Algorithmica (1988) 3: 79–119

Algorithmica

Competitive Snoopy Caching[1]

Anna R. Karlin,[2] Mark S. Manasse,[3] Larry Rudolph,[4] and Daniel D. Sleator[5]

Abstract. In a snoopy cache multiprocessor system, each processor has a cache in which it stores blocks of data. Each cache is connected to a bus used to communicate with the other caches and with main memory. Each cache monitors the activity on the bus and in its own processor and decides which blocks of data to keep and which to discard. For several of the proposed architectures for snoopy caching systems, we present new on-line algorithms to be used by the caches to decide which blocks to retain and which to drop in order to minimize communication over the bus. We prove that, for any sequence of operations, our algorithms' communication costs are within a constant factor of the minimum required for that sequence; for some of our algorithms we prove that no on-line algorithm has this property with a smaller constant.

Key Words. Shared-bus multiprocessors, Amortized analysis, Potential functions, Page replacement, Shared memory, Cache coherence.

1. Introduction. *Snoopy caching* is a promising new technique for enhancing the performance of bus-based multiprocessor systems [G], [F], [PP], [KEW], [RS1], [AB], [VH]. In these designs each processor is connected to its own snoopy cache. All of a processor's memory requests are serviced by its cache. These caches and a main memory are connected by a common bus. Since the bus can service only one request at a time, inefficient use of the bus may cause a processor to idle while its cache is waiting for the bus. Each snoopy cache monitors the activity on the bus and in its own processor, and can dynamically choose which variables to keep and which to drop in order to reduce bus traffic.

In practice, programs for multiprocessors exhibit locality of reference just as sequential programs do. Since fetching a variable over the bus requires one bus cycle to send the address and one bus cycle to receive the value, most designs reduce the overhead of address cycles by clustering variables into blocks. A block fetch of size b takes only $b+1$ bus cycles, thus saving nearly a factor of two if b is reasonably large and the expectation of locality is justified.

To reduce the cost of reading further, a block may reside in more than one cache at a time. Requests by a processor to modify a memory location that is

[1] A preliminary and condensed version of this paper appeared in the *Proceedings of the 27th Annual Symposium on the Foundations of Computer Science*, IEEE, 1986.

[2] Computer Science Department, Stanford University, Stanford, California, USA. This author received support from an IBM doctoral fellowship, and did part of this work while a research student associate at IBM Almaden Research Center. Current Address: Computer Science Department, Princeton University, Princeton, NJ 08544, USA.

[3] DEC Systems Research Center, Palo Alto, CA 94301, USA.

[4] Computer Science Department, Hebrew University, Jerusalem, Israel.

[5] Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, USA.

Received October 28, 1986; revised May 30, 1987. Communicated by Jeffrey Scott Vitter.

stored in more than one cache require bus communication. This is because each copy of the block must either be marked invalid, or updated to reflect the new value. (If a particular cache decided not to take either of these actions, then the possibility exists that this cache would later return an incorrect value to its processor.) Thus, a bus cycle is required to broadcast information to all the relevant caches.

A snoopy cache system must use some *block retention* strategy to decide for each cache which blocks to keep and which to discard. In choosing such a strategy we face a tradeoff: if a block is replicated, then the system must pay the cost of updating all copies following a write to that block; if a block is dropped from a processor's cache, then the system must pay for the processor's next request to read a variable in that block.

Most snoopy cache designs use either *exclusive-write* or *pack-rat* as a block retention strategy. In the exclusive-write strategy a write to a location in a block causes all other copies of the block to be invalidated. In the pack-rat strategy a block is only dropped from a cache as a result of a conflict with another block.

Exclusive-write can be a bad strategy. Suppose a processor writes to a specific location and then each of n processors reads that location. This pattern is repeated k times. Let p be the number of bus cycles needed to read a block. Since $n-1$ processors have to re-read the block containing the location on each iteration, the cost per iteration is at least $(n-1)p$, for a total cost of at least $k(n-1)p$. The optimal strategy for this sequence is to keep the location shared among all processors with each write updating the contents of all the other caches for a total cost of at most $k+np$. Hence, in the limit, the number of bus cycles used by exclusive-write exceeds the optimal by a factor of $(n-1)p$.

Pack-rat can also be a bad strategy. Consider the situation in which two processors read a location, and then one of the processors writes the location w times. Since pack-rat keeps the location shared, it must use a bus cycle for each of the writes, for a total cost of at least w. The optimal strategy for this sequence is for all but the writing processor to drop the block immediately after reading it, incurring a total cost of at most $2p$. Hence, in the limit as w increases, pack-rat uses unboundedly more cycles than are needed.

Is there another strategy that performs well in both of these situations? In this paper we answer this question affirmatively. In fact, we prove something far stronger: we construct strategies that perform nearly optimally for any sequence of operations.

An on-line algorithm is *c-competitive* (or simply *competitive*) if its cumulative cost on any sequence (plus a constant) is within a factor of c of the cost of the optimal off-line algorithm on the same sequence. The constant c and the additive constant are independent of the sequence of requests. If c is the smallest possible such constant for any on-line algorithm, then the algorithm is said to be *strongly c-competitive* (or *strongly competitive*). Sleator and Tarjan [ST] showed that move-to-front is a competitive algorithm for maintaining a linear search list. In this paper we extend their techniques to analyze new competitive block retention algorithms for several snoopy cache models.

We believe that competitive analysis is more useful than other theoretical methods for analyzing snoopy caching strategies. There are sequences on which no snoopy caching strategy does well (which rarely occur in real systems). It is therefore not sensible to use the worst-case performance of two schemes as an indicator of which is better. To do average-case analysis, a statistical model of the sequence of requests is required. It is extremely difficult to devise a realistic model, since the pattern of accesses changes dynamically with time and with different applications. Without prior knowledge of the structure of the sequence of operations, a scheme that is competitive is more attractive than one that is not. A competitive scheme can be only slightly worse than any other scheme, and may be very much better.

The principle used to devise these schemes is roughly the following: the change from a state A to a state A' is made when the cost of making the change equals the extra cost incurred due to being in state A instead of A'. Like the strategies of [RS2], our algorithms keep counters. We use these counters to make these cost estimates and to decide when to change the arrangement of blocks.

We use competitive analysis to compare block retention schemes for particular architectures. Our analyses shed little light on the question of which architecture to choose, or how to choose the parameters of a given architecture.

Section 2 of this paper defines our terminology, notation, and models. Section 3 presents lower bounds. Section 4 describes and analyzes a strongly competitive algorithm for direct-mapped snoopy caching. The approach is described in detail, and is used in subsequent proofs. Section 5 re-examines paging algorithms and is concerned with dynamically mapped (associative) caches. The algorithm presented in this section is not strongly competitive, however, its performance is within a constant factor of the optimum algorithm running on a system with less memory. Sections 6, 7, and 8 present and analyze algorithms for different generalizations of direct-mapped snoopy caching. Section 9 considers configurations with multiple busses. Finally, the appendices address various implementation issues and present those proofs of lesser technical interest.

2. Definitions, Notation, and Models. In a snoopy caching system there is a single address space used by all of the processors. A location in this space is called a *variable*. The operation of the caching system is transparent to the processors; from their point of view the system is simply a memory which allows them to read and write variables.

The caching system partitions the memory space into *blocks*, groups of variables of uniform size. We let p denote the block size, and $[v]$ denote the block containing the variable v.

Each cache may contain a collection of blocks. The number of blocks in a cache is bounded by the cache size, which may be less than the number of blocks in the address space. The collection of blocks in a cache may change with time. Although main memory is large enough to store all of the blocks of the address space, not all the blocks are stored there at all times.

The slots that may contain blocks in a cache are called *cache lines*. A *direct-mapped* cache uses a hash function $h_i(B)$ to determine the unique cache line in

which block B will reside. If $h_i(B) = h_i(B')$ then cache i can contain at most one of the blocks B and B' at any time. (Some authors describe this situation by saying that the set size is 1.) In an *associative cache* any line can store any block.

A cache line is said to be *empty* if it does not contain a block. A *cache collision* occurs when a block is needed in a cache line which currently contains another block.

A block may reside in any subset of the caches, and may or may not reside in main memory. If a block is in none of the caches then it must be in main memory. If main memory does not have an up-to-date copy of the block then it is *dirty*, otherwise it is *clean*. A block that is contained in more than one cache is *replicated*. A block residing in only one cache is *unique*.

Main memory plays a more passive role than a cache. A request for data is only answered by main memory if no cache responds to the request. This means that main memory does not have to know which blocks are clean or dirty. Main memory is said to *snoop* if, whenever a new value of a variable or block is broadcast on the bus, the memory updates it own copy of the variable or block. If main memory snoops then any replicated block also resides in main memory, since a change to any variable in a replicated block is broadcast over the bus. A snooping main memory sees this broadcast and updates its copy of the variable. Therefore any replicated block is clean. If main memory does not snoop then the moment a variable is changed by a cache containing the block, the block becomes dirty.

All of the caches discussed in this paper *snoop on variables*. That is, if a new value of variable v is broadcast on the bus then any cache containing block $[v]$ updates its copy of the variable. In some models the caches can *snoop on blocks*. This means that when a block is broadcast over the bus, any cache that wants the block (and has space for it) can grab the block and put it into its cache.

A processor interacts with the caching system by making a sequence of *requests* of two types:

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

In response to a request by processor i, the caches execute one or more of the following *actions*. Each action has an associated cost, which is the number of bus cycles it requires.

Fetchblock(i, B): Block B is added to cache i. The contents of block B are copied to an empty line. This action costs p.
Writeback(i, B): Block B is made clean. The contents of block B in cache i are broadcast on the bus. Main memory copies the block. This action costs p.
Drop(i, B): The line containing block B in cache i is made empty. Block B is present in cache i, and is either clean or replicated. This action costs 0.
Supply(i, v): Variable v is supplied to processor i by cache i. Block $[v]$ is present in cache i. This action costs 0.

Supplythrough(i, v): Variable v is supplied to processor i. Block $[v]$ is not present in cache i. The value is retrieved over the bus. This action costs 1.

Updatelocal(i, v): Variable v is updated in cache i. This action is only allowed if block $[v]$ is unique to cache i. After this action block $[v]$ is dirty. The action costs 0.

Updateglobal(i, v): Variable v is updated in cache i and is broadcast on the bus. This action is only taken only if block $[v]$ is in cache i. If the block is initially clean, and main memory snoops, then the block remains clean, otherwise the block becomes dirty. This action costs 1.

Updatethrough(i, v): This action is the same as *Updateglobal*(i, v) except that it is taken when variable $[v]$ is not in cache i. The new value of v is broadcast on the bus and is updated in all caches containing it (and main memory if it snoops). The cost of this action is 1.

A block-retention algorithm responds to a READ request with a sequence of actions terminated by either a *Supply* or *Supplythrough* action. Similarly, the response to a WRITE request is a sequence of actions ending with an *Updatelocal*, *Updateglobal*, or *Updatethrough*. We will use the symbol *Update* to denote any of these three types of update operations.

We are now ready to extract from this framework several specific models. These models are distinguished by:

(i) Whether the caches are direct-mapped or associative.
(ii) Whether *Supplythrough* and *Updatethrough* are allowed. If not, $\text{READ}_i(v)$ and $\text{WRITE}_i(v)$ requests can only be satisfied after cache i contains block $[v]$.
(iii) Whether or not main memory snoops. Sometimes our algorithm and its analysis are independent of this parameter. We only specify it in cases where it matters.
(iv) Whether the caches snoop on variables only, snoop on blocks, or snoop on blocks in a limited fashion.

We have obtained competitive algorithms for each of the following models.

Direct-Mapped Snoopy Caching

- Direct-mapped.
- No *Updatethrough* or *Supplythrough* actions.
- Caches snoop only on variables. (Thus, snooping only occurs during *Updateglobal*(i, v) actions.)

The only freedom a cache has in this model is deciding when it should drop a block. Voluntarily dropping a block is advantageous only if that block becomes unique to a single cache. The only way a block can enter cache i is in response

to a $\text{READ}_i(v)$ or $\text{WRITE}_i(v)$ request. Our algorithm for this model is strongly 2-competitive.

Associative Snoopy Caching

- Associative mapping.
- No *Updatethrough* or *Supplythrough* actions.
- Caches snoop only on variables (*Updateglobal* actions).

A block retention strategy for this model has a great deal more freedom than in direct-mapped snoopy caching. If a block is to be read into a full cache then the strategy may select any block to drop in order to make room for the new one. The problem of deciding which block to drop is closely related to the demand paging problem.

Block Snoopy Caching

- Direct-mapped. We assume that in each cache no two blocks map to the same line, i.e., each cache is as large as main memory.
- No *Updatethrough* or *Supplythrough* actions.
- Caches snoop on variables and blocks. That is, the caches snoop on *Fetchblock*, *Writeback*, and *Updateglobal* actions.

Because of the unlimited cache size and the ability to snoop on block transfers, any block that is not unique can be replicated in all caches. As in direct-mapped snoopy caching, the freedom of a block retention scheme is limited to deciding when to drop a block. Our algorithm for this model is strongly 2-competitive.

Limited Block Snoopy Caching

- Direct-mapped (bounded memory).
- No *Updatethrough* or *Supplythrough* actions.
- Caches snoop on variables and blocks. Snooping on blocks is limited to those that were the last to occupy the line. That is, cache i can acquire block B by snooping only if the last action of cache i for any block B' with $h_i(B) = h_i(B')$ was $Drop(i, B)$.
- Main memory snoops.

An example of a situation in which block snooping is advantageous is when a block that is replicated in several caches is subjected to a sequence of writes by a single processor. After all the caches drop this block but one, they can all recover it at the cost of a single *Writeback* or *Fetchblock*. The algorithm we obtain for this model is strongly 2-competitive. The restriction on which blocks may be snooped is necessary to obtain a competitive algorithm in this model. Relaxing this restriction gives an off-line algorithm too much power: an on-line algorithm cannot know which of two blocks that hash to the same line will be used next. Unlike all the previous models, *Updateglobal* actions are used even when a block is not replicated. This keeps the block clean when it would otherwise become dirty.

General Snoopy Caching

- Direct-mapped.
- *Updatethrough* and *Supplythrough* allowed. The block $[v]$ need *not* be placed in cache i to satisfy a $\text{READ}_i(v)$ or $\text{WRITE}_i(v)$ request.
- Caches snoop only on variables (*Updateglobal* and *Updatethrough* actions).
- Main memory snoops.

In this model a cache can voluntarily drop a block (i.e., not in response to a collision), and can retain a block that collides with the current access request. We obtain a strongly 3-competitive algorithm for this model if *Writeback* costs are ignored, and a 4-competitive algorithm if they are considered.

The model features can be combined in other ways to create other snoopy caching models. Most of these models are either unrealistic, or a straightforward application of the techniques of this paper will yield a competitive algorithm for them. For example, associativity is not an issue in block snoopy caching since the memory is infinite. For models of general snoopy caching or limited block snoopy caching that are associative, competitive algorithms can be obtained by using the techniques of Section 5. We also believe that a competitive algorithm can be obtained for general snoopy caching without main memory snooping. We do not know how to obtain a competitive algorithm for limited block snoopy caching without a snooping main memory. However, the model we analyze is natural since a snooping main memory can be viewed as an extra large cache.

As stated above, a block-retention algorithm takes a sequence σ of $\text{READ}_i(v)$ and $\text{WRITE}_i(v)$ requests and generates a sequence of actions in response which satisfy the constraints. An algorithm is *on-line* if it generates its response to a request up to the completing *Supply*, *Supplythrough*, or *Update*, before examining any future requests. An on-line algorithm exhibits *local-control* if, after having examined a request by processor i, no actions for caches other than i are taken until either the request is completed or cache i takes an action with nonzero cost. In the remainder of the paper on-line means on-line local-control.

Let A be any on-line algorithm which takes a sequence σ of $\text{READ}_i(v)$ and $\text{WRITE}_i(v)$ requests, and generates a sequence of actions satisfying the constraint of the model. Let $C_A(\sigma)$ denote the maximum cost of any sequence of actions generated by A on input σ. Let *opt* be any *off-line* algorithm that examines the entire sequence of requests in advance and generates a sequence of actions satisfying the constraints above with minimum cost. Then $C_{opt}(\sigma)$ is the minimum over all algorithms A of $C_A(\sigma)$, since for any sequence σ there is an algorithm that guesses that the input will be σ and performs optimally.

We will sometimes consider separately the costs attributable to *Fetchblock*, *Writeback* and *Update* actions. These costs are denoted FBC, WBC and UC, respectively.

It is important to realize that our algorithms are abstractions presented in a manner that simplifies the analysis. Although in this form they appear to require centralized control, Appendix B presents techniques for implementing them in

a distributed fashion. We assume that requests are sequential, even though this is not true in a parallel machine, since the bus serializes all communication.

3. Lower Bounds

THEOREM 3.1. *Let A be any on-line block-retention algorithm in a model without Supplythrough and Updatethrough. If there are at least two caches then there is an infinite sequence of requests σ such that $C_A(\sigma(n)) \geq n$, and*

$$C_A(\sigma(n)) \geq 2 \cdot C_{opt}(\sigma(n))$$

for infinitely many values of n, where $\sigma(n)$ denotes the first n requests of σ. Moreover, for all n,

$$C_A(\sigma(n)) + p \geq 2 \cdot C_{opt}(\sigma(n)).$$

PROOF. Consider two caches, 1 and 2, and a block B initially replicated in both caches. The sequence σ will consist of just two types of operations: $\text{WRITE}_1(B)$ and $\text{READ}_2(B)$.[6] The first request in σ is a $\text{READ}_2(B)$.

We generate the rest of σ by applying the following rule: if A has block B in cache 2, then we issue a $\text{WRITE}_1(B)$ request. Otherwise we issue a $\text{READ}_2(B)$ request.

The cost incurred by Algorithm A for this sequence is at least one for each $\text{WRITE}_1(B)$ request and p for each $\text{READ}_2(B)$ request after the first. (Note that since all control is local, A cannot first *Drop*$(2, B)$ at 0 cost then *Supply* at 0 cost, when processing a $\text{WRITE}_1(B)$ request; processor 2 does not "know" that processor 1 has just executed a write until either an update or an invalidation request is sent over the bus.)

We will now describe an off-line algorithm H and show that its cost on $\sigma(n)$ is at most half that of A. Algorithm H uses a look-ahead of only p, and is independent of A. After each read, H chooses to make B unique to cache 1 if and only if the read is followed by at least p consecutive writes.

We prove by induction that after each read $2 \cdot C_H \leq C_A$. It is true after the first read since $C_H = C_A = 0$. If there are k writes between one read and the next, then the cost incurred by A during that interval is $k + p$, and that incurred by H is $\min(k, p)$. Since $2 \cdot \min(k, p) \leq k + p$, the result follows by induction. □

The proof of this theorem shows that if an on-line algorithm does not spend nearly equal amounts reading and writing, then an off-line algorithm can beat it by more than a factor of 2. In devising the algorithms in this paper we were guided by the constraint that the cost of reading and writing must be balanced.

Another question to consider is whether there is a "best" on-line algorithm. Here we show in a strong sense that there is no such thing.

[6] Here we abuse notation somewhat and use $\text{WRITE}_i(B)$ to denote a request $\text{WRITE}_i(v)$, where $B = [v]$. $\text{READ}_i(B)$ is defined similarly.

THEOREM 3.2. *Let A be any on-line block-retention algorithm in a model without Supplythrough and Updatethrough and with at least two caches. For any on-line algorithm A there is another on-line algorithm G such that for all sequences of requests* τ,

$$C_G(\tau) \leq C_A(\tau) + 2p,$$

and for every N there exists a sequence σ *such that* $C_A(\sigma) \geq N$ *and*

$$2 \cdot C_G(\sigma) \leq C_A(\sigma).$$

PROOF. Given an algorithm A, construct σ and H as in Theorem 3.1. On any input τ Algorithm G emulates H while τ is a prefix of σ. As soon as τ deviates from σ, G sets its state to match A's and emulates A thereafter.

During the prefix of τ that is a prefix of σ, G performs like H, and incurs a cost at most p more than A. At the point at which τ and σ first differ, G must change into A's current state, at a cost of at most p. During the remainder of τ, G and A pay exactly the same amount. Therefore G incurs a cost within $2p$ of A. □

THEOREM 3.3. *Let A be any on-line block-retention algorithm in a model allowing Supplythrough and Updatethrough. If there are at least two caches then there is an infinite sequence of requests* σ *such that* $C_A(\sigma(n)) \geq n$, *and*

$$C_A(\sigma(n)) \geq 3 \cdot C_{opt}(\sigma(n))$$

for infinitely many values of n, where $\sigma(n)$ *denotes the first n requests of* σ.

PROOF. The sequence σ we construct refers to only two caches, 1 and 2, and one memory block B. We assume that Algorithm A starts with block B unique to cache 1. We construct σ inductively: if A has block B unique to cache 1, the next request in σ is $\text{WRITE}_2(B)$. Similarly, if A has block B unique to cache 2, the next request in σ is $\text{WRITE}_1(B)$. Finally, if A is in any other state, the next request is the same as the previous one.

We partition σ into a sequence of *runs*. Each run is a contiguous subsequence of identical requests, bounded at each end by a different request, or by the beginning of σ. (We assume that there is no infinite run. If there were, the trivial algorithm which stays in one state beats the on-line algorithm by an arbitrary factor infinitely often.) The *length* of a run is the number of requests in it. A run is *long* if its length is at least $2p$, and *short* otherwise.

The cost of σ to A from the beginning of σ to the end of a run is at least p times the number of runs plus the length of all the runs. This is because each request costs A one to satisfy, and during each run A must execute a *Fetchblock*.

The off-line algorithm we shall analyze, H, starts in the same state as A (B unique to cache 1) and oscillates between this state and the one with B unique to cache 2. Algorithm H looks ahead at the sequence of upcoming runs and decides how to process them based on the following rules:

1. If what follows is a sequence of consecutive short runs with total length at least $6p^2 - 3p$ then H chooses to stay in the same state during all of these

runs. The state chosen is the one which minimizes the cost of the possible state transition at the beginning of these runs, plus the cost of processing the remaining runs in that state.

2. If what follows is a sequence of zero or more consecutive short runs with total length less than $6p^2-3p$ followed by a long run, then H processes the long run in the state which makes the requests in it free, and processes the short runs in a fixed state. The state chosen for the short runs is the one which minimizes the state transition costs at the beginning and end of these runs plus the cost of processing the runs in that state.

Algorithm H works in stages. In each state it invokes rule 1 or rule 2, and decides how it will process another portion of σ. We will show that the cost incurred by H during a stage is at most one-third of the cost incurred by A for the corresponding portion of σ. The theorem follows by induction.

Consider a stage for which rule 1 was invoked. Let l be the total length of the runs processed by H in this stage. The two options that H considers are to stay in the state it was in at the end of the previous stage, or to change to the other state at the beginning of the stage. The costs of these options are a and $(l-a)+p$, where a is the number of requests in this portion of σ that are costly to H if it decides to say in the same state. Thus, the cost to H in this stage is $\min(a, l-a+p) \leq \frac{1}{2}(l+p)$.

The cost of this stage to A is l plus p times the number of runs in the stage. Since each run is short, the number of runs is at least $l/2(2p-1)$. Thus the cost to A is least $l+p(l/(2p-1))$. By applying the fact that $l \geq 6p^2-3p$ it is easy to see that this cost is at least $\frac{3}{2}(l+p)$, which is at least three times that incurred by H in this stage.

Consider a stage in which rule 2 was invoked. Let l be the total length of the short runs in this stage, and let a be the number of requests in these short runs that are the same as the requests in the long run that follows. If the state chosen by H for the long run is different from that at the end of the previous stage, then the cost to H of this stage is $\min(p+a, p+l-a)$. If the state chosen by H for the long run is the same as that at the end of the previous stage, then the cost to H of this stage is $\min(2p+a, l-a)$. In either case the cost to H is bounded by $p+\frac{1}{2}l$.

The cost of this stage to A is l plus p times the number of runs in this stage plus the length of the final long run. The number of runs is at least $1+l/2p$. Thus the cost to A is at least $l+p(1+l/2p)+2p=\frac{3}{2}l+3p$, which is at least three times the cost to H. □

We can prove a theorem analogous to Theorem 3.2 in this case as well.

4. Direct-Mapped Snoopy Caching. Our block-retention algorithm for the direct-mapped snoopy cache model, *dsc*, uses an array of counts to decide when to drop a block B from cache i. Each element of this array (denoted $w[i, B]$) takes on an integer value between 0 and p. If a block is replicated, then every write to it requires a bus cycle. Each other cache containing the block is partially guilty

of causing this bus cycle. Consequently, in the following algorithm, a write to a replicated block reduces a counter in one of the other caches sharing the block. When the counter reaches zero the block is dropped. When a block is brought into a cache its count is set to p.

Two invariants are maintained that relate the state of the caches to the $w[i, B]$ values. First, $w[i, B]$ is 0 if and only if block B is not in cache i. Second, if i is the last processor to modify a dirty block B then $w[i, B] = p$.

```
Algorithm Direct-Mapped-Snoopy-Caching;
  for t := 1 to length(σ) do
    if σ(t) = READ_i(v) then
      B := [v];
      if w[i, B] = 0 then Getblock(i, B);
      else w[i, B] := q, where q ∈ [w[i, B] ... p]fi;
      Supply(i, v)
    elsif σ(t) = WRITE_i(v) then
      B := [v];
      if w[i, B] = 0 then Getblock(i, B)
      else w[i, B] := p fi;
      if ∃j ≠ i s.t. w[j, B] ≠ 0 then
        Updateglobal(i, v); C_dsc := C_dsc + 1;
        w[j, B] := w[j, B] − 1;
        if w[j, B] = 0 then Drop(j, B) fi
      else Updatelocal(i, v)
      fi
    fi
  od
end Direct-Mapped-Snoopy-Caching;

Procedure Getblock(i, B);
  if ∃B' s.t. h_i(B') = h_i(B) ∧ w[i, B'] ≠ 0 then
    { B collides with B', so drop B'. }
    if w[i, B'] = 0 then Writeback(i, B'); C_dsc := C_dsc + p fi;
    w[i, B'] := 0;
    Drop(i, B')
  fi;
  Fetchblock(i, B); C_dsc := C_dsc + p;
  w[i, B] := p
end Getblock;
```

Algorithm *dsc* is underdetermined at two points: when $w[i, B]$ is incremented during a $\text{READ}_i(v)$ request, and when j is chosen during a write to a replicated block. It turns out that amortized analysis of the algorithm is insensitive to these choices, so other criteria must be used to make them.

A *Writeback* is done by this algorithm when the count of a block that must be dropped equals p. This is done to maintain the second of the two invariants. The

effect of this is that a *Writeback* may be done while the block is still replicated. There is no advantage in doing such a *Writeback*. Algorithm *dsc* can easily be modified to avoid doing this, but its analysis becomes slightly more complicated.

THEOREM 4.1. *Algorithm dsc is strongly 2-competitive, that is, for any sequence σ and any on-line or off-line algorithm A,*

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

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

PROOF. When any algorithm is run on a sequence of requests σ with a particular initial state of the caches, it generates a sequence of actions. In order to compare the performance of two algorithms on the sequence σ we will need to correlate the actions of the two algorithms. To do this we construct a sequence of actions τ by merging the actions generated by A and *dsc* on input σ in a particular order.

We construct the sequence of actions τ as follows: start with the empty sequence. For each request in σ, we extend τ first by the actions taken by A up to the *Supply* or *Update* that completes the request. We label each of these A. We then extend τ by the actions taken by *dsc*, up to the *Supply* or *Update*. We label each of these *dsc*. Finally, we extend τ by the completing *Supply* or *Update*, which we label with both A and *dsc*. We will denote by $C_{dsc}(\tau, t)$ the cost of the actions labeled with *dsc* in the first t steps of τ. $C_{opt}(\tau, t)$ is defined similarly.

We will prove by induction on t that

$$C_{dsc}(\tau, t) - 2 \cdot C_A(\tau, t) \leq \Phi(t) - \Phi(0), \tag{4.1}$$

where $\Phi(t)$ is a potential function that depends on the cache states of *dsc* and A after t steps of τ. The theorem follows with $k = -\Phi(0)$, since Φ is chosen to be always nonpositive.

For $t = 0$, both sides of (4.1) are 0. The inductive step reduces to showing $\Delta C_{dsc} - 2 \cdot \Delta C_A \leq \Delta\Phi$ where $\Delta \cdot = \cdot(t) - \cdot(t-1)$. Let S_A be the set of pairs (i, B) of caches and blocks such that B is kept in cache i by A after t steps of τ. We take the potential function to be

$$\Phi(t) = \sum_{(i,B) \in S_A} (w[i, B] - 2p) + \sum_{(i,B) \notin S_A} (-w[i, B]).$$

Every step in *dsc* and in A that changes the potential or incurs a cost results in an action in τ. Therefore to prove the theorem it is sufficient to analyze the effect of every type of action in τ. The following case analysis does this.

If step t of τ is an action labeled only with A, then one of the following cases holds:

A. The action is *Fetchblock*(i, B):

$\Delta C_A = p$ and so we must show $\Delta\Phi \geq -2p$. Before this action $(i, B) \notin S_A$.

After the action $(i, B) \in S_A$. Therefore $\Delta\Phi = w[i, B] - 2p - (-w[i, B]) = 2w[i, B] - 2p \geq -2p$.

B. The action is *Drop*(i, B):
$\Delta C_A = 0$ and so we must show $\Delta\Phi \geq 0$. This is the reverse of the previous case. Before the action $(i, B) \in S_A$, and after the action $(i, B) \notin S_A$. The change in potential is $2p - 2w[i, B] \geq 0$.

C. The action is *Writeback*(i, B):
$\Delta C_A = p$ and so we must show $\Delta\Phi \geq -2p$. Here $\Delta C_A = p$ and $\Delta\Phi = 0$, maintaining the assertion.

If step t of τ is an action labeled only with *dsc*, then one of the following cases holds:

A. The action is *Fetchblock*(i, B):
$\Delta C_{dsc} = p$, so we must show that $\Delta\Phi \geq p$. The count $w[i, B]$ changes from 0 to p. Because of the way the actions are ordered in τ, when this *Fetchblock*(i, B) is done it must be the case that $(i, B) \in S_A$. Thus the potential increases by p.

B. The action is *Writeback*(i, B):
$\Delta C_{dsc} = p$, so again we must show that $\Delta\Phi \geq p$. This time $w[i, B']$ changes from p to 0, and $(i, B') \notin S_A$. $\Delta\Phi = -0 - (-p) = p$.

C. The action is *Drop*(i, B'), and was caused by a collision:
The cost of the operation is 0, so we need to show that $\Delta\Phi \geq 0$. The count of the block that is dropped, $w[i, B']$, is set to 0. This cannot decrease the potential since $(i, B') \notin S_A$.

D. The action is *Drop*(j, B), and was caused by a write to a replicated block:
The cost of the operation is 0 and the potential does not change since $w[j, B]$ is 0.

If step t of τ is an action labeled with both A and *dsc* then one of the following cases holds:

A. The action is *Supply*(i, v):
The cost to both *dsc* and to A is 0, and $\Delta w[i, B] \geq 0$. Since $(i, B) \in S_A$, $\Delta\Phi \geq 0$.

B. The action is *Updatelocal*:
This is the same as case A except that A may incur a cost of 1 which just improves the situation.

C. The action is *Updateglobal*. There are two subcases depending on whether or not block B is unique to A:

C1. B is replicated in *dsc* and is unique in A:
$\Delta C_{dsc} = 1$ and $\Delta C_A = 0$, so we need to show that $\Delta\Phi \geq 1$. This is the case since $w[j, B]$ is decreased by 1, and $(j, B) \notin S_A$.

C2. B is replicated in *dsc* and in A:
$\Delta C_{dsc} = \Delta C_A = 1$, so we need to show that $\Delta\Phi \geq -1$. This is the case since $w[j, B]$ changes by 1 causing the potential to change by 1. □

The following theorem is a slightly stronger version of Theorem 4.1. The proof (which we omit) uses two separate potential functions (one for *Fetchblock* costs and one for *Update* costs) and requires a slightly more careful accounting of costs.

THEOREM 4.2. *For any sequence σ and any on-line or off-line algorithm A,*

$$\mathrm{FBC}_{dsc}(\sigma) \leq \mathrm{FBC}_A(\sigma) + \mathrm{UC}_A(\sigma) + k_1$$

and

$$\mathrm{UC}_{dsc}(\sigma) + \mathrm{WBC}_{dsc}(\sigma) \leq \mathrm{FBC}_A(\sigma) + \mathrm{UC}_A(\sigma) + k_2.$$

The constants k_1 and k_2 depend on the initial cache states of dsc and A. If all caches are initially empty then k_1 and k_2 are zero.

5. Associative Caching. We now examine block-retention strategies for the *associative cache model.* In this model a block can reside anywhere in the cache. A strategy in the associative cache model has the burden of deciding which block to drop when a new block is read into a full cache, as well as having to decide which blocks to drop because of writes to replicated blocks. The problem of deciding which block to drop when a new block is read into a full cache is a demand paging problem. Therefore, in order to obtain competitive performance in the associative caching model we must combine demand paging strategies with the standard snoopy caching strategy of Section 4.[7]

We begin by examining demand paging strategies, and formulating their analysis in terms of potential functions. We then apply this machinery to the associative caching problem.

Demand Paging. Consider a two-level memory divided into pages of fixed uniform size. Let n be the number of pages of fast memory. A sequence of page accesses is to be performed, and each access requires that the desired page be put into fast memory. If the page is already in fast memory the access costs nothing. If the page is in slow memory we must swap it for a page in fast memory at a cost of one page fault. A paging rule is an algorithm for deciding which page to move from fast memory to slow memory.

We consider the following paging rules:

Least recently used (lru)

When swapping is necessary, replace the page whose last access was longest ago.

First in, first out (fifo)

Replace the page that has been in fast memory the longest.

Flush when full (fwf)

When attempting to read a page into a full fast memory, discard all other pages.

Longest forward distance (min)

Replace the page whose next access is latest.

All of these but *min* are on-line algorithms. *min* is off-line because it requires knowledge of the sequence in advance. It is also optimal in the sense that it minimizes the number of page faults for any sequence [B].

[7] Our techniques can be applied to the situation in which the cache is k-way set associative. Each set is regarded as an independent cache.

We compare each of the on-line algorithms described above with the *min* algorithm. Let A be any algorithm, n_A the number of pages of fast memory available to A, σ a sequence of page accesses, and $F_A(\sigma)$ the number of page faults made by A on σ. When comparing A and *min*, we assume that $n_A \geq n_{min}$.

Sleator and Tarjan proved the following lower bound:

THEOREM 5.1 [ST]. *Let A be any on-line algorithm. Then there are arbitrarily long sequences σ such that*

$$F_A(\sigma) \geq \left(\frac{n_A}{n_A - n_{min} + 1}\right) F_{min}(\sigma).$$

Sleator and Tarjan also proved that the performance of *lru* and *fifo* is within an additive constant of this lower bound. In their proof they considered sequences on which *lru* makes n_{lru} faults and showed that for those sequences *min* must make $n_{lru} - n_{min} + 1$ faults. We have new analyses of *lru* and *fifo* using potential functions. Using these methods we show that the bound is also tight for *fwf*. Our results are summarized by the following theorem.

THEOREM 5.2. *For any input sequence σ, and A any of the three algorithms fwf, fifo, and lru,*

$$F_A \leq \left(\frac{n_A}{n_A - n_{min} + 1}\right) F_{min}(\sigma) + k,$$

where k depends only on the initial state of the caches and is zero if both sets of caches start out empty.

PROOF SKETCH. The detailed proof of this theorem is presented in Appendix C. Since the snoopy caching algorithm and potential function below are based on *fifo*, we present a brief description of *fifo* here. *fifo* was chosen for the snoopy caching result only because it has the simplest algorithm and potential function. We believe that either of the other strongly competitive paging rules also lead to competitive snoopy caching algorithms.

For each page P, maintain an integer-valued variable $a[P]$ in the range $[0, n_{fifo}]$. $a[P] = 0$ if P is not in fast memory. When page P is read into fast memory, $a[P]$ is set to n_{fifo}, and for all other pages P' in fast memory $a[P']$ is decremented. (The page whose new $a[P]$ value is 0 is the one replaced. This is the page that has been in the fast memory the longest.)

The potential function

$$\Phi(t) = \sum_{P \in S_{min}} \frac{a[P] - n_{fifo}}{n_{fifo} - n_{min} + 1}$$

is used to show that

$$\Delta F_{fifo}(\sigma, t) - \left(\frac{n_{fifo}}{n_{fifo} - n_{min} + 1}\right) \Delta F_{min}(\sigma, t) \leq \Delta\Phi,$$

and hence prove the theorem. □

Combining Caching with Paging. We may now combine caching strategies with paging strategies. First, as an immediate corollary of Theorems 4.1 and 5.1, we obtain:

THEOREM 5.3. *Let A be any on-line algorithm for associative snoopy caching where each cache managed by A has size n_A, and each cache managed by opt has size n_{opt}. Then there are arbitrarily long sequences σ such that*

$$C_A(\sigma) \geq \max\left(\frac{n_A}{n_A - n_{opt} + 1}, 2\right) \cdot C_{opt}(\sigma).$$

Algorithm *scwf*, presented below, nearly achieves this lower bound. This algorithm combines the *fifo* algorithm for paging and the *dsc* algorithm for direct-mapped snoopy caching. For each block B and each cache i we maintain two variables:

$a[i, B]$: a real-valued variable in the range $[0, n_{scwf}]$ that roughly represents the maximum number of other blocks cache i can read until block B in cache i is invalidated.

$w[i, B]$: an integer-valued variable in the range $[0, p]$. $w[i, B] = 0$ if and only if block B is not present in cache i.

Algorithm *Snoopy-Caching-With-fifo*;
  for $t := 1$ to length(σ) **do**
    if $\sigma(t) = \text{READ}_i(v)$ **then**
      $B := [v]$;
      if $w[i, B] = 0$ **then** *Getblock*(i, B) **fi**;
      Supply(i, v)
    elsif $\sigma(t) = \text{WRITE}_i(v)$ **then**
      $B := [v]$;
      if $w[i, B] = 0$ **then** *Getblock*(i, B);
      else $w[i, B] := p$ **fi**;
      if $\exists j$ **s.t.** $w[j, B] \neq 0 \wedge j \neq i$ **then** { block B is replicated }
        Updateglobal(i, v);
        $\text{UC}_{scwf} := \text{UC}_{scwf} + 1$;
        $w[j, B] := w[j, B] - 1$; $a[j, B] := \min\left(a[j, B], \frac{w[j, B]}{p} n_{scwf}\right)$ (*)
        if $w[j, B] = 0$ **then** *Drop*(j, B) **fi**
      else *Updatelocal*(i, v) **fi**;
    fi
  od
end *Snoopy-Caching-With-fifo*;

Procedure *Getblock*(i, B);
{ first do *fifo* decrement }
for B' **s.t.** $w[i, B'] \neq 0$ **do**
$a[i, B'] := a[i, B'] - 1$;
if $a[i, B'] \leq 0$ **then**
{ may need space, drop B' }
if B' dirty and i is the last cache storing it **then**
Writeback(i, B'); $\mathrm{WBC}_{scwf} := \mathrm{WBC}_{scwf} + p$
fi;
Drop(i, B'); $a[i, B'] := 0$; $w[i, B'] := 0$
fi
od;
{ cache is no longer full, get block }
Fetchblock(i, B); $a[i, B] := n_{scwf}$; $w[i, B] := p$;
$\mathrm{FBC}_{scwf} := \mathrm{FBC}_{scwf} + p$
end *Getblock*;

The only link in this algorithm between the variable a accounting for paging and the variable w accounting for writes to replicated blocks is on line $(*)$. The effect of this line is to keep the a variable at roughly the same proportion to its maximum value as the w variable is to its maximum value. This technicality is necessary to prevent a large potential swing when a block is invalidated due to replicated writes. As a consequence, we may drop more than one block when a new block is read in.

THEOREM 5.4. *Let A be any algorithm (on-line or off-line) for deciding on block retention in an associative snoopy cache. Let n_{scwf} be the size of the caches managed by scwf and n_A be the size of the caches managed by A.*

Then, for all sequences σ,

$$\mathrm{FBC}_{scwf}(\sigma) \leq \left(\frac{n_{scwf}}{n_{scwf} - n_A + 1}\right)(\mathrm{FBC}_A(\sigma) + \mathrm{UC}_A(\sigma)) + k_1, \tag{5.1}$$

$$\mathrm{UC}_{scwf}(\sigma) \leq \mathrm{FBC}_A(\sigma) + \mathrm{UC}_A(\sigma) + k_2, \tag{5.2}$$

where k_1 and k_2 are zero, if the caches are initially empty. Hence, the combined Fetchblock and Update costs of scwf are $(1 + n_{scwf}/(n_{scwf} - n_A + 1))$-competitive.

PROOF. As in the proof of Theorem 4.1 we let τ denote the labeled, merged sequence of actions taken by *scwf* and A. At each step t of τ, S_A is the set of pairs (i, B) such that Algorithm A has block B stored in cache i. The potential functions

$$\Phi_F(t) = \sum_{(i,B) \in S_A} p \cdot \frac{a[i, B] - n_{scwf}}{n_{scwf} - n_A + 1}$$

and

$$\Phi_U(t) = \sum_{(i,B) \in S_A} (-p) + \sum_{(i,B) \notin S_A} (-w[i, B])$$

are used to show that the invariants

$$\Delta\mathrm{FBC}_{scwf}(\tau, t) - \left(\frac{n_{scwf}}{n_{scwf} - n_A + 1}\right)(\Delta\mathrm{FBC}_A(\tau, t) + \Delta\mathrm{UC}_A(\tau, t)) \leq \Delta\Phi_F(\tau, t)$$

and

$$\Delta\mathrm{UC}_{scwf}(\tau, t) - (\Delta\mathrm{FBC}_A(\tau, t) + \Delta\mathrm{UC}_A(\tau, t)) \leq \Delta\Phi_U(\tau, t)$$

hold. From these equations (5.1) and (5.2) above follow.

If step t of τ is an action labeled only with A, then one of the following cases holds:

A. The action is *Fetchblock*(i, B):
$\mathrm{FBC}_A = p$ and so we must show

$$\Delta\Phi_F \geq \frac{-pn_{scwf}}{n_{scwf} - n_A + 1} \quad \text{and} \quad \Delta\Phi_U \geq -p.$$

(i, B) enters S_A, so

$$\Delta\Phi_F = \frac{p(a[i, B] - n_{scwf})}{n_{scwf} - n_A + 1} \geq \frac{-pn_{scwf}}{n_{scwf} - n_A + 1} \quad \text{and} \quad \Delta\Phi_U = -p - (-w[i, B]) \geq -p.$$

B. The action is *Drop*(i, B):
$\mathrm{C}_A = 0$ and so we must show $\Delta\Phi_F, \Delta\Phi_U \geq 0$. (i, B) exits S_A, so

$$\Delta\Phi_F = p\left(\frac{n_{scwf} - a[i, B]}{n_{scwf} - n_A + 1}\right) \geq 0 \quad \text{and} \quad \Delta\Phi_U = p - w[i, B] \geq 0.$$

If step t of τ is an action labeled only with *scwf*, then one of the following cases holds:

A. The action is *Fetchblock*(i, B):
$\mathrm{FBC}_{scwf} = p$ and so we must show $\Delta\Phi_F \geq p$, $\Delta\Phi_U \geq 0$. $\Delta w[i, B] = p$, $\Delta a[i, B] = n_{scwf}$, and at most $n_A - 1$ other pages in S_A have their a values decremented. Since A's actions precede *scwf*'s, $(i, B) \in S_A$ already, and so

$$\Delta\Phi_F \geq p\left(\frac{n_{scwf} - (n_A - 1)}{n_{scwf} - n_A + 1}\right) = p.$$

$\Delta\Phi_U \geq 0$, since some w's may decrease.

B. The action is *Drop*(i, B), and was caused by fetching a block into a full cache:
$\mathrm{C}_{scwf} = 0$ and so we must show $\Delta\Phi_F, \Delta\Phi_U \geq 0$. If $(i, B) \in S_A$, then, since $\Delta a[i, B] \geq 0$, $\Delta\Phi_F \geq 0$, and, since $w[i, B]$ decreases to 0, $\Delta\Phi_U \geq -\Delta w[i, B] \geq 0$.

C. The action is *Drop*(j, B), and was caused by a write to a replicated block:
$\mathrm{C}_{scwf} = 0$ and so we must show $\Delta\Phi_F, \Delta\Phi_U \geq 0$. Since S_A is unchanged and this case only happens when $w[i, B]$ has already reached 0 (and then by (*) $a[i, B]$ is also already 0), $\Delta\Phi_F = \Delta\Phi_U = 0$.

If step t of τ is an action labeled with both A and *scwf*, then one of the following cases holds:

A. The action is *Supply*(i, v):
$C_A = C_{scwf} = 0$ and so we must show $\Delta\Phi_F, \Delta\Phi_U \geq 0$. $\Delta a[i, B] = \Delta w[i, B] = 0$. Since $(i, B) \in S_A$, $\Delta\Phi_F = \Delta\Phi_U = 0$.

B. The action is *Updatelocal*(i, v):
As $UC_{scwf} = 0$ we must show

$$\Delta\Phi_F \geq -\frac{n_{scwf}}{n_{scwf} - n_A + 1}\Delta UC_A \quad \text{and} \quad \Delta\Phi_U \geq -\Delta UC_A.$$

$\Delta w[i, B] \geq 0$, $\Delta a[i, B] = 0$, and $(i, B) \in S_A$, hence

$$\Delta\Phi_F = 0 \geq -\frac{n_{scwf}}{n_{scwf} - n_A + 1}\Delta UC_A \quad \text{and} \quad \Delta\Phi_U = 0 \geq -\Delta UC_A.$$

C. The action is *Updateglobal*(i, v). There are two subcases depending on whether or not block B is unique to A:

C1. B is replicated in *scwf* and is unique in A:
$\Delta UC_A = 0$, $\Delta UC_{scwf} = 1$ and so we must show $\Delta\Phi_F \geq 0$ and $\Delta\Phi_U \geq 1$. Since $\Delta UC_A = 0$, $\forall j \neq i$, $(j, B) \notin S_A$. Hence, $\Delta\Phi_F = 0$; $\Delta\Phi_U = -\Delta w[j, B] \geq 1$.

C2. B is replicated in *scwf* and in A:
$\Delta UC_A = \Delta UC_{scwf} = 1$ and so we must show

$$\Delta\Phi_F \geq -\frac{n_{scwf}}{n_{scwf} - n_A + 1} \quad \text{and} \quad \Delta\Phi_U \geq 0.$$

$$\Delta\Phi_F \geq \frac{p}{n_{scwf} - n_A + 1}\Delta a[j, B].$$

Because of (*), the invariant $a[j, B] \leq (w[j, B]/p)n_{scwf}$ is maintained. Hence, $\Delta a[j, B] \geq -n_{scwf}/p$, and so

$$\Delta\Phi_F \geq \frac{p}{n_{scwf} - n_A + 1}\left(\frac{-n_{scwf}}{p}\right).$$

If $(j, B) \notin S_A$, $\Delta\Phi_U = -\Delta w[j, B] = 1$, otherwise $\Delta\Phi_U = 0$. □

COROLLARY. *Algorithm scwf is* $(1 + 2 \cdot [n_{scwf}/(n_{scwf} - n_A + 1)])$*-competitive.*

PROOF. The cost of writebacks to *scwf* is bounded by the cost of reads. □

6. General Snoopy Caching. Like associative caching, general snoopy caching is a generalization of direct-mapped snoopy caching. Here we maintain the direct-mapped feature, but allow a different freedom. A cache can use the *Supplythrough* and *Updatethrough* actions to read or modify a variable in a block that is not in the cache. This gives the cache the freedom to decide both when a block should be fetched as well as when it should be dropped.

Our algorithm for general snoopy caching, *gsc*, maintains two variables, $w[i, B]$ and $x[i, B]$ for each cache i and block B. Each variable assumes an integer value in the range 0 to p. As in *dsc*, $w[i, B] \neq 0$ if and only if block B is in cache i. Furthermore, if $x[i, B] \neq 0$ then block B is not in cache i. Therefore at any time one or both of $w[i, B]$ or $x[i, B]$ is 0. As in *dsc*, $w[i, B]$ decreases as cache i loses interest in keeping its copy of block B. Analogously $x[i, B]$ increases as cache i gains interest in getting its own copy of block B.

In the program below C'_{gsc} denotes the cost incurred by *gsc* for all actions except *Writebacks*. The numbers in braces are labels used in the analysis.

```
Algorithm General-Snoopy-Caching;
  for t := 1 to length(σ) do
    if σ(t) = READ_i(v) then
      B := [v];
      if w[i, B] = 0 then Adjust(i, B) fi;
      if w[i, B] = 0 then Supplythrough(i, v) {1}; C'_gsc := C'_gsc + 1
      else Supply(i, v) {2}
      fi
    elsif σ(t) = WRITE_i(v) then
      B := [v];
      if w[i, B] = 0 then Adjust(i, B) fi;
      if w[i, B] = 0 then Updatethrough(i, v) {3}; C'_gsc := C'_gsc + 1
      elsif w[i, B] < p then
        w[i, B] := w[i, B] + 1;
        Updateglobal(i, v) {4}; C'_gsc := C'_gsc + 1
      elsif w[i, B] = p then
        if block B is in another cache j then
          Updateglobal(i, v) {5}; C'_gsc := C'_gsc + 1;
          w[j, B] := w[j, B] - 1;
          if w[j, B] = 0 then Drop(j, B) {6} fi
        else Updatelocal(i, v) {7}
        fi
      fi
    fi
  od
end General-Snoopy-Caching;

Procedure Adjust(i, B);
  if x[i, B] < p then {a} x[i, B] := x[i, B] + 1
  elsif another block B' is occupying the cache line h_i(B) then {b}
```

```
    w[i, B'] := w[i, B'] − 1;
    if w[i, B'] = 0 then
      if B' dirty then Writeback(i, B') {8} fi;
      Drop(i, B') {9}
    fi
  elsif x[i, B] = p ∧ cache line h_i(B) is empty then {c}
    Fetchblock(i, B) {10}; C'_gsc := C'_gsc + p;
    x[i, B] := 0; w[i, B] := p
  fi
end Adjust;
```

In procedure *Adjust* one of the three options {a}, {b}, or {c} will be taken. The analysis below applies to a slightly more general version of the algorithm in which any of these three options for which the guard is satisfied may be taken. (For example, option {b} may be taken if its guard is satisfied regardless of whether the guard of option {a} is satisfied.)

THEOREM 6.1. *For any sequence σ and any on-line or off-line algorithm A,*

$$C'_{gsc}(\sigma) \leq 3 \cdot C'_A(\sigma) + k,$$

where the costs do not count Writeback. The constant k depends on the initial cache states of gsc and A. If both cache sets are initially empty then k is zero.

PROOF. As before we let τ denote the labeled, merged sequence of actions taken by *gsc* and *A*. The action taken by *gsc* that satisfies a request (*Supply*, *Supply-through*, *Update*, or *Updatethrough*) and that used by *A* to satisfy the same request are combined together to make a single element of τ, even though they may be different types of actions.

The potential function

$$\Phi(t) = \sum_{(i,B) \in S_A} (2w[i, B] + x[i, B] - 3p) + \sum_{(i,B) \notin S_A} (-w[i, B] - 2x[i, B])$$

will be used. For each action we will verify the assertion

$$\Delta C'_{gsc} - 3\Delta C'_A \leq \Delta\Phi,$$

which suffices to prove the theorem.

If step t of τ is an action labeled only with *A*, then one of the following cases holds:

A. The action is *Fetchblock*(i, B):
$\Delta C_A = p$ and so we must show $\Delta\Phi \geq -2p$. Before this action $(i, B) \notin S_A$. After the action $(i, B) \in S_A$. Therefore $\Delta\Phi = 2w[i, B] + x[i, B] - 3p - (-w[i, B] - 2x[i, B]) = 3(w[i, B] + x[i, B] - p) \geq -3p$. Since $\Delta C'_A = p$, the assertion is verified.

B. The action is *Drop*(i, B):
$\Delta C_A = 0$ and so we must show $\Delta\Phi \geq 0$. Before this action $(i, B) \in S_A$. After the action $(i, B) \notin S_A$. Therefore $\Delta\Phi = 3(p - w[i, B] - x[i, B]) \geq 0$. The cost of the operation is 0, so the assertion is verified.

If step t of τ is an action labeled only with *gsc*, then one of the following cases holds:

A. The action is *Fetchblock*(i, B) {10}:
$\Delta C'_{gsc} = p$, so we must show that $\Delta\Phi \geq p$. The count $w[i, B]$ changes from 0 to p, and $x[i, B]$ changes from p to 0. Regardless of whether $(i, B) \in S_A$ or not, $\Delta\Phi = p$.

B. The action is *Drop* {6} {9}:
$\Delta C'_{gsc} = 0$, so we must show that $\Delta\Phi \geq 0$. At both places in the program where a drop is issued, there is no change in potential and no cost is incurred.

It remains for us to deal with the situation in which step t of τ is an action labeled with both *gsc* and A. The action taken by *gsc* is one of the statements labeled {1}, {2}, {3}, {4}, {5}, or {7}. The action taken by A is one of two types, free (*Supply* or *Updatelocal*) or costly (*Supplythrough*, *Updateglobal* or *Updatethrough*).

A. {1} or {3}:
$\Delta C'_{gsc} = 1$ and so we must show that $\Delta\Phi \geq 1 - 3\Delta C_A$. Just before the action, *Adjust* was called and option {a}{b} was chosen. (Option {c} was not taken because if it had been, then $w[i, B]$ would not be 0, and {1} or {3} would not have been done.) There are four cases depending on whether option {a} or {b} was chosen and on whether the action was free or costly to A. Suppose option {b} was chosen and the action was free to A. Then $w[i, B']$ is decreased by 1, and $(i, B') \notin S_A$, so the potential increases by 1, which is required to maintain the assertion. The other three cases are similar and are easy to verify.

B. {2} or {7}:
These actions cost nothing and do not change the potential.

C. {4}:
$\Delta C'_{gsc} = 1$ and so we must show that $\Delta\Phi \geq 1 - 3\Delta C_A$. The statement before this action increases $w[i, B]$ by 1. If the action is costly to A then $(i, B) \notin S_A$, and $\Delta\Phi = -1$. If the action is free to A then $(i, B) \in S_A$, and $\Delta\Phi = 2$. Either way, there is sufficient potential increase.

D. {5}:
$\Delta C'_{gsc} = 1$ and so we must show that $\Delta\Phi \geq 1 - 3\Delta C_A$. The count $w[j, B]$ has just been decreased by 1. Whether (j, B) is in S_A or not, $\Delta\Phi \geq -2$, so if the action is costly to A then the assertion is verified. If the action is free to A, then $(j, B) \notin S_A$, and $\Delta\Phi = 1$, verifying the assertion. □

Combined with Theorem 3.3, this shows that Algorithm *gsc* is strongly 3-competitive when writeback costs are ignored. We do not know if *gsc* is strongly competitive if these costs are considered. However, the following corollary shows that it is 4-competitive.

COROLLARY. *Algorithm gsc is 4-competitive*

PROOF. For every *Writeback*(i, B) there is a sequence of actions leading up to it that have a cost of $3p$: initially $w[i, B] = x[i, B] = 0$. Before block B is fetched, $x[i, B]$ is increased from 0 to p, at a cost of 1 for each of these actions. Block B is then fetched at a cost of p. Finally, before the writeback, $w[i, B]$ is decreased from p to 0, at a total cost of p (for the *Updatethrough*'s to other blocks). Thus we have

$$\mathrm{WBC}_{gsc} \leq \tfrac{1}{3}\mathrm{C}'_{gsc}.$$

Combining this with the fact that $\mathrm{C}_{gsc} = \mathrm{C}'_{gsc} + \mathrm{WBC}_{gsc}$ and Theorem 6.1 gives the result. □

7. Block Snoopy Caching. The models we have considered so far allow snooping only on *Update* actions. We now propose a model which allows snooping on *Fetchblock* and *Writeback* actions as well.

In the *block snoopy caching* model each processor has a cache of size equal to all of memory. Every block will be stored in some cache, so in this model there is no main memory. When a block is sent over the bus in response to a READ, other caches can grab the data at no additional cost. Since writes are just as costly if a block is replicated in two caches as in all of them, we assume that every block is either unique or replicated in all caches.

This model is only realistic if one considers that the function of the cache is to reduce memory bus contention, not to conserve on memory. If the amount of shared memory in a system is relatively small, this may be a practical restriction. Even in settings where this model is not realistic, the principle of block caching is interesting, and leads us to the more practical model of the next section.

In Algorithm *bsc* we make a block unique only after some processor writes to it p times without any other processor reading or writing the block. To record the number of uninterrupted writes the last writer has performed on a block B, we maintain the following state variables:

$last[B] = i$, where i is the last cache to issue an update action to block B.
$w[B] = \min(n, p)$, where n is the number of uninterrupted updates $last[B]$ has issued to block B.

Algorithm *bsc* maintains the invariant that $w[B] = p$ if and only if B is unique to cache $last[B]$.

Algorithm *Block-Snoopy-Caching*;
  for $t := 1$ to length(σ) **do**
    if $\sigma(t) = \mathrm{READ}_i(v)$ **then**
      $B := [v]$;
      if $w[B] = p \wedge last[B] \neq i$ { block B unique to some other cache } **then**
        Fetchblock(i, B); $w[B] := 0$; $\mathrm{C}_{bsc} := \mathrm{C}_{bsc} + p$
          { B now replicated in all caches }

```
      elsif last[B] ≠ i { B replicated, but i not last writer } then
        w[B] := 0
      fi;
      Supply(i, v)
    elsif σ(t) = WRITE_i(v) then
      B := [v];
      if w[B] < p ∧ i = last[B] { B replicated and i last writer } then
        Updateglobal(i, v) {1}; w[B] := w[B] + 1; C_bsc := C_bsc + 1;
        if w[B] = p then { make B unique to i }
          for j s.t. j ≠ i do Drop(j, B) od
        fi
      elsif w[B] < p ∧ i ≠ last[B] { B replicated and i not last writer } then
        w[B] := 1; last[B] := i;
        Updateglobal(i, v) {2}; C_bsc := C_bsc + 1
      elsif w[B] = p ∧ i ≠ last[B] { B unique to some other cache } then
        Fetchblock(i, B); w[B] := 0; C_bsc := C_bsc + p;
      { B now replicated in all caches }
          last[B] := i;
          Updateglobal(i, v) {3}; w[B] := w[B] + 1; C_bsc := C_bsc + 1
      elsif w[B] = p ∧ i = last[B] { B unique to i } then
        Updatelocal(i, v)
      fi
    fi
  od
end Block-Snoopy-Caching;
```

THEOREM 7.1. *Algorithm bsc is strongly 2-competitive, that is, for any sequence* σ *and any on-line or off-line algorithm A in the block snoopy caching model,*

$$C_{bsc}(\sigma) \leq 2 \cdot C_A(\sigma) + k,$$

where k is a constant that depends on the relative initial cache states of bsc and A. If every block is initially unique to its last writer then $k = 0$.

PROOF SKETCH. As usual we let τ denote the labeled, merged sequence of actions taken by *bsc* and *A*. At time *t*, *L* is the set of blocks *B* which *A* stores only in cache *last*[*B*], and *S* is the set of blocks *B* which *A* stores in some cache other than *last*[*B*]. In this model, *S* is the complement of *L*, since every block is in some cache. The potential function

$$\Phi(t) = \sum_{B \in S} (-w[B] - p) + \sum_{B \in L} (w[B] - p)$$

is used to prove the theorem. The details are presented in Appendix D. □

8. Limited Block Snoopy Caching. The block snoopy caching model presented in Section 7 is sometimes unrealistic, since it assumes caches as large as memory. In this section we consider *limited block snoopy caching*, a version of block snoopy caching for direct-mapped caches. In the limited snoopy caching model, a cache is allowed to grab a block only if that block was the last to occupy its cache line. Caches can grab B on both *Fetchblock* and *Writeback* actions. It is assumed in this model that main memory snoops on all transactions. (Thus, main memory acts like all other participants on the bus, except that it has no collisions. It is therefore equivalent to giving each block B some cache i in which $h_i(B) = h_i(B') \Rightarrow B = B'$, distributing the functionality of main memory.) Hence, a block can become dirty only when it has been written to since it became unique to some cache.

The algorithm we propose, *lbsc*, retains the simplicity of block snoopy caching and the practicality of direct-mapped snoopy caching. We believe that it is the most promising algorithm in this paper. This model is not as general as others we consider, but we see no way of having practical on-line algorithms take advantage of greater flexibility.

Algorithm *lbsc* maintains the variables $w[B]$ and $last[B]$ just as Algorithm *bsc*, except that $last[B]$ takes on the value -1 if the block B is present only in main memory. Algorithm *lbsc* maintains two additional variables:

Store$[B]$: The set of processors storing block B.
Reserved$[B]$: The set of processors that dropped block B and have not since fetched any new block into cache line $h_i(B)$.

```
Algorithm Limited-Block-Snoopy-Caching;
  for t := 1 to length(σ) do
    if σ(t) = READ_i(v) then
      B := [v];
      if i ∉ Store[B] then { block B not present in i's cache }
        Getblock(i, B)
      else { do read at no cost }
        if last[B] ≠ i { B replicated, but i not last writer } then
          w[B] := q, where q ∈ [0 · · · w[B]] fi
      fi;
      Supply(i, v)
    elsif σ(t) = WRITE_i(v) then
      B := [v];
      if i ∉ Store[B] then Getblock(i, B) fi;
      if w[B] < p ∧ i = last[B] { i last writer } then
        w[B] := w[B] + 1; Updateglobal(i, v); C_lbsc := C_lbsc + 1;
        if w[B] = p then { make B unique to i }
          for j s.t. j ≠ i do Drop(j, B) od;
          Reserved[B] := Store[B] − {i}; Store[B] := {i}
        fi
      elsif i ≠ last[B] { B actively replicated } then
        w[B] := 1; last[B] := i; Updateglobal(i, v); C_lbsc := C_lbsc + 1
```

```
            elsif i = last[B] ∧ w[B] = p then
              Updatelocal(i, v)
            fi
          fi
        od
      end Limited-Block-Snoopy-Caching;

  Procedure Getblock(i, B);
    for B' s.t. h_i(B') = h_i(B) ∧ i ∈ Store[B'] do
      if i = last[B'] ∧ w[B'] = p then
        Writeback(i, B'); w[B'] := 0; C_lbsc := C_lbsc + p;
        Store[B'] := Store[B'] ∪ Reserved[B']; Reserved[B'] := ∅; last[B] := −1
      fi;
      Drop(i, B'); Store[B'] := Store[B'] − {i};
      if last[B] = i then last[B] := −1 fi
      { the cache line is now empty, do read }
    od;
    Fetchblock(i, B); w[B] := 0; { B replicated by everyone in reserved set }
    C_lbsc := C_lbsc + p;
    Store[B] := Store[B] ∪ Reserved[B] ∪ {i}; Reserved[B] := ∅
  end Getblock;
```

THEOREM 8.1. *Algorithm lbsc is strongly 2-competitive, that is, for any sequence* σ *and any on-line or off-line algorithm A,*

$$C_{lbsc}(\sigma) \leq 2 \cdot C_A(\sigma) + k,$$

where k is a constant that depends on the relative initial cache states of lbsc and A. If both cache sets are initially full, and all blocks are dirty, then $k = 0$.

PROOF. The potential function

$$\Phi(t) = \sum_{B \text{ clean to } A} (-w[B]) + \sum_{B \text{ dirty to } A} (w[B]) - \sum_i p \cdot n_l$$

is used to prove this theorem. The quantity n_l is the number of cache lines in a cache. The third term in the potential is used merely to ensure that the potential is always nonpositive. Note that if B is dirty to A, then B is also unique to A.

If step t of τ is an action labeled only with A, then one of the following cases holds:

A. The action is *Fetchblock*(i, B):
   $\Delta C_A = p$ and so we must show $\Delta\Phi \geq -2p$. A either stays clean or becomes clean. In the former case $\Delta\Phi = 0$ and in the latter $\Delta\Phi = -w[B] - w[B] = -2w[B] \geq -2p$.
B. The action is *Drop*(i, B):
   $\Delta C_A = 0$ and so we must show $\Delta\Phi \geq 0$. Since the drop does not change the fact that A is clean or dirty, and $\Delta w[B] = 0$, $\Delta\Phi = 0$.

C. The action is *Writeback*(i, B):
$\Delta C_A = p$ and so we must show $\Delta\Phi \geq -2p$. Since A goes from dirty to clean, $\Delta\Phi = -2w[B] \geq -2p$.

If step t of τ is an action labeled only with *lbsc*, then one of the following cases holds:

A. The action is *Fetchblock*(i, B):
$\Delta C_{lbsc} = p$ and so we must show $\Delta\Phi \geq p$. There are two cases. Since A's actions precede *lbsc*'s, either A just fetched block B or else A already had B in cache i. In the former case, B is clean to A. In the latter case, $i \in Reserved[B]$ and *lbsc* has B unique to $last[B]$. Since $i \neq last[B]$, B is clean to A and so in either case $\Delta\Phi = -\Delta w[B] = p$.

B. The action is *Writeback*(i, B):
$\Delta C_{lbsc} = p$ and so we must show $\Delta\Phi \geq p$. Since we only writeback when $i = last[B]$, A cannot have B in $last[B]$, since A suffers the same cache collisions. Hence A has B clean. Therefore, $\Delta\Phi = -\Delta w[B] = p$.

C. The action is *Drop*(i, B):
$\Delta C_{lbsc} = 0$ and so we must show $\Delta\Phi \geq 0$. Since $w[B]$ is constant, and A's state does not change, $\Delta\Phi = 0$.

If step t of τ is an action labeled with both A and *lbsc*, then one of the following cases holds:

A. The action is *Supply*(i, v):
$\Delta C_A = \Delta C_{lbsc} = 0$ and so we must show $\Delta\Phi \geq 0$. If $i = last[B]$, then $\Delta w[B] = 0$ and $\Delta\Phi = 0$. Otherwise, if $i \neq last[B]$, then B must be clean to A, $\Delta w[B] \leq 0$ and so $\Delta\Phi \geq 0$.

B. The action is *Updatelocal*(i, v):
$\Delta C_{lbsc} = 0$ and so we must show $\Delta\Phi \geq -2\Delta C_A$. Since $\Delta w[B] = 0$ and A does not change state, $\Delta\Phi = 0 \geq -2\Delta C_A$.

C. The action is *Updateglobal*(i, v). There are two cases depending on whether or not block B is unique to A:

C1. B is replicated in *lbsc* and is unique in A:
$\Delta C_A = 0$, $\Delta C_{lbsc} = 1$ and so we must show $\Delta\Phi \geq 1$. Since $\Delta C_A = 0$, A has B unique. If $i = last[B]$, then to A, B either remains dirty or becomes dirty. In the former case $\Delta\Phi = \Delta w[B] = 1$ and in the latter case $\Delta\Phi = w[B] + 1 + w[B] \geq 1$. If $i \neq last[B]$, then A has B clean before the write, since the writer is changing and B is unique, and B is dirty after the write. Hence, $\Delta\Phi = (1) - (-w) \geq 1$.

C2. B is replicated in *lbsc* and in A:
$\Delta C_A = \Delta C_{lbsc} = 1$ and so we must show $\Delta\Phi \geq -1$. Since $\Delta C_A = 1$, B is clean to A. If $i = last[B]$ before the write, then $\Delta\Phi = -(w[B] + 1) + w[B] = -1$ and if $i \neq last[B]$ before the write, then $\Delta\Phi = -1 + w[B] \geq -1$. □

9. Gateways and Multibus Models. As the number of processors on a bus increases, the bounded communication capacity of the bus becomes a more serious bottleneck. In this section we consider architectures that address this

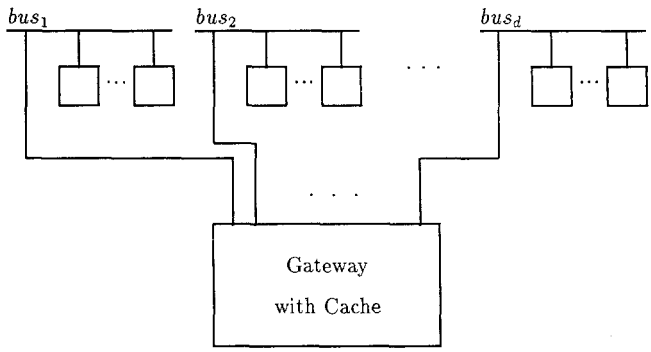

Fig. 1. A gateway connection.

problem by having several interconnected buses. Since caches attached to different buses can share blocks, there must be some mechanism for buses to communicate with one another.

The interconnection mechanism that we propose is a *gateway*. A gateway connects d block snoopy caching buses, acting like a regular participant on each of the buses (see Figure 1). Each of these buses can then be connected to additional buses via other gateways. In this fashion, trees of buses can be built.

The operation of the gateway is as follows. The gateway snoops all of the buses, keeping in its memory all blocks replicated in any of them. The gateway handles a read request from one of the buses by either supplying the requested block or forwarding the request to the bus where the block is stored. Shared writes by one of the buses are forwarded as necessary to the other buses, with the gateway identifying itself as the writer to the latter buses. This masquerade causes the other buses to invalidate the block after p consecutive writes through the gateway. Note that if this did not happen, it would be impossible for the caching algorithm to be competitive.

Before delving into a detailed analysis of the multibus, we should point out that although this model is based on the assumption of unbounded caches, there are several reasons for going through it in detail. First, the model is of theoretical interest in its own right. Second, it suggests that closely related practical variants may have good amortized performance. In particular, these results apply to a multibus of limited block snoopy caching buses, after making some suitable restrictions on the behavior of a gateway. (When a gateway takes a collision, we must invalidate that block entirely on one side of the gateway. We restrict both the on-line and off-line algorithm to invalidate on the side not containing the last writer.) Our current model, being simpler, provides a good expository device. Finally, the specific multibus system suggested at the end of this section, the d-dimensional n-cube, and the proposal for partitioning the memory among several different spanning trees (of buses) is a promising idea, independent of the caching algorithm.

Let the d buses connected by a particular gateway be numbered $1 \cdots d$. The gateway will utilize an integer-valued variable $Has[B]$ for each block B. $Has[B]$

will take values in the range $[0\,..\,d]$. If $Has[B]=0$ then the gateway itself is storing B. Otherwise the value of $Has[B]$ is the number of the bus on which block B is stored. (It will always be the case that if more than one of the buses to which the gateway is connected stores B, then so does the gateway.) Each bus treats the gateway as an ordinary cache in a block snoopy caching system and uses Algorithm *bsc* to manage the caches. (We let $w_j[B]$ and $last_j[B]$ be the $w[B]$ and $last[B]$ values for the jth bus connected to the gateway.)

We call the multibus algorithm *mb*. We will assume that initially each block is stored in a single cache in the system. The procedures below specify the operation of the gateway in response to the read and write requests it sees on the buses it interconnects.

procedure *Service-Read*$_j(i, B)$;
{ response of the gateway to a read request by processor i on the jth bus for block B. }
  if $w_j[B]=p \wedge last_j[B]=$ gateway **then**
    if $Has[B]=0$
      then Broadcast block B on bus j
    else
      Issue a read request by the gateway for block B on bus $Has[B]$;
      $Has[B]:=0$;
      When read request is satisfied, broadcast B on bus j;
    fi
  elsif $w_j[B]=p \wedge last_j[B]=k$ $(k \neq$ gateway$)$ **then**
    When processor k broadcasts B, gateway snoops bus and picks up B;
    $Has[B]:=0$;
  fi
end *Service-Read*;

procedure *Service-Write*$_j(i, B)$
{ response of the gateway to a write request by processor i on the jth bus for block B. }
  for $k \neq j \wedge w_k[B]<p$ $(1 \leq k \leq d)$ **do in parallel**
    Issue a write request by the gateway for block B on bus k
  od
end *Service-Write*;

We will now show that for Algorithm *mb* the traffic per bus is at most twice optimal.

We begin by showing that from the point of view of a bus J, the gateway and buses on the other side of the gateway just look like an ordinary processor on a snoopy reading bus. Let g be a gateway connecting bus J to a tree of buses T. Call this multibus system $\mathcal{B}$ and let σ be a sequence of READ and WRITE requests by processors in $\mathcal{B}$. Let $\mathcal{B}'$ be the system $\mathcal{B}$ where the tree T and gateway g are replaced by a single processor q, storing all blocks stored on any processor or gateway in $T \cup g$. Let σ' be the sequence σ of READ/WRITE requests in which any request made by a processor or gateway in $T \cup g$ is replaced by the

same request issued by processor q. Formally, we define $C_A^{\mathcal{B}}(\sigma, J)$ to be the cost incurred by bus J in executing Algorithm A on sequence σ on the system $\mathcal{B}$.

LEMMA 9.1. *Let A be any on-line or off-line algorithm for processing a sequence σ of* READ/WRITE *requests on the multibus system $\mathcal{B}$. Then*

$$C_A^{\mathcal{B}'}(\sigma', J) \leq C_A^{\mathcal{B}}(\sigma, J).$$

PROOF. We claim that a read request by q in σ' uses the bus J only if the gateway g had to use J to satisfy the corresponding request in σ. This is because processor q in $\mathcal{B}'$ stores all blocks stored in $T \cup g$ in $\mathcal{B}$ and hence could satisfy the request locally if A had any opportunity and would have done so on $\mathcal{B}$.

A write request by q for block B in σ needs to be forwarded across bus J only if some processor on J is storing B in $\mathcal{B}'$. Since that same processor would be storing B in $\mathcal{B}$, the gateway would have had to forward that write request across bus J anyway, to maintain cache consistency. □

The effect of Algorithm *mb* will be to simulate the clustering of $T \cup g$ into a single processor by maintaining the fiction that any request the gateway g forwards to bus J is in fact issued by the gateway itself. Hence, the gateway will create the illusion (with respect to bus J) of being a regular participant in a block snoopy caching bus.

Since the total cost incurred on bus J by our algorithm is bounded by twice the cost of any other Algorithm A on a standard block snoopy caching bus, and A's cost decreases by clustering everything beyond the gateway into a single "processor," we will be able to prove that with respect to each bus on the multibus system, the algorithm is competitive.

First, we must show that the following invariant holds.

LEMMA 9.2. *If some processor on bus J stores block B and any processor or gateway in T stores B, then the gateway also stores B.*

PROOF. We prove this by induction on the number of requests in σ that have been issued. Initially, the proposition holds since only one processor in the system stores B.

Suppose the proposition is true after $t-1$ requests in σ have been issued and satisfied. There are two cases:

I. Before step t only processors on one side of the gateway are storing block B, say in the tree T. If, after $\sigma(t)$, a processor on J stores B, then $\sigma(t)$ is a $\text{READ}_{q_J}(B)$ by some processor q_J on J. Since no other processor on J stores B, the request must be satisfied by having the gateway supply B (by getting it from T). When the gateway supplies B, B becomes shared on bus J and the gateway (as a member of the bus) also stores B ($Has[B] := 0$). Analogously, if only processors in J store B before $\sigma(t)$, and processors in T store B after, then the read request that caused this must have supplied B through the gateway g. As B becomes shared on the bus in T to which g

supplies B, and g is an ordinary participant on this bus, the g stores B after $\sigma(t)$.

II. Before $\sigma(t)$, processors on both sides of the gateway store B (and hence by hypothesis so does the gateway g). Suppose that, after $\sigma(t)$, the gateway no longer stores B. Then to the gateway g the last p writes to B were issued by a single processor (gateway) on one of the buses to which g is connected. But because the topology of the network is a tree and all p of these writes were forwarded to all other subtrees with g as a root (with the forwarding gateway to each bus identifying itself as the writer), B must be invalidated on all other subtrees of g. Therefore, in particular, if the writes originate in J, then no processor in T will store B after $\sigma(t)$. Similarly, if the writes originate in T, then no processor in J will store B after $\sigma(t)$.

Hence, our invariant is maintained. □

LEMMA 9.3. *With respect to the bus J, Algorithm mb simulates clustering $T \cup g$ into a single processor q, storing everything stored anywhere in $T \cup g$. Specifically,* $C_{mb}^{\mathcal{B}'}(\sigma', J) = C_{mb}^{\mathcal{B}}(\sigma, J)$.

PROOF. All we need to show is that the gateway issues a read request to bus J only if the block requested is stored only on bus J (not in $T \cup g$) and the gateway issues a write request to bus J only if the block B is stored both in T and in J. The latter is trivially true since the gateway only forwards a write request to bus J if B is shared on J (in Algorithm *bsc*). The former follows directly from Lemma 9.2, since if a processor q_T in T issues a $\text{READ}_{q_T}(B)$ and processors in T and J store B, then so does g ($Has[B] = 0$) and hence g would not forward the read request to J. □

LEMMA 9.4. *For any on-line or off-line algorithm A in the snoopy reading tree-of-buses model $\mathcal{B}$,*

$$C_{mb}^{\mathcal{B}}(\sigma, J) \leq 2 \cdot C_A^{\mathcal{B}}(\sigma, J) + k.$$

PROOF. $C_{mb}^{\mathcal{B}}(\sigma, J) = C_{mb}^{\mathcal{B}'}(\sigma', J) \leq 2 \cdot C_A^{\mathcal{B}'}(\sigma', J) \leq 2 \cdot C_A^{\mathcal{B}}(\sigma, J) + k$, where the equality follows from Lemma 9.3, the first inequality follows from Theorem 7.1, and the second inequality follows from Lemma 9.1. □

We may inductively extend these arguments to show that for an arbitrary tree of block snoopy caching buses (where buses are connected to one another by gateways), the traffic per bus is bounded by twice the traffic of any other on-line or off-line algorithm A on that bus. Hence, we have proven the following theorem.

THEOREM 9.1. *Let $\mathcal{M}$ be any multibus system for which it is possible to select a spanning tree of block snoopy caching buses for each block B. Then there is an on-line algorithm for processing* READ/WRITE *requests on $\mathcal{M}$ that, for any sequence and each bus, never incurs a cost more than twice that of any on-line or off-line algorithm that uses the same set of spanning trees.*

The simplest choice for the system $\mathcal{M}$ is a tree of buses. Unfortunately, trees have very low bandwidth. For this reason we propose the d-dimensional cube of buses, each connected to n gateways. Such a structure is called a *d-dimensional n-cube.* Attached to each gateway is a processor which uses the gateway's memory as a snoopy cache.

For each block B we select a spanning tree of buses that will be used for communicating about block B. By using different spanning trees for different blocks, we increase the effective bandwidth of the memory. We describe the spanning trees as follows: an edge is in dimension i if it is parallel to the ith coordinate axis. The children of an edge in dimension i are all edges in dimension $i+1$ that intersect it. If we choose any edge in the first dimension as a root, then the tree of its descendents under this relation is a suitable spanning tree. Thus, the selection of a spanning tree is reduced to the selection of an ordering of the dimensions and an edge in the first dimension. If the selection is sufficiently random, and shared write traffic is not too heavy, we can expect an $O(n^{d-1})$ speedup over a single bus, since when a gateway is not in use all the buses connected to it can operate concurrently.

COROLLARY 9.1. *There is an on-line algorithm for processing* READ/WRITE *requests on a block snoopy caching d-dimensional n-cube that, for any sequence and each bus, never incurs a cost more than twice that of any on-line or off-line algorithm that uses the same subtrees.*

10. Remarks. In this paper we have defined several snoopy caching models and devised competitive algorithms for them. We consider it surprising that most variations of snoopy caching have competitive algorithms with acceptably small constants. A worthy and tractable goal for future work is to devise competitive algorithms for other snoopy caching problems, and other scheduling problems.

A problem very closely related to those described in this paper is that of deciding where to keep files in a distributed file system. We believe that the ideas of this paper can be applied to obtain competitive algorithms for that problem as well. File system caching differs from snoopy caching in several ways. Files are not of fixed size, and file updates are usually performed at the level of blocks. Most networks (other than token rings) do not support unit-cost reliable broadcast.

We have developed skill at devising competitive snoopy caching algorithms and at choosing potential functions to analyze them. The potential functions for all of the preceding algorithms use the adversary state to select a linear function of the state variables of the competitive algorithm. In general, the coefficients of the potential function are found by considering the hard cases in the proof, and setting additive and multiplicative constants just large enough to cover the changes in state. We then check the remaining cases, and renormalize the potential function to minimize the additive constant. The only difficult step in this is determining which part of the adversary's state is relevant. We do not yet understand how to

automate this process. In particular, the selection of the potential function seems critically dependent on the way that we intend to account for write-back costs.

We have not yet transformed our skill into a general theory of competitiveness. Such a theory would ideally take a problem description and determine if there exists a strongly competitive algorithm for it, and would also evaluate the multiplicative constant. This is another very interesting area for future work.

The practical significance of these results is unclear. The only concrete evidence that competitive analysis is more meaningful than average case analysis was supplied by Bentley and McGeoch [BM]. They showed that the move-to-front heuristic (a competitive algorithm) for maintaining a list performs better than some other heuristics on sequences obtained from real data, while average case analysis predicts the opposite outcome. This anomaly is a result of the locality of reference exhibited by real programs. That same locality makes it likely that careful tuning of our algorithms to exploit locality will significantly improve them in practice.

Acknowledgments. We thank Bill Coates, Butler Lampson, Greg Nelson, and Chuck Thacker for their comments and critiques of the work in progress. Guy Jacobson, Albert Greenberg, and Marc Snir made valuable comments about the exposition, and Zary Segall started us thinking about snoopy caching. We thank DEC Systems Research Center for travel support. Finally, we are deeply indebted to Hania Gajewska for her assistance in writing and editing this paper.

Appendix A. The Architecture of the Bus. The results in this paper make various assumptions concerning the capabilities of the underlying architectures. These assumptions involve not only the cost of the actions, but also the ability to perform functions that are beyond the ability of many older, uniprocessor bus architectures. Our schemes have been presented in a general fashion to facilitate the analysis as well as providing the implementation architect much freedom. In this appendix we address some of our assumptions as well as a sketch of some implementation details.

A bus is simply a set of wires or lines. The caches and the main memory are electrically connected to these lines. The lines are logically grouped to provide various functionality. The groups that concern us are: (i) address, (ii) data, (iii) function, (iv) control, and (v) replication.

The meaning of the first two groups is self-evident. The function lines encode the type of operation. We are concerned with: read, write, read-block, and write-block. The control lines are used to request and receive permission to use the bus.

In conventional snoopy cache architectures the replication lines are set at the end of every bus transaction and indicate if the block that contains the address just referenced is currently replicated. Thus, each block in a cache can maintain a bit indicating if the block was replicated. Note that this bit may not always be correct. A block may have been replicated during the last bus access, but subsequently dropped from all other caches. Since drops can be free, a cache may

incorrectly think the block is replicated. Note, however, that this bit is conservative, since an indication of nonreplication is always correct.

In standard uniprocessor bus architectures each bus transaction has a single master and a single slave communicating over the bus. In a snoopy cache architecture there are many slaves. A master places data on the bus and many other slaves concurrently read this data. Architects must take care to ensure that the data remains on the bus until all the interested slaves have read it.

Not only are there many slaves that can read from the bus, but there may be many slaves that want to respond to a request on the bus. For example, a read function on the bus may require that the block be supplied by all caches that have a dirty copy or by the main memory if no cache has the correct data.

In addition, our schemes sometimes require that one of the caches replicating a block be chosen when that block is updated. Care is required to ensure that only one cache will modify its counter even though the block may be replicated in many caches. In Appendix B we present a solution that requires a small number of additional bus lines.

We are free to consider only the cost of communication, since standard techniques for using separate tag stores in the cache for the bus and for the processor can effectively reduce the contention for the cache, as seen by the processor.

Appendix B. Decentralized Implementations. In this section we present a complete description of a decentralized algorithm for limited block snoopy caching. We then exhibit a short suite of techniques that can be applied to decentralize the other algorithms in this paper.

For simplicity, we choose the version of limited block snoopy caching in which a cache hit for reading leaves the w values unchanged. In the description that follows we continue to assume that the problem of bus arbitration has already been solved, and that the actions of the caches are sequential. In an actual implementation the caches must be prepared for some other cache gaining control of the bus, suspending processing of that cache's action, and servicing the bus request.

Each processor i has a cache C_i, an array of pointers to records indexed by a hash function h_i of the referenced address. Each record has the following fields:

addr: the address of the block resident in this line, or the address of the block for which this line is reserved.

valid: a bit; true iff the block is resident. If the line is valid, the data stored in this line must be the correct data for block *addr*.

D: the data for block *addr*.

last: a bit; true iff processor i was the last processor to write to a location in block *addr*.

w: a value in the range $[0 \cdots p]$; 0 unless *last* is true.

We treat main memory as cache 0; main memory is different in that h_0 is the identity function, and main memory has no processors which can make requests.

Thus the values of *addr*, *last*, and w need not be stored, since they either are determined by the hash index or are constant.

The shared variables of the algorithm in Section 8 can be synthesized from these variables. Let $L = C_i[h_i(B)]$; $L \triangleright addr$ means dereference L, and select the *addr* field. Then

$$i \in Store[B] \equiv L \triangleright addr = B \wedge L \triangleright valid,$$
$$i \in Reserved[B] \equiv L \triangleright addr = B \wedge \neg L \triangleright valid,$$
$$i = last[B] \equiv L \triangleright addr = B \wedge L \triangleright last.$$

Determining the value of $w[B]$ is a little bit harder. If there is no i such that $i = last[B]$, then $w[B] = 0$. Otherwise, for i such that $i = last[B]$, $w[B] = C_i[h_i(B)] \triangleright w$. Since $w[B]$ is tested only by $i = last[B]$, this can be done locally.

The bus has address, data, and command lines. The legal commands are:

Fetch(B): request the data for block B. All caches storing block B broadcast each of the values in the data block, using the Value command described below.

Invalidate(B): cause all caches other than the one transmitting this request to drop B.

Value(B, v, d): report that the value of the vth variable in block B is now d. Simultaneous Value commands issued by different caches are legal if they coincide in all arguments. A cache receiving a Value command while not sending one ignores it if the block is not stored or reserved. Otherwise it accepts the value, marks line $h_i(B)$ as valid, and not last, and sets w to zero.

```
Algorithm Limited-Block-Snoopy-Caching;
  for t := 1 to length(σ) do
    if σ(t) = READ_i(v) then
      B_i := [v]; L_i := C_i[h_i(B_i)];
      if L_i ▷ addr ≠ B_i ∨ ¬(L_i ▷ valid) then Getblock(i, B_i, L_i) fi;
      Supply L_i ▷ D[v mod b] to processor i
    elsif σ(t) = WRITE_i(v) then
      { Suppose d is the value being written to v }
      B_i := [v]; L_i := C_i[h_i(B_i)];
      if L_i ▷ addr ≠ B_i ∨ ¬(L_i ▷ valid) then Getblock(i, B_i, L_i) fi;
      if ¬(L_i ▷ last[B]) ∨ L_i ▷ w < p then
        L_i ▷ last[B] := TRUE; L_i ▷ w := L_i ▷ w + 1;
        if L_i ▷ w < p then Value(B_i, v mod b, d)
        else Invalidate(B_i)
        fi
      fi;
      L_i ▷ D[v mod b] := d
    fi
  od
end Limited-Block-Snoopy-Caching;
```

Remember that when Value(B_i, v, d) is called, for every cache $j \neq i$ storing or reserving B_i, i.e., such that $L_j = C_j[h_j(B_i)]$ and $L_i \triangleright addr = B_i$, cache j sets $L_j \triangleright$ *valid* to TRUE, $L_j \triangleright last[B]$ to FALSE, $L_j \triangleright w$ to 0, and $L_j \triangleright D[v]$ to d.

```
Procedure Getblock(i, B, L);
  if L▷addr ≠ B ∧ L▷last[B] ∧ L▷w = p then
    for j := 0 ... b−1 do Value(L▷addr, j, l▷D[j]) od
  fi;
  L▷addr := B; Fetch(B);
  for j := 0 ... b−1 do
    { All k that had B valid at the start of the Fetch execute }
    Value(B, j, C_k[h_k(B)]▷D[j])
  od
end Getblock;
```

We can implement the variant of the algorithm in which a read sets w to zero by having each cache line remember if its processor has read any variable in its block since the line was last written to. The next time an update happens for that block, the caches can OR these bits, and broadcast the result back on an extra return line. This adds some complexity to the cache controller, but probably improves the performance. This completes the decentralized implementation of limited block snoopy caching.

We turn now to aspects of decentralized implementation of some of the other algorithms. The problem of selecting a unique victim may be handled by additional lines on the bus. Suppose that there are n caches. We give each cache a unique id in the range 1 to n, and allocate log n bus lines for broadcasting an id. When a block is replicated, all caches containing the block maintain a distributed linked list. Each cache has a pointer to the next cache that has a copy of the block. Care is required to maintain this list for each block. Precise details depend on the exact abilities of the model.

For example, we can decentralize the implementation of the direct-mapped snoopy caching algorithm *dsc* as follows. Each cache i maintains $w[i, B]$ values for each block B it currently stores, and a pointer $ptr[B]$ to the next cache which stores B. If $ptr[B] = i$, the block is unique to cache i.

On a write to a replicated block, cache i broadcasts the value j of $ptr[B]$. Cache j decrements $w[j, B]$, and, on the following bus cycle, broadcasts its id if $w[j, B] > 0$ and its value of $ptr[B]$ if $w[j, B] = 0$. The writing cache updates its value of $ptr[B]$ to the broadcast value.

Alternatively, this algorithm can be distributed by keeping track of replication conservatively, and using an arbitration scheme to elect a cache in which to decrement the w value.

Some of the schemes we have described may delay invalidation by a small constant number of bus cycles, k. In that case, if a processor writes to a several locations in a block in consecutive cycles, we may use more bus cycles than the centralized version of the algorithm. However, by making our bus arbiter use

some fair scheduling, a given cache will only get control of the bus in consecutive cycles if the bus would have been idle anyway. Thus, although the total bus utilization is increased, bus contention remains strongly competitive. Alternatively, if the scheduling is unfair, the competitive factor may suffer by k/p.

By combining the ideas used to decentralize *dsc* and *lbsc*, we can decentralize the other algorithms as well.

Appendix C. Proofs of Demand Paging Theorems

THEOREM C.1. *Algorithm fwf is strongly competitive, that is, for any input sequence* σ,

$$F_{fwf}(\sigma) \leq \left(\frac{n_{fwf}}{n_{fwf} - n_{min} + 1}\right) F_{min}(\sigma) + k,$$

where k depends only on the initial state of the caches and is zero if both sets of caches start out empty.

PROOF. We maintain an array a of binary variables. $a[P]$ is 1 if and only if page P is in the fast memory maintained by *fwf*.

Let $\rho = n_{fwf}/(n_{fwf} - n_{min} + 1)$, and let S_{min} be the set of pages the *min* algorithm has in fast memory after step t of σ. The potential function

$$\Phi(t) = \sum_{P \in S_{min}} (a[P] - \rho) - \sum_{P \notin S_{min}} a[P](\rho - 1)$$

is used to show that

$$\Delta F_{fwf}(\sigma, t) - (\rho)\Delta F_{min}(\sigma, t) \leq \Delta\Phi,$$

and hence prove the theorem.

Step t of σ is a read request for some page P. One of the following cases holds:

A. Page P is in fast memory for both *min* and *fwf*: $\Delta F_{min} = \Delta F_{fwf} = 0$.
Hence, we must show $\Delta\Phi \geq 0$.
Since no a values or state change, $\Delta\Phi = 0$.

B. *fwf* has a page fault: $\Delta F_{min} = 0$; $\Delta F_{fwf} = 1$.
Hence, we must show $\Delta\Phi \geq 1$.
$\Delta a[P] = 1$ and since *min* has no fault $P \in S_{min}$. If *fwf*'s cache is not full then no other a values change and $\Delta\Phi = 1$. If *fwf*'s cache is full, then everything but P gets flushed: at most $n_{min} - 1$ other pages in S_{min} have their a value decremented and at least $n_{fwf} - n_{min} + 1$ pages not in S_{min} have their a value decremented. Hence, $\Delta\Phi \geq 1 - (n_{min} - 1) + (n_{fwf} - n_{min} + 1)(\rho - 1) = 1$.

C. Both algorithms have a page fault: $\Delta F_{min} = \Delta F_{fwf} = 1$.
Hence, we must show $\Delta\Phi \geq 1 - \rho$.

We may think of these two faults as occurring sequentially, case D followed by case B.

D. *min* has a page fault, *fwf* is ignored: $\Delta F_{min} = 1$; $\Delta F_{fwf} = 0$.
Hence, we must show $\Delta\Phi \geq -\rho$.
P enters S_{min} so $\Delta\Phi = a[P] - \rho + a[P](\rho - 1) = (a[P] - 1)\rho \geq -\rho$.

If *min* drops page P from fast memory, then $\Delta\Phi = -a[P](\rho - 1) - (a[P] - \rho) = \rho(1 - a[P]) \geq 0$. □

We could stop at this point, since *fwf* is uniformly inferior to both *lru* and *fifo*, since the cache of *fwf* is always a subset of the other algorithms' caches. However, the potential function for *fwf* is sufficiently complex that we find it easier to use *fifo* in our proof for associative snoopy caching.

THEOREM C.2. *Algorithm fifo is strongly competitive, that is, for any sequence* σ,

$$F_{fifo}(\sigma) \leq \left(\frac{n_{fifo}}{n_{fifo} - n_{min} + 1}\right) F_{min}(\sigma) + k,$$

where k depends only on the initial state of the caches and is zero if the cache of min is initially empty.

PROOF. Consider the following implementation of the *fifo* strategy. For each page P, maintain an integer-valued variable $a[P]$ in the range $[0, n_{fifo}]$. $a[P] = 0$ if P is not in fast memory. When page P is read into fast memory, $a[P]$ is set to n_{fifo}, and and for all other pages P' in fast memory $a[P']$ is decremented. (The page whose new $a[P]$ value is 0 is the one replaced. This is the page that has been in the fast memory the longest.)

We use the potential function

$$\Phi(t) = \sum_{P \in S_{min}} \frac{a[P] - n_{fifo}}{n_{fifo} - n_{min} + 1}$$

to prove the theorem.

Step t of σ is a read request for some page P. One of the following cases holds:

A. Page P is in fast memory for both *min* and *fifo*: $\Delta F_{min} = \Delta F_{fifo} = 0$.
Hence, we must show $\Delta\Phi \geq 0$.
Since no a values or state change, $\Delta\Phi = 0$.

B. *fifo* has a page fault: $\Delta F_{min} = 0$; $\Delta F_{fifo} = 1$.
Hence, we must show $\Delta\Phi \geq 1$.
Since *min* has no fault $P \in S_{min}$. $\Delta a[P] = n_{fifo}$ and at most $n_{min} - 1$ other pages in S_{min} have their a value decremented. Hence,

$$\Delta\Phi \geq \frac{n_{fifo}}{n_{fifo} - n_{min} + 1} - \frac{(n_{min} - 1)}{n_{fifo} - n_{min} + 1} = 1.$$

C. A has a page fault: $\Delta F_{min} = 1$; $\Delta F_{fifo} = 0$.
Hence, we must show $\Delta\Phi \geq -n_{fifo}/(n_{fifo} - n_{min} + 1)$.
P enters S_{min} whence

$$\Delta\Phi \geq \frac{a[P] - n_{fifo}}{n_{fifo} - n_{min} + 1} \geq -\frac{n_{fifo}}{n_{fifo} - n_{min} + 1}.$$

D. Both algorithms have a page fault: $\Delta F_{min} = \Delta F_{fifo} = 1$.
Hence, we must show $\Delta\Phi \geq 1 - n_{fifo}/(n_{fifo} - n_{min} + 1)$.
We may think of these two faults as occurring sequentially, case C followed by case B.

If *min* drops page P from fast memory, then

$$\Delta\Phi = -a[P]\left(\frac{n_{fifo}}{n_{fifo} - n_{min} + 1} - 1\right) - \left(a[P] - \frac{n_{fifo}}{n_{fifo} - n_{min} + 1}\right)$$

$$= \frac{n_{fifo}}{n_{fifo} - n_{min} + 1}(1 - a[P]) \geq 0. \qquad \square$$

Similarly, we can prove

THEOREM C.3. *Algorithm lru is strongly competitive, that is, for any sequence* σ,

$$F_{lru}(\sigma) \leq \left(\frac{n_{lru}}{n_{lru} - n_{min} + 1}\right) F_{min}(\sigma) + k,$$

where k depends only on the initial state of the caches and is zero if the cache of min is initially empty.

PROOF SKETCH. The same potential function as that used in Theorem A.2 can be used to prove this theorem. $\square$

It may appear counterintuitive that an algorithm as crude as *fwf* can be strongly competitive. The reason is simple: bad sequences for demand paging algorithms exhibit almost no locality of reference. Competitive analysis yields the tightest information about sequences on which the algorithm is particularly well fooled. Therefore, competitive analysis provides a tool for rejecting algorithms. Given a choice between several strongly competitive algorithms, however, one must use other criteria such as average case analysis. In the case of *fwf* the simplicity of the algorithm (and its adaptability to caching situations with variable-sized blocks) may make it the algorithm of choice in some situations.

Appendix D. Proof of Competitiveness of Block Snoopy Caching Model

THEOREM D.1. *Algorithm bsc is strongly 2-competitive, that is, for any sequence* σ *and any on-line or off-line block-retention algorithm A in the block snoopy caching model,*

$$C_{bsc}(\sigma) \leq 2 \cdot C_A(\sigma) + k,$$

where k is a constant that depends on the relative initial cache states of bsc and A. If every block is initially unique to its last writer then $k = 0$.

PROOF. As usual we let τ denote the labeled, merged sequence of actions taken by *bsc* and *A*. At time t, L is the set of blocks B which A stores only in cache $last[B]$, and S is the set of blocks B which A stores in some cache other than $last[B]$. In this model, S is the complement of L, since every block is in some cache. The potential function

$$\Phi(t) = \sum_{B \in S} (-w[B] - p) + \sum_{B \in L} (w[B] - p)$$

is used to prove the theorem.

If step t of τ is an action labeled only with A, then one of the following cases holds:

A. The action is *Fetchblock*(i, B):
$\Delta C_A = p$ and so we must show $\Delta\Phi \geq -2p$. Before the action B may or may not be in L, and after the action $B \in S$. In either case, $\Delta\Phi \geq -w[B] - p - (w[B] - p) = -2w[B] \geq -2p$.
B. The action is *Drop*(i, B):
$\Delta C_A = 0$ and so we must show $\Delta\Phi \geq 0$. If $B \in S$ before and after the drop, then $\Delta\Phi = 0$. Otherwise $B \in L$ after the drop and $\Delta\Phi = w[B] - p - (-w[B] - p) = 2w[B] \geq 0$.

If step t of τ is an action labeled only with *bsc*, then one of the following cases holds:

A. The action is *Fetchblock*(i, B):
$\Delta C_{bsc} = p$ and so we must show $\Delta\Phi \geq p$. Since $\Delta C_A = 0$ and $last[B] \neq i$, A has $B \in S$. Hence, since $\Delta w[B] = -p$, $\Delta\Phi = p$.
B. The action is *Drop*(j, B), and was caused by a write to a replicated block:
Since $\Delta w[B] = 0$, and A's state does not change, $\Delta\Phi = 0$.

If step t of τ is an action labeled with both A and *bsc* then one of the following cases holds:

A. The action is *Supply*(i, v):
$\Delta C_A = \Delta C_{bsc} = 0$ and so we must show $\Delta\Phi \geq 0$. If $i \neq last[B]$ then $B \in S$ and $\Delta w[B] \leq 0$. Hence $\Delta\Phi = -\Delta w[B] \geq 0$. Otherwise $i = last[B]$ $(B \in L)$ and $\Delta w[B] = 0$, whence $\Delta\Phi = 0$.
B. The action is *Updatelocal*(i, v):
$\Delta C_{bsc} = 0$ and so we must show $\Delta\Phi \geq -2\Delta C_A$. Since $\Delta w[B] = 0$ and A does not change state, $\Delta\Phi = 0 \geq -2\Delta C_A$.
C. The action is *Updateglobal*(i, v). There are two subcases depending on whether or not block B is unique to A:
C1. B is replicated in *bsc* and is unique in A:
$\Delta C_A = 0$, $\Delta C_{bsc} = 1$ and so we must show $\Delta\Phi \geq 1$. There are three places in the algorithm (labeled {1}, {2}, and {3}) which might have issued this

Update. In case {2} $w[B]$ is set to 1, and $last[B]$ is changed. This change in $last[B]$ changes the sets S and L such that $B \in S$ before the action and $B \in L$ after it. The potential associated with B after the request is $1-p$, and before the request it is $-w-p$ (where w denotes $w[B]$ before the request). So, $\Delta\Phi = w+1 \geq 1$. In cases {1} and {3}, B starts in either L or S and ends up in L, and then $w[B]$ is incremented. The change in the state of B does not decrease the potential, and increasing $w[B]$ increases it by 1.

C2. B is replicated in *bsc* and in A:
$\Delta C_A = \Delta C_{bsc} = 1$ and so we must show $\Delta\Phi \geq -1$. Since the cost of the action is 1 to A, and the block is in cache i, A must be keeping the block replicated. This means that even though $last[B]$ may change, $B \in S$ both before and after this action. In case {2} the change in the potential is $(-1-p)-(-w-p) = w-1 \geq -1$. In cases {1} and {3} $w[B]$ increases by 1, so $\Delta\Phi = -1$. □

References

[AB] Archibald, J., and Baer, J.-L. An evaluation of cache coherence solutions in shared-bus multiprocessors. Technical Report 85-10-05, Department of Computer Science, University of Washington, 1985.

[B] Belady, L. A. A study of replacement algorithms for virtual storage computers. *IBM Systems J.*, **5** 78-101, 1966.

[BM] Bentley, J. L., and McGeoch, C. C. Amortized analysis of self-organizing sequential search heuristics. *Comm. ACM*, **28**(4) 404-411, 1985.

[F] Frank, S. J. Tightly coupled multiprocessor system speeds memory access times. *Electronics*, **57**(1), 164-169, 1984.

[G] Goodman, J. R. Using cache memory to reduce processor-memory traffic. *Proc. 10th Annual IEEE International Symposium on Computer Architecture*, pp. 124-131, 1983.

[KEW] Katz, R., Eggers, S., Wood, D. A., Perkins, C., and Sheldon, R. G. Implementing a cache consistency protocol. *Proc. 12th Annual IEEE International Symposium on Computer Architecture*, pp. 276-283, 1985.

[PP] Papamarcos, M., and Patel, J. A low overhead coherence solution for multiprocessors with private cache memories. *Proc. 11th Annual IEEE International Symposium on Computer Architecture*, pp. 348-354, 1984.

[RS1] Rudolph, L., and Segall, Z. Dynamic decentralized cache schemes for MIMD parallel processors. *Proc. 11th Annual IEEE International Symposium on Computer Architecture*, pp. 340-347, 1984.

[RS2] Rudolph, L., and Segall, Z. Dynamic paging schemes for MIMD parallel processors. Technical Report, Computer Science Department, Carnegie-Mellon University, 1986.

[ST] Sleator, D. D., and Tarjan, R. E. Amortized efficiency of list update and paging rules. *Comm. ACM*, **28**(2), 202-208, 1985.

[VH] Vernon, M. K., and Holliday, M. A. Performance analysis of multiprocessor cache consistency protocols using generalized timed Petri nets. Technical Report, Computer Science Department, University of Wisconsin, 1986.