood that the corresponding block will remain in that cache. Moreover, during a write operation, if the required block B is replicated, then the writing processor i needs to broadcast the update on the bus. Here each other cache containing the block is guilty of causing this bus cycle. Hence, a write to a replicated block decrements the count in one of the other caches (say j) sharing the block. Thus a cache hit on write operation to a shared block increases the likelihood that the block will be dropped from other caches holding the block.

Now we state the following theorem about the competitiveness of this algorithm.

Theorem 4 *Algorithm dsc is strongly 2-competitive, that is, for any sequence σ and any online or offline algorithm A,*

$$C_{dsc}(\sigma) \leq 2 * C_A(\sigma) + k$$

The constant k depends only on the initial cache states of dsc and A, and is zero if all caches are initially empty.

A formal proof of this theorem will be presented in the next lecture. Intuitively, this algorithm is similar to the ski-rental algorithm mentioned in Section 3.1, which is also a 2-competitive algorithm. We maintain a (replicated) block in a cache until the overhead of maintaining it becomes equal to the cost involved in fetching it from the memory.

References

[1] J. R. Goodman, "Using cache-memory to reduce processor-memory traffic," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 124–131, 1983.

[2] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive snoopy caching," in *Proceedings of the 27th Annual Symposium on the Foundations of Computer Science*, 1986.