LOWER BOUNDS ON LATENCY FOR SCALABLE LINKED-LIST CACHE COHERENCE

by

Ross Johnson

Computer Sciences Technical Report #1029

June 1991

Lower Bounds on Latency for Scalable Linked-List Cache Coherence

Ross Johnson
ross@cs.wisc.edu
University of Wisconsin - Madison

Abstract

Cache-coherence protocols are used in multiprocessors to maintain a programming view of a single shared memory. Some scalable cache-coherence protocols, like the Scalable Coherent Interface (proposed IEEE-P1596 standard), use linked lists of cache lines to maintain hardware cache coherence and prevent system-wide broadcasts. As processors read and write data, the cache-coherence protocols create lists, distribute data, and destroy lists of cache lines. It is important that these list operations are efficient as the system size (number of processors) increases. Therefore, additional pointers are added to the lists to make trees that are efficient for the two multicasts (data and invalidation) required by the protocols.

In this paper we determine lower bounds on the latency of reads and writes in systems with cache-coherence protocols that use linked lists. This work is particularly relevant to the extensions for the proposed IEEE-P1596 standard. First, we show that the potential advantages of additional pointers decreases rapidly as more pointers per list element are added. Second, we show that using the same pointers for the two multicasts necessarily increases the latency of each multicast. Third, we give an algorithm for creating pointers and show that the time to create additional pointers with this algorithm is a significant part of one multicast latency (for data). Finally, since theoretically optimal solutions are not feasible for implementations, we propose one implementable tree and demonstrate that its use results in latency that is only twice that of a lower bound.

1. Introduction

One of the goals of multiprocessing is to keep a large number of processors busy doing useful work. The goal of the memory system is to manage reads and writes so that processor wait-times are short and infrequent. This is becoming more difficult since processor speeds are currently increasing faster than memory and interconnect speeds. In shared-memory multiprocessors, memory is cached (duplicated) near the processors in order to reduce memory access times and interconnect congestion [Good83]. Many implementations also cache multiple copies of shared data for simultaneous reads.

As processors read and write data in a shared-memory multiprocessor, it is important to maintain a unified view of main memory. When one copy of shared data is modified, all other copies of that data (if any) must be *invalidated* (or updated)[1]. This is the problem of *cache coherence*. Specifically, all caches with current copies are told to invalidate (mark or destroy) their copies. After the data has been invalidated, caches will request new data as needed.

Broadcasting of invalidations to all caches in the system is infeasible for large multiprocessors because the bandwidth required by this solution grows faster than the bandwidth provided by any cost-effective interconnect. Interconnect congestion, caused by broadcasts, stalls the processors, effectively reducing the total computing power of the system. Instead, *multicasting* is required, where only the necessary subset of the system's caches are notified of an invalidation. Note that since single buses and their associated snooping protocols [ArBa86] are broadcast oriented, they are infeasible for large multiprocessors.

In order to multicast an invalidation, the set of destinations must be known. Copies of data in caches are organized into *cache lines*, about 64 bytes each. *Directories*, one for each equivalent set of cache lines, remember the destination sets. Each directory must reside in a well-known location (a function of the memory address to which it corresponds) so that it can be accessed. Spreading the directories throughout the system (often with memory) avoids hot-spots [PfNo85]. Some have proposed storing all elements of one destination set in one location [ASHH88]. However, hardware constraints limit the set size for each directory to a constant number of destinations, requiring special overflow mechanisms. These mechanisms either default to broadcast invalidation or limit the number of simultaneous reads.

One way to avoid overflow complications is to further spread-out the directories, changing the destination set into a linked list of cache lines. Then, one hardware-maintained linked list of cache lines is kept for each directory. This is part of the proposed IEEE-P1596 standard. Each cache line in the system is in at most one list and each cache may be in several lists, one for each line. In addition to data, each cache line stores part of the directory, *pointers* to the next and previous caches in the list. In particular, these pointers are called *list pointers* and no cache line needs to store more than two. In order to provide

[1]Providing a global ordering on data modifications, called sequential consistency [AdHi90, DuSB86, Lamp79], is also important, but it is not an explicit issue for this paper.

a well-known location for the linked-list directory, memory (or another fixed-location entity) keeps a pointer to one end of the list. Using linked lists to maintain cache coherence is called *linked-list cache coherence* and this is the kind of protocols we are studying.

Recall that invalidation of cache lines is a result of processor reads and writes. When processors read data, copies are cached and lists of cache lines are built. When processors write data, copies are invalidated and lists are broken-down. Before discussing the problem of making reads and writes efficient for large lists, we give more detailed examples of linked-list cache coherence in the Scalable Coherent Interface (SCI) [JLGS90]. SCI is a proposed IEEE standard (IEEE-P1596) for interconnects of shared-memory multiprocessors, including cache coherence.

When a processor tries to read data that is not in its cache, the cache must get a current copy. This is done in two request-response transactions, as shown in figure 1a. First, the cache that needs the data consults the memory to obtain the end of the list where a current copy of the data resides. Before responding, the memory updates its pointer to point to the requesting cache. Second, the cache requests the data via the returned pointer and waits for a response, storing the pointer until needed later. The last cache in the list to get the data is called the *head.*

When a processor tries to write data, its cache must first *purge* the list (invalidate cached copies and break-down the list). Otherwise, some processors may read new copies while others are reading old copies. This causes some programs to behave incorrectly. If the cache line is not the head of the list, it must become the head by consulting the memory (removing itself first if it is already in the list). Then, it can purge the other cache lines one by one, as shown in figure 1b. The head sends a purge request to the second cache. The second cache invalidates the appropriate cache line and responds with a pointer to the next cache, effectively removing itself from the list. Then, the head can purge the next cache, etc. When the list is completely purged, the head can modify the data. In some cases the data modification may occur before the list is completely purged [AdHi90, DuSB86]. Note that as processors again read the data, the most recent writer becomes the end of the list that is opposite the head. This end is called the

Examples of Linked-List Cache-Coherence Operations

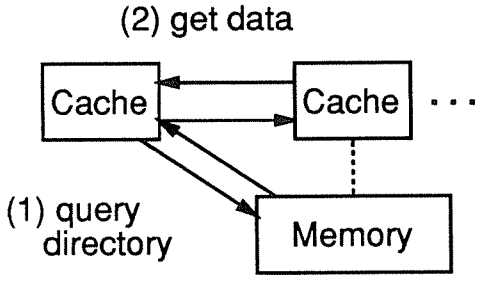


Figure 1a

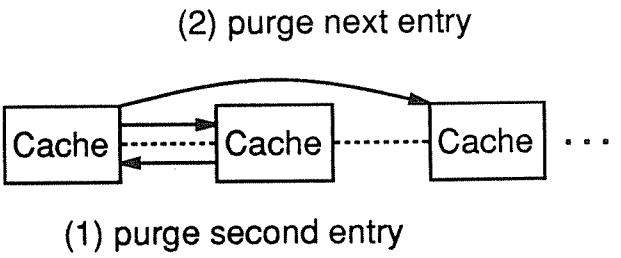


Figure 1b

tail.

To summarize, caches in shared-memory multiprocessors are needed for efficient reads and writes. Hardware cache coherence can be maintained with linked lists that prevent broadcasting.

1.1. Problem Description

Processor speeds have increased dramatically in the last few years, but interconnect latencies have remained relatively constant. Since this trend is expected to continue, the latency of a message between caches is modeled as two interconnect delays, one for sending the request and one for returning the response. In particular, *latency* of cache coherence protocols is measured by the number of messages on the critical path, ignoring processing speed. *Traffic* is measured by the total number of messages.

Consider a shared variable that is read often by many processors. Each time the variable is written, the associated list is purged. When the processors read the new value of the variable, the list is created and the data is distributed. For the coherence protocol to be efficient, these three operations must be efficient. For lists of size N, the defined cache coherence protocol has $O(N)$ traffic and $O(N)$ latency. Although the traffic is acceptable (it is impossible to do better), the latency is not. We can do much better. This is the problem of *scalable*[2] *linked-list cache coherence* (SLC) and it is particularly relevant to the extensions [IEEE90] for the proposed IEEE-P1596 standard.

The key idea [JLGS90] is to add a small constant number of *temporary pointers* to the lists. Shown in figure 2, temporary pointers increase the connectivity of the lists, making them more like trees. Using temporary pointers, protocols can distribute data and purge lists with sublinear latency, often $O(\log(N))$. Of course, the choice of which temporary pointers to add affects these two latencies as well as the latency to create the temporary pointers.

The latency of SLC protocols is the sum of the latencies for three operations: list creation, data distribution, and list purging. However, we are only interested in part of this latency: temporary pointer creation and two multicasts (data and invalidations). In particular we assume that the list has already been created and that each cache knows its position in the list (distance from the last writer). These

Example of Temporary Pointers

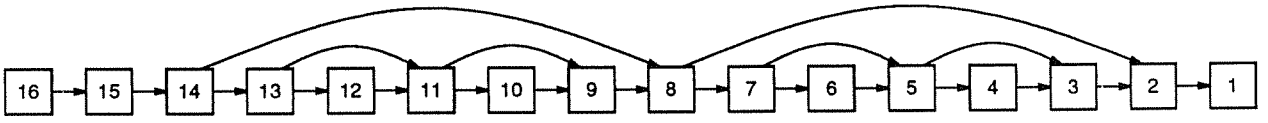


Figure 2

[2]For this paper, scalable intuitively means (with respect to system size) sublinear latencies (usually logarithmic), linear traffic that is uniformly distributed, and nearly constant space per processor (except normal memory).

assumptions are reasonable in that efficient preliminary solutions have been developed for list creation [JLGS90] and position numbering [IEEE90]. Furthermore, we are not interested in the latency of breaking-down the list, only the latency of propagating the signal to do so (invalidation multicasting). Efficient preliminary solutions have also been developed for efficient list destruction [IEEE90]. We leave ourselves with the problem of deciding which temporary pointers to create and how to create and use them. We want to minimize the latency for their creation and their use for two multicasts, while limiting the traffic to $O(N)$.

In this paper we give a number of lower bounds on latency for temporary pointer creation and two associated multicasts. Lower bounds on this problem are important in that they provide a standard against which proposed implementations can be measured. Furthermore, it does not suffice to determine complexity orders since constants affect which solutions are candidates for hardware implementations. A study of lower bounds also yields insight into what characteristics make solutions good and bad.

In studying lower bounds, we have discovered some important results. First, we show that the potential advantages of additional temporary pointers decreases rapidly as more pointers per cache line are added. This is reassuring since hardware limitations probably restrict the number of temporary pointers to one or two per cache line. Second, we show that using the same temporary pointers for data distribution and invalidation necessarily increases the latency of both, assuming one temporary pointer per cache line. Third, we give an algorithm for creating temporary pointers and show that creation of temporary pointers contributes significantly to latency and can not be easily overlapped with data distribution. Finally, since control complexity in hardware is an issue, we propose an implementable choice of temporary pointers that yields about twice a lower bound on latency.

1.2. Related Work

Scalable linked-list cache coherence (SLC) with temporary pointers is fundamentally different than most communication problems in that no cache line knows all of the desired destinations for a multicast. Cache lines are allowed to change which destinations they know, effectively changing the logical communication topology. Furthermore, no cache line can remember more than a constant number of destinations at any one time (probably less than five) and message sizes are limited to a constant size[3]. Broadcasting variants of the gossiping problem [HeHL88] are not multicasting and do not allow modification of the communication topology. For multicasting in networks [Deer88, FrWB85], including the Steiner problem in networks [Wint87], algorithms either create new nodes or make use of nodes that are not destination set. This is not allowed by SLC.

The skip lists of Pugh [Pugh90] are similar to our cache lists in that additional pointers are added to increase list connectivity. However, some nodes of Pugh's lists may have up to a logarithmic number of

[3]Maximum packet sizes for coherent data in the Scalable Coherent Interface are one cache line (64 bytes) plus a header.

pointers per node. This is not appropriate for SLC since current constraints for hardware implementations would limit the number of additional pointers to one or two.

The algorithm outlined by Hillis and Steele [HiSt86] for finding the end of a linked list has logarithmic latency, but requires $O(N \log(N))$ traffic. A refinement of this algorithm, that creates temporary pointers and requires only $O(N)$ traffic, is described in the next section.

A number of groups are currently exploring linked-list cache coherence [TDLS90]. The Scalable Coherent Interface (SCI) [JLGS90] uses the protocols described earlier (see figure 1). Preliminary solutions for scalable extensions to SCI [IEEE90] are not optimal and no analysis of latency is yet offered. Gjessing et al [GJKM90] are studying different connectivities for list purging in SCI. However, their analysis tries to balance latency with traffic, whereas our analysis gives priority to latency. SCI is based on a doubly linked list of cache lines for easier error recovery and more efficient cache line replacement, discussed in the references.

The Stanford Distributed-Directory Protocol (SDD) [ThDe90] is based on a singly linked list of cache lines and makes no attempt to achieve scalable performance. It reduces the number of messages from four to three by having the memory forward requests for data (see figure 1). However, the four message protocol is also included to avoid special deadlock situations related to finite queues. SDD also uses forwarding for invalidation, thereby reducing the number of messages to about half, but they give no description of deadlock avoidance for invalidations in the cited reference.

Throughout this paper we give lower bounds on latency for a number of constraints. In most cases we solve the equivalent problem of maximizing the number of cache lines for a given latency instead of minimizing the latency for a given number of cache lines. In section 2 we describe the the mechanism for creating temporary pointers. In section 3 we give lower bounds on the latency for one multicast and determine that the number of temporary pointers per cache line need not be large. In section 4 we give lower bounds on the latency for both multicasts (data and invalidation), given that they must use the same temporary pointers. In section 5 we give lower bounds on the latency of the first multicast (data), including the latency for creating temporary pointers. In section 6 we compare the lower bounds of sections 4 and 5 to one solution that is reasonable to implement. Finally, we conclude with a summary of the results in section 7. The appendix contains additional equations, graphs, secondary text, and some of the longer arguments.

2. Recursive Doubling

The object of *recursive doubling* is to efficiently create temporary pointers that will be useful for efficient distribution of data. First, each cache line determines the position number of the line to which it wants a temporary pointer, if any. This is a function f of the system size M and the line's own position number x such that temporary pointers point *downstream* (towards the tail) and pointers are nested. Formally, $x \geq f_M(x)$ and if $y > x > f_M(y)$ then $f_M(x) > f_M(y)$. It remains to show how to create the temporary pointers for some function f_M. For all cache lines u such that $u \neq f_M(u)$, line u requests a pointer to line

$f_M(u)$ from its forward neighbor, line $u-1$. When a line $v=f_M(v)$ receives a request for a pointer to $v-1$, it returns pointer $v-1$ to the requester (figure 3a). Otherwise, line v forwards the request to $v-1$ (figure 3b). When line u receives a request while waiting (figure 3c), it delays the request in its constant storage until it receives the pointer to $f_M(u)$. Finally, u forwards the delayed request to $f_M(u)$.

After the temporary pointers are created, the data and invalidations can be efficiently multicasted. If a cache line has a temporary pointer, then it receives the data from line $f_M(u)$ and later forwards invalidations to that line and line $u-1$. Otherwise, it requests data and forwards invalidations with respect to line $v-1$. Note that line $v=f_M(u)-1$ requests the data from line $f_M(u)$ on behalf of line u. This prevents an extra message delay.

Since pointers are nested, it can be shown by induction on $x-f_M(x)$ that all cache lines will eventually receive their desired pointers. Furthermore, pointer nesting along with delayed forwarding guarantees that no line will receive more than one request for a pointer. It can also be shown that no line will need to respond to more that two requests for data and no line will receive more than two invalidations. Therefore, traffic is $O(N)$.

This algorithm can work with the request-response paradigm as well. When cache line x receives a request that would normally be forwarded, it returns the forwarding pointer[4], namely $\min\{x-1, f_M(x)\}$. Then if required, the requester simulates the forwarding by sending a new request via the forwarding pointer. Data propagates with request-response as well, but invalidations must revert back to the sequential method. Algorithm termination and constant space requirements are still satisfied and latency is no more than doubled, except for invalidations.

3. Fundamental Constraints

In this section we analyze the effectiveness of multiple temporary pointers per cache line in doubly linked lists and show that the latency of one multicast is proportional to $1 / \log(1+p\pm1)$, where p is the maximum number of temporary pointers per cache line. In other words, we show that the advantage of

Examples of Recursive Doubling

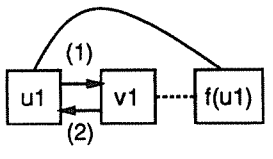


Figure 3a

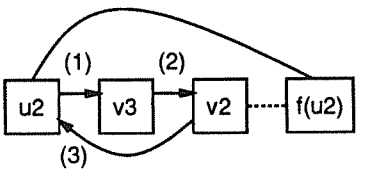


Figure 3b

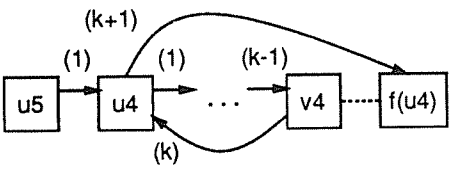


Figure 3c

[4]The response is delayed, if necessary, until the pointer is known.

additional temporary pointers decreases rapidly as more pointers per cache line are added, shown in figure 4. Since each additional temporary pointer is costly (about 16 bits per cache line in a system with 64K caches), it is doubtful that implementations will allow more than one or two. We also analyze the disadvantages of unidirectional information flow (information propagates away from the originator) and bidirectional temporary pointers (two pointers simulate an undirected edge in the graph). We show that these two constraints are less important than the number of temporary pointers (see figure 4). Note that lower bounds for doubly linked lists are also lower bounds on singly linked lists because the extra link can be simulated with an additional temporary pointer. It should also be noted that in this section we are only considering one multicast in isolation.

The equations in the appendix relate to *circular* linked lists, where the ends of the list have pointers to each other. These results show that the difference in latency between circular and non-circular lists is negligible for large doubly linked lists.

3.1. General Bound

The following equation bounds the maximum number of cache lines in a non-circular list. This result bounds all other results for non-circular lists. Based on this equation we derive a lower bound on latency for one multicast with one temporary pointer,

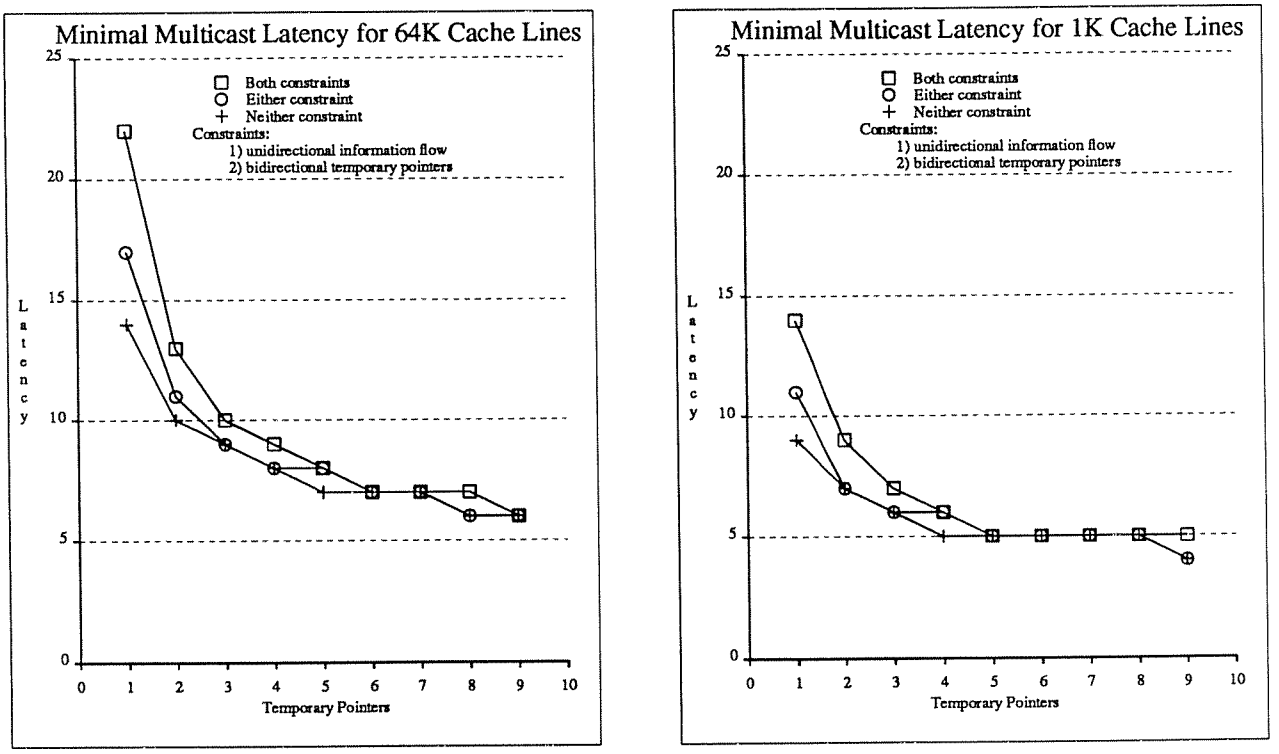


Figure 4

$$h(N) = 0.7864 \log_2(N) + 0.5729 \pm \frac{0.6160}{N - 0.5429}, \quad N \geq 1.$$

Equation 3.1 The maximum number of cache lines contained in a non-circular doubly linked list is

$$N(h,p) = K_1 \left[\frac{p+1+\sqrt{D_1}}{2} \right]^h - \frac{1}{2p} + K_2 \left[\frac{p+1-\sqrt{D_1}}{2} \right]^h \approx K_1 \left[\frac{p+1+\sqrt{D_1}}{2} \right]^h - \frac{1}{2p},$$

$$K_1 = \frac{D_1 + (3p+1)\sqrt{D_1}}{4p D_1}, \quad K_2 = \frac{D_1 - (3p+1)\sqrt{D_1}}{4p D_1}, \quad D_1 = p^2 + 6p + 1,$$

where each cache line has at most $p \geq 1$ temporary pointers and $h \geq 0$ is the height of the shortest-path spanning tree that is rooted at one end of the list.

Arguments: We can define two interdependent recurrence relations by identifying the two types of cache lines in a spanning tree and noting their maximum branching factors. Note that temporary pointers are unidirectional and list pointers come in pairs. See figure 5. If a line is connected to its parent by a temporary pointer, then the line can have $p+2$ children: p by temporary pointers and 2 by list pointers. However, if a line is connected to its parent by a list pointer, then the line can have only $p+1$ children because its second list pointer connects to the parent. This leads us to the following recurrence relations, where h is the height of a subtree and $T(h)$ and $L(h)$ are the maximum number of lines in a subtree whose root is visited by a temporary pointer and a list pointer respectively.

Maximum Lines for One Temporary Pointer

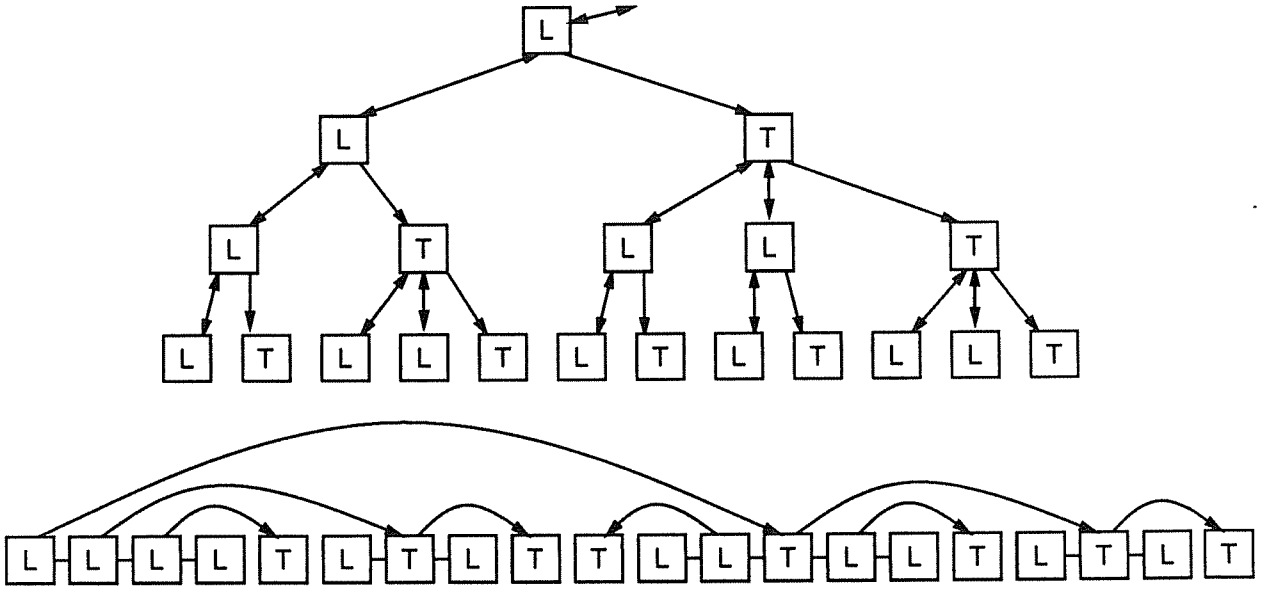


Figure 5

$$T(h) = 1 + p\,T(h-1) + 2\,L(h-1),\ T(1) = 1,\ T(0) = 0,$$
$$L(h) = 1 + p\,T(h-1) + L(h-1),\ L(1) = 1,\ \text{and}\ L(0) = 0.$$

$L(h)$ is the desired function because the end of the list can have only one list pointer. Standard methods [PuBr85] yield the result to be proved. Note that the contribution of one root becomes negligible for large h. □

3.2. Additional Constraints

Here, we examine two constraints, separately and collectively. One constraint is that temporary pointers are bidirectional, meaning that two temporary pointers simulate an undirected edge in the graph. This constraint may help in the process of creating the temporary pointers for algorithms not discussed in this paper. It may also prevent drastic loss of performance due to cache line removals[5]. The other constraint is that information flow is unidirectional, meaning that information propagates away from the originator, as determined by the list ordering. This constraint will help reduce the amount of traffic because the number of outgoing messages is reduced for some cache lines. It may also allow implementations with singly linked lists. The analysis shows that these constraints have less effect, separately and collectively, than changing the number of temporary pointers by one (see figure 4).

Equation 3.2 If *either* temporary pointers are bidirectional *or* information flow is unidirectional, then the maximum number of cache lines contained in a non-circular doubly linked list is

$$N(h,p) = \frac{1}{p}(p+1)^h - \frac{1}{p}$$

where p and h are as given in equation 3.1.

Arguments: We can derive the formula for $N(h,p)$ by showing that the maximum branching factor is $p+1$ for all non-leaf cache lines in the spanning tree. □

Equation 3.3 If temporary pointers are bidirectional *and* information flow is unidirectional, then the maximum number of cache lines contained in a *non-circular* doubly linked list is

$$N(h,p) = K_7\left[\frac{p+\sqrt{D_2}}{2}\right]^h - \frac{2}{p} + K_8\left[\frac{p-\sqrt{D_2}}{2}\right]^h \approx K_7\left[\frac{p+\sqrt{D_2}}{2}\right]^h - \frac{2}{p},$$

$$K_7 = \frac{D_2 + 2\sqrt{D_2}}{p\,D_2},\ K_8 = \frac{D_2 - 2\sqrt{D_2}}{p\,D_2},\ D_2 = p^2 + 4,$$

where p and h are as given in equation 3.1.

Arguments: Similar to equation 3.1 and given in the appendix. □

Since the equations of this section are related by

[5]Cache lines may be victimized by cache replacement algorithms, causing them to be removed from the list. We assume that these removals do not occur often.

$$p < \frac{p+1+\sqrt{D_1}}{2} < p+1 < \frac{p+\sqrt{D_2}}{2} < p+2,$$

we make two conclusions. First, the latency of one multicast is proportional to $1 / \log(1+p\pm1)$. Second, the issues of unidirectional information flow and bidirectional temporary pointers are less important than the number of temporary pointers.

4. Dual Constraints

In this section we study the interaction between the flows of data and invalidations, while ignoring the time to create the temporary pointers. We give the optimal set of temporary pointers, assuming that the worst-case latencies is when the list size N equals the system size M. Since these assumptions are more permissive than SLC, the analysis gives a lower bound on latency for SLC.

There is a fundamental difference between the two flows of information. Data flow is demand driven because reads in SLC introduce cache lines into the list. On the other hand, invalidation flow is knowledge driven because the time of a future write is unknown. This means that data will propagate as responses to requests, but invalidations will simply propagate as requests. Therefore, pointers used for data flow are stored in cache lines that *need* the information, but pointers used for invalidation flow are stored in cache lines that *have* the information.

In keeping with the assumptions of equation 3.2, we assume that data flow is unidirectional, meaning that all requests must be sent and forwarded downstream (towards the tail). We also assume that invalidation flow is unidirectional and that the same temporary pointers are used for data and invalidations.

Equation 4.1 Given two spanning trees of a non-circular doubly linked list with only one temporary pointer per cache line such that (1) each tree uses only one direction of the list pointers and (2) the trees are rooted at opposite ends of the list, *then* the maximum number of cache lines contained in the list is $T(l,d)$:

$$T(0,0) = T(0,d) = T(l,0) = 0,$$

$$T(l,d) = \max_{0 \le j < \min\{l,d\}} \left\{ 1 + j + \sum_{i=0}^{j-1} T(l-i-1, d-j+i) \right\}$$

where $l \ge 1$ and $d \ge 1$ are the heights of the spanning trees.

Arguments: Given in the appendix. □

Note that we want to know the maximal number of cache lines N for a given sum h of the two multicasts latencies in SLC. Using regression analysis we can show that

$$N(h) = \max\{T(l,d): l+d = h\} \approx 1.1\,(1.3)^h.$$

Comparing this to equation 3.1 for $p = 1$ we conclude that using the same temporary pointers for both multicasts necessarily increases the latency of both.

5. Recursive Doubling Constraints

In this section we give the lower bound on the latency for simultaneously creating the pointers with recursive doubling and distributing the data. We conclude that the latency of recursive doubling contributes significantly to the latency of reads and can not be easily overlapped with data distribution. This conclusion follows by comparing the following result with equation 3.1, where $N(h)=\Theta(2^h)$ for $p=1$.

Equation 5.2 Assuming at most one temporary pointer per cache line and that information flow is unidirectional, the maximum number of cache lines to which information can be distributed in $h \geq 1$ message delays using recursive doubling is

$$N(h) = 2.9460\,(1.3247)^h - 4$$
$$+ (0.0270+0.1184\,\mathrm{i})\,(-0.6624-0.5623\,\mathrm{i})^h + (0.0270-0.1184\,\mathrm{i})\,(-0.6624+0.5623\,\mathrm{i})^h,$$

where $i=\sqrt{-1}$.

Arguments: Formal arguments are given in the appendix. It suffices to say that the exact formula in the appendix is verified by exhaustive search for $0 \leq N \leq 73$ $(1 \leq h < 12)$. Therefore, $N(h)$ is correct as stated. Note that the contributions of two roots become negligible for large h. □

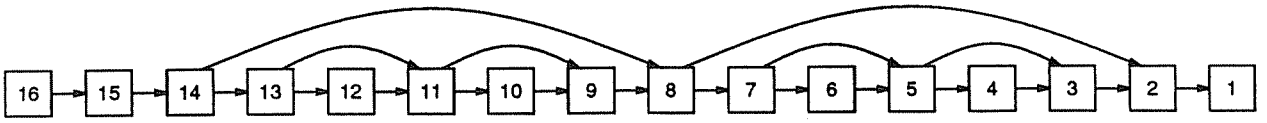
6. Simple Solution

In the section 2 we described recursive doubling, a simple mechanism for constructing temporary pointers. This mechanism assumes that a cache controller can easily compute the list position of the cache line to which it needs a temporary pointer, given its own position x and the maximum list size M. Functions for near-optimal solutions (often recursive functions) are probably not appropriate for hardware implementation, due to their complexity. In this section we give a function that is appropriate for hardware implementation and useful for efficient pointer creation and multicasts.

Figure 6 gives a function (written in C and tested for sizes 32 and 64) that returns the distance of the desired temporary pointer. One subtraction (from its position x) then yields the function required by recursive doubling. The figure also shows the pointer structure (same as figure 2) that would then be created by recursive doubling for 16 cache lines. Although this is similar to the figure in [IEEE90], ours is well-defined by the given function. It is important to note the simplicity of the hardware components that would be required in the implementation of this function for system size M being a power of two. There is only one loop and no recursion. Each loop iteration requires only three compares for equality, only one compare for inequality against a power of two, and only one decrement by either a one or a power of two minus one. Outside the loop the hardware is similarly simple. Although we have not implemented this function in hardware, our experience leads us to believe that it would be straightforward and appropriate as part of a real implementation.

Using recursive doubling with this function produces a solution to SLC with a latency of $9\log_2(N)-21$ message delays, $N \geq 16$. This is 123 message delays for 64K caches. In particular, recursive doubling creates the longest temporary pointers in $3\log_2(N)-7$ message delays $(N \geq 8)$, data is distributed to the remaining caches in $2\log_2(N)-4$ message delays $(N \geq 16)$, and the worst-case invalidation

Choice of Temporary Pointers




```
distance(size, position)          /* returns distance of temporary pointer */
int  size, position;              /* 1 <= position <= size == power of two */
{
  if (position <= 2 || size-1 <= position)  return (0);
  while (position != size-2 && position != size/2 && position != 2) {
    size = size / 2;
    if (position > size)  position = position - (size - 1);
    else                  position = position - 1;
  }
  if (position == 2)  return (0);
  return (size/2-2);
}
```

Figure 6

takes $4\log_2(N)-10$ message delays ($N \geq 16$). The invalidation latency is this bad because some of the longest temporary pointers can only be used when the list size N is maximal. One of the worst-cases for invalidation occurs when the list size is $M-\log_2(M)$, which is also the one of the worst cases for data distribution. When the list size is maximal, the latency for invalidations is about $2\log_2(N)$. Due to symmetry, data distribution is also about $2\log_2(N)$ when ignoring the time to create the temporary pointers.

These latencies compare favorably with the lower bounds given in previous sections. The latency of recursive doubling alone (see appendix) and recursive doubling with data distribution (see section 5) are each about twice the lower bound on latency. When data distribution and invalidation are considered together (see section 4), the latency is only half more than the lower bound. These results are summarized in figure 7b.

This function is the simplest known function that recursive doubling can use to produce temporary pointers as part of a logarithmic-latency solution to SLC. Also, there is no known non-recursive function that recursive doubling can use to provide a lower-latency solution. Since the protocol latency that results from the use of this function is about twice that of a lower bound and since its implementation complexity is reasonable, it is doubtful that a better function will be proposed. If improvements are to be found, it is likely that recursive doubling will be replaced by another algorithm, possibly interacting with list creation and combining (discussed in [JLGS90]).

7. Summary

In this paper we described the problem of scalable linked-list cache coherence (SLC). The problem is particularly relevant to the extensions [IEEE90] for the proposed IEEE-P1596 standard. As processors

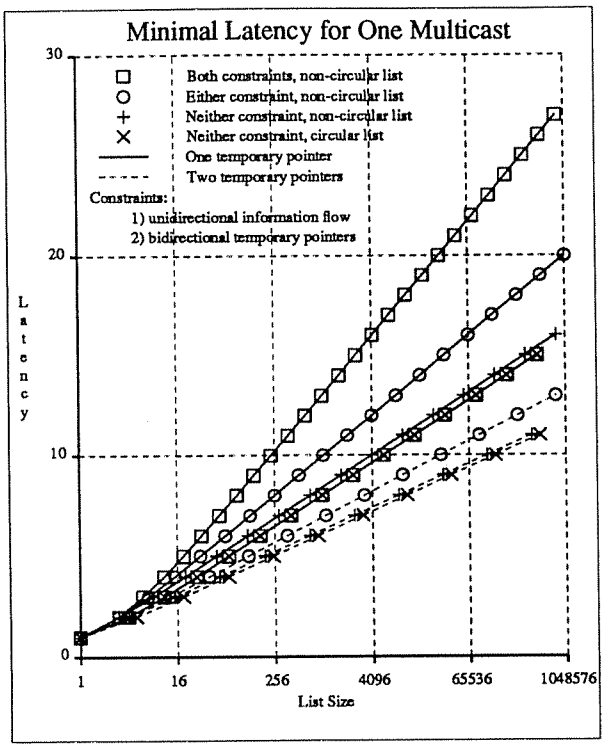


Figure 7a

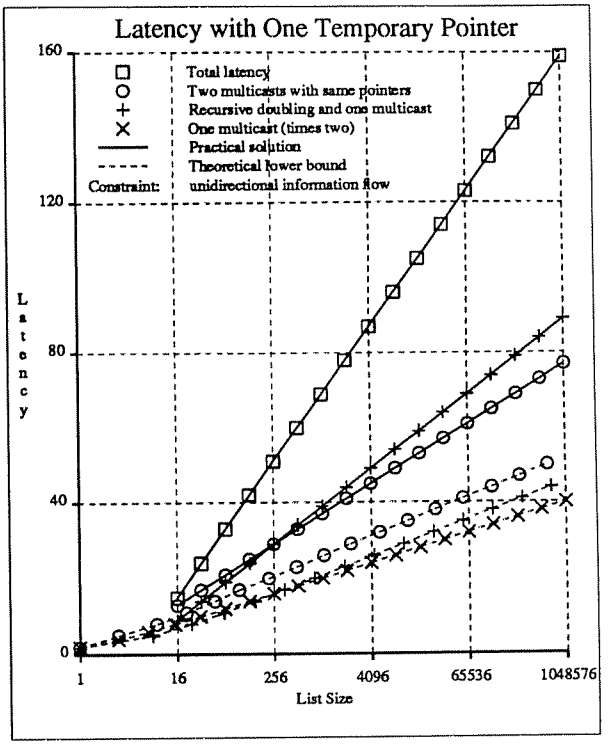


Figure 7b

read and write shared data, linked lists of cache lines are maintained by hardware. The cache-coherence protocols create these lists, distribute data, and purge these lists. To be scalable, temporary pointers are added, a small constant number per cache line. The problem is to determine which pointers are best to add and then determine how to create and use them.

We gave lower bounds on latency for creating these temporary pointers and using them for two separate multicasts. We showed (figure 4) that the potential advantages of additional temporary pointers decreases rapidly as more temporary pointers per cache line are added. This is reassuring since space limitations in the cache probably restrict the number of temporary pointers to one or two per cache line. We showed (summarized in figure 7a) that a number of other issues (circular lists (see appendix), unidirectional information flow, and bidirectional temporary pointers) are less important for one multicast than the number of temporary pointers.

We examined the pairwise interactions between recursive doubling and the two required multicasts, summarized in figure 7b. We showed that using the same temporary pointers for data distribution and invalidation necessarily increases the latency of both, assuming one temporary pointer per cache line. We gave an algorithm for creating temporary pointers, called recursive doubling, and showed that creation of temporary pointers contributes significantly to the latency of reads and can not be easily overlapped with data distribution. Finally, since control complexity in hardware is an issue, we proposed an

implementable choice of temporary pointers that yields about twice the latency of some theoretical lower bounds.

8. Acknowledgements

Advisor Jim Goodman, Dave James, and other members of the SCI working group provided interesting problems to solve and stimulating discussions. Eric Bach also provided stimulating comments. Stuart Friedberg provided papers on multicast. Many provided helpful suggestions on earlier drafts: Eric Bach, Jim Goodman, Dave James, and Steve Scott. This work was supported in part by NSF Grant CCR-892766.

9. References

[AdHi90] Sarita V. Adve and Mark D. Hill, "Weak Ordering - A New Definition," *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture* **18**, 2 (May 1990), 2-14.

[ASHH88] Anant Agarwal, Richard Simoni, John Hennessy and Mark Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture*, May 1988, 280-289.

[ArBa86] James Archibald and Jean-Loup Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems* **4**, 4 (November 1986), 273-298.

[Deer88] Stephen E. Deering, "Multicast Routing in Internetworks and Extended LANs," *Proceedings of the 1988 SIGCOMM Symposium*, August 1988, 55-64.

[DuSB86] M. Dubois, C. Scheurich and F. A. Briggs, "Memory Access Buffering in Multiprocessors," *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture* **14**, 2 (June 1986), 434-442.

[FrWB85] Ariel J. Frank, Larry D. Wittie and Arthur J. Bernstein, "Multicast Communication on Network Computers," *IEEE Software* **2**, 3 (May 1985), 49-61.

[GJKM90] Stein Gjessing, Sverre Johansen, Stein Krogdahl and Ellen Munthe-Kaas, "Fast Distribution of Information in SCI-like Cache Protocols," manuscript in preparation, Department of Informatics, University of Oslo, Norway, November 1990.

[Good83] James R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the Tenth Annual International Symposium on Computer Architecture*, June 1983, 124-131.

[IEEE90] "The Extended SCI Cache-Coherence Protocols," in *The Scalable Coherent Interface*, vol. 0.74 - P1596/Part III-C, David B. Gustavson and David V. James (eds.), October 1990.

[HeHL88] Sandra M. Hedetniemi, Stephen T. Hedetniemi and Arthur L. Liestman, "A Survey of Gossiping and Broadcasting in Communication Networks," *Networks* **18**, 4 (1988), 319-349.

[HiSt86] W. Daniel Hillis and Guy L. Steele, Jr., "Data Parallel Algorithms," *Communications of the ACM* **29**, 12 (December 1986), 1170-1183.

[JLGS90] David V. James, Anthony T. Laundrie, Stein Gjessing and Gurindar S. Sohi, "Scalable Coherent Interface," *IEEE Computer* **23**, 6 (June 1990), 74-77.

[Lamp79] Leslie Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs," *IEEE Transactions on Computers* **C-28**, 9 (September 1979),

690-691.

[PfNo85] G. F. Pfister and V. A. Norton, "'Hot-Spot' Contention and Combining in Multistage Interconnection Networks," *ACM Transactions on Computer Systems* **C-34**, 10 (October 1985), 943-948.

[Pugh90] William Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM* **33**, 6 (June 1990), 668-676.

[PuBr85] P. W. Purdom, Jr. and C. A. Brown, *The Analysis of Algorithms*, CBS Publishing, New York, 1985.

[TDLS90] Shreekant Thakkar, Michel Dubois, Anthony T. Laundrie and Gurindar S. Sohi, "Scalable Shared-Memory Multiprocessor Architectures," *IEEE Computer* **23**, 6 (June 1990), 71-74.

[ThDe90] Manu Thapar and Bruce Delagi, "Stanford Distributed-Directory Protocol," *IEEE Computer* **23**, 6 (June 1990), 78-80.

[Wint87] Pawel Winter, "Steiner Problem in Networks: A Survey," *Networks* **17**, 2 (1987), 129-167.

10. Appendix

The appendix contains additional equations, graphs, secondary text, and some of the longer arguments. Exact analytical solutions have been computed for all recurrence relations and have then been verified by Vaxima, a symbolic manipulator. In addition, exhaustive searches of all pointer combinations for small numbers of cache lines have verified many of the equations. Where possible, exact solutions have been transformed to numerical solutions (accurate to four decimal places) to aid understanding and the exact solutions have been placed here.

In most cases we solve the equivalent problem of maximizing the number of cache lines for a given latency instead of minimizing the latency for a given number of cache lines. An approximation for number of cache lines in terms of latency is given here for the relevant cases. It can be shown that for

$$N = k_1 r_1^h + c + \sum_{i=2}^{t} k_i r_i^h,\ h \ge 1,\ |r_i| \le 1,\ t \ge 1,$$

$$h = \frac{\log_2(N)}{\log_2(r_1)} - \frac{\log_e(k_1)}{\log_e(r_1)} \pm \frac{|c| + \sum_{i=2}^{t} |k_i r_i|}{\log_e(r_1)\left[N - |c| - \sum_{i=2}^{t} |k_i r_i|\right]},\ N > |c| + \sum_{i=2}^{t} |k_i r_i|.$$

Although a smaller error term could be found by more detailed analysis, the given one is sufficient for interesting (large) N.

Equation 3.1

See figures 4 and 8. Also, $N(h) = 0.6036\,(2.4142)^h - 0.5000 - 0.1036\,(-0.4142)^h$ and

$$h(N) = 0.7864 \log_2(N) + 0.5729 \pm \frac{0.6160}{N - 0.5429},\ p = 1,\ N \ge 1.$$

Equation 3.1c The maximum number of cache lines contained in a *circular* doubly linked list is

$$N(h,p) = K_3\left[\frac{p+1+\sqrt{D_1}}{2}\right]^h - \frac{1}{p} + K_4\left[\frac{p+1-\sqrt{D_1}}{2}\right]^h \approx K_3\left[\frac{p+1+\sqrt{D_1}}{2}\right]^h - \frac{1}{p},$$

$$K_3 = \frac{D_1 + (p+1)\sqrt{D_1}}{2p\,D_1},\ K_4 = \frac{D_1 - (p+1)\sqrt{D_1}}{2p\,D_1},\ D_1 = p^2 + 6p + 1,$$

where p and h are as given in equation 3.1.

Arguments: $A(h)$ from equation 3.1 is the desired function to solve because the "end" of a circular list has two list pointers. Standard methods [PuBr85] yield the result to be proved. Note that the contribution of one root becomes negligible for large h. □

Also, $N(h) = 0.6768\,(2.4142)^h - 1 - 0.1464\,(-0.4142)^h$ and

$$h(N) = 0.7864 \log_2(N) + 0.1797 \pm \frac{1.2034}{N - 1.0607},\ p = 1,\ N \ge 2.$$

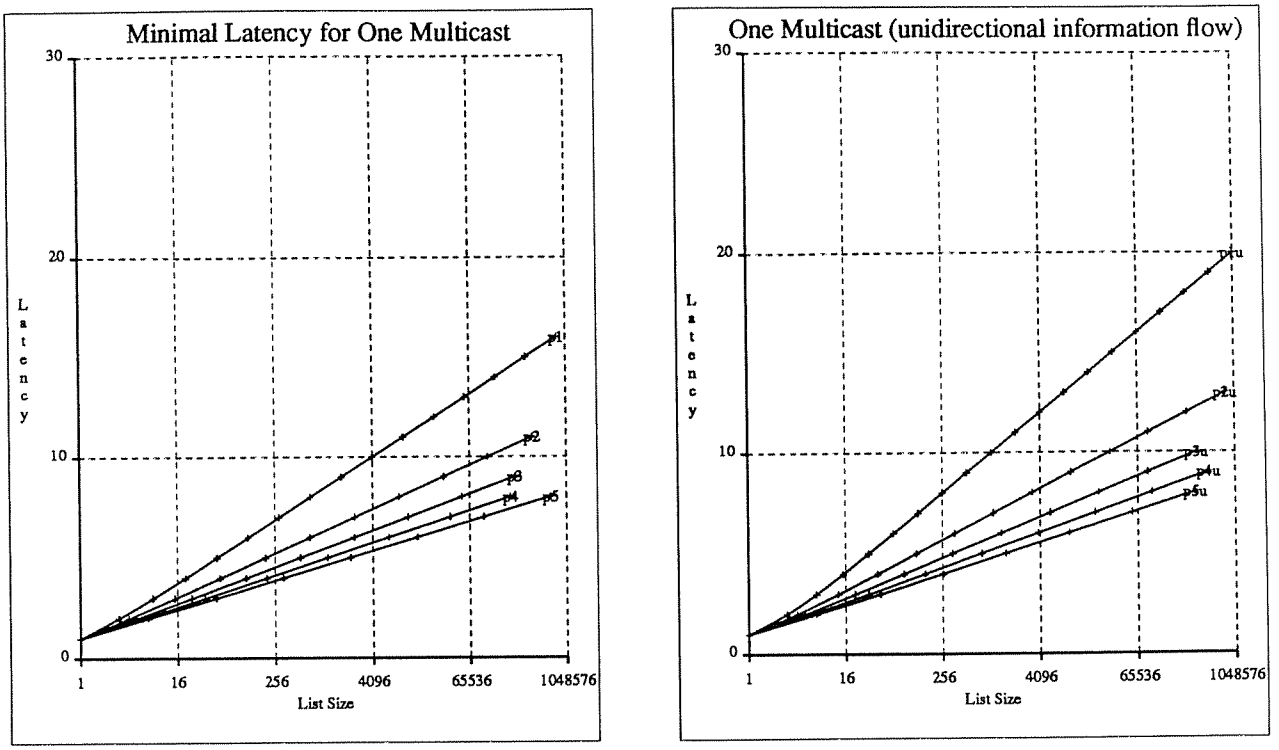


Figure 8

Equation 3.2

See figures 4 and 8. Also, $h(N) = \log_2(N+1) = \log_2(N) + 0 \pm \frac{1.4427}{N-1}$, $p = 1$, $N \geq 2$.

Equation 3.2c If information flow is unidirectional then the maximum number of cache lines contained in a *circular* doubly linked list is the same as for a non-circular list. However, if temporary pointers are bidirectional then the maximum number of cache lines contained in a *circular* doubly linked list is

$$N(h,p) = \frac{p+2}{p}(p+1)^{h-1} - \frac{2}{p}, \quad h \geq 2,$$

where p and h are as given in equation 3.1.

Arguments: If information flow is unidirectional then the "end" of the circular list still has only $p+1$ usable pointers, but if temporary pointers are bidirectional then the "end" of the circular list has two usable list pointers. Except for the root cache line, the maximum branching factor is $p+1$, yielding a geometric sum,

$$N(h,p) = 1 + (p+2)\sum_{i=0}^{h-2}(p+1)^i.$$

Algebraic manipulation yields the result to be proved. □

Also, $h(N) = \log_2(N) - 0.5850 \pm \frac{2.8854}{N-2}$, $p = 1$, $N \geq 3$.

Equation 3.3

Arguments: Again, we can define two interdependent recurrence relations by identifying the two types of cache lines in the spanning tree and noting their maximum branching factors. If a line is connected to its parent by a list pointer, then the line can have $p+1$ children: p by temporary pointers and 1 by the list pointer of the appropriate direction. However, if a line is connected to its parent by a temporary pointer, then the line can have only p children because one of its temporary pointers connects to the parent. This leads us to the following recurrence relations, where h is the height of a subtree and $L(h)$ and $T(h)$ are the maximum number of lines in a subtree whose root is visited by a list pointer and a temporary pointer respectively.

$$L(h) = 1 + L(h-1) + p\,T(h-1),\ L(1) = 1,\ L(0) = 0,$$
$$T(h) = 1 + L(h-1) + (p-1)\,T(h-1),\ T(1) = 1,\ \text{and}\ T(0) = 0.$$

$L(h)$ is the desired function because the end of the list can have only one list pointer. Standard methods [PuBr85] yield the result to be proved. Note that the contribution of one root becomes negligible for large h. □

See figure 4. Also, $N(h) = 1.1708\,(1.6180)^h - 2 - 0.1708\,(-0.6180)^h$ and

$$h(N) = 1.4404 \log_2(N) - 1.3277 \pm \frac{4.2918}{N - 2.0652},\ p = 1,\ N \geq 3.$$

Equation 3.3c If temporary pointers are bidirectional *and* information flow is unidirectional, then the maximum number of cache lines contained in a *circular* doubly linked list is the same as for a non-circular list, where p and h are as given in equation 3.1.

Arguments: The "end" of a circular list also has $p+1$ children. □

Equation 4.1

Arguments: Let j be the number of pointers in $T(l,d)$ that are not nested (see figure 9). Intuitively, this

Construction for Two Multicasts

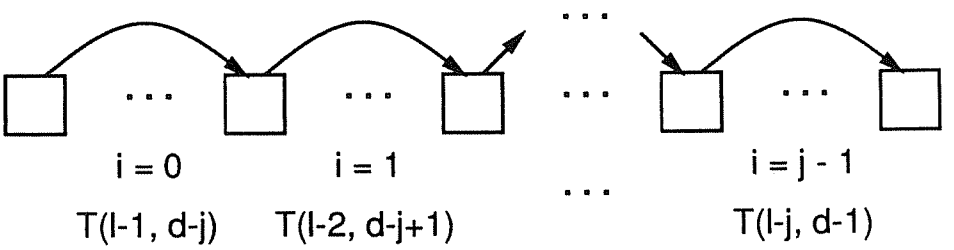


Figure 9

divides $T(l,d)$ into two groups of cache lines: (1) $j+1$ lines, including the endpoints, that are sequentially connected by temporary pointers (or list pointers for the base cases) and (2) the j segments of the lines that are spanned by the j pointers. (If a segment is spanned by a list pointer, its length is zero.) Note that it takes $i+1$ hops for information to reach and enter the i^{th} segment ($0 \le i < j$) from the left endpoint and so the spanning tree from that endpoint has $l-(i+1)$ hops remaining. Likewise, it takes $j-i$ hops to reach and enter the segment from the right endpoint, leaving $d-(j-i)$ hops. Therefore, the i^{th} segment contains exactly $T(l-i-1\ d-j+i)$ cache lines and $T(l,d)$ contains the unnested cache lines plus the sum of the segments, as given by the formula.

It remains to determine j. There can be at most $\min\{l,d\}-1$ nested segments because spanning trees for nested segments cannot have negative height. For the same reason, j can not be less than zero. Therefore, $T(l,d)$ is maximized according to the given formula. □

See figure 10. Experimentally, we have found that j is *not* fixed as l and d vary. Due to this, we have been unable to find a closed-form solution. However, linear regression analysis for $1 \le \log_2(N(h)) < 31$ gives $h(N) \simeq 2.6\log_2(N)+0.2$. We have also noted that $\lim_{l\to\infty} T(l,d) = \lim_{d\to\infty} T(l,d) = 2^{\min\{l,d\}} - 1$ for $l,d \le 40$, which agrees with equation 3.2 for $p=1$.

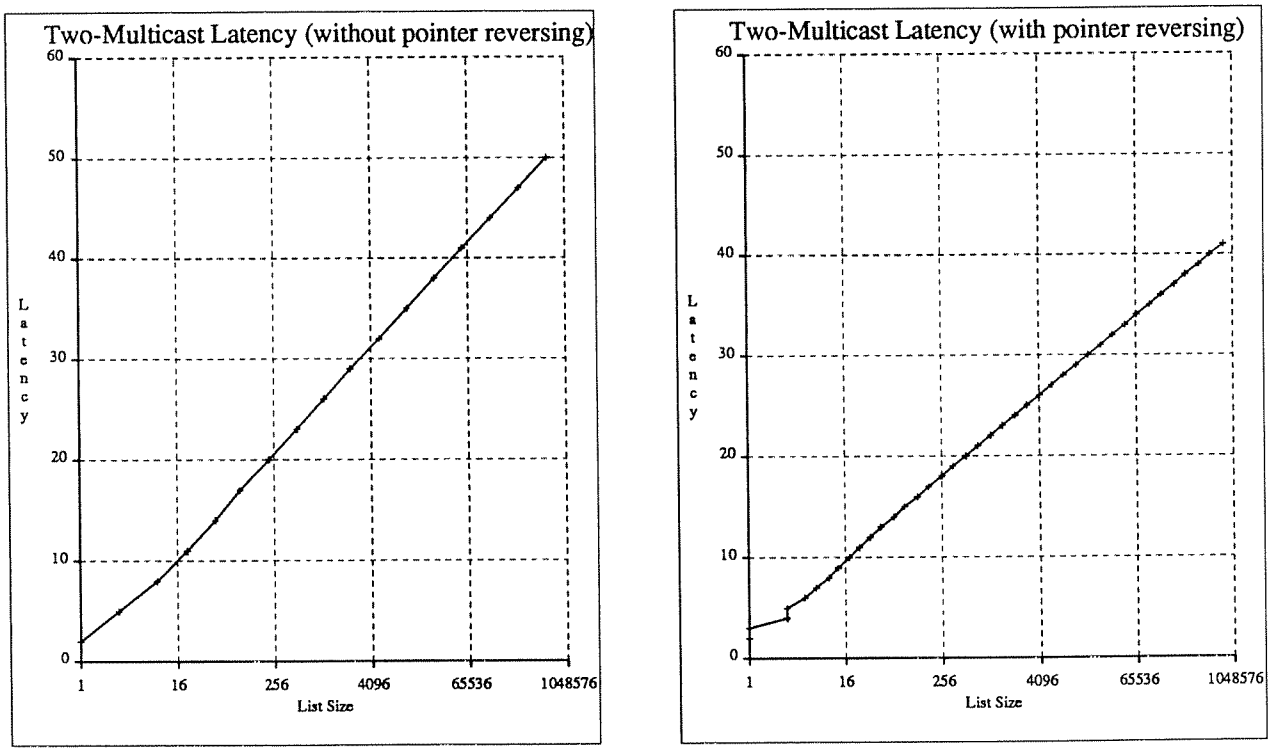


Figure 10

We relax the constraints of the previous equation for the next equation. Since all temporary pointers for data flow must point downstream (towards the tail) to be useful, some pointers in the equation may need to be reversed for invalidation. Pointer reversing is as follows. When a cache line responds to a request for data, the destination of the response is saved and the downstream pointer is discarded, if any. The choice of which pointers to reverse is independent of the mechanism.

Equation 4.2 Given three end-rooted trees of a *non-circular* doubly linked list with only one temporary pointer per cache line such that (1) the first and second trees collectively span the list and are rooted at opposite ends of the list and (2) the third tree (a) is rooted at the same end of the list as the second tree, (b) independently spans the list, and (c) allows only unidirectional information flow, *then* the maximum number of cache lines contained in the list is $T(l,r,d)$:

$$T(0,0,0) = T(0,0,d) = T(l,r,0) = 0,$$

$$T(l,r,d) = \max\{\min\{l+r,\ d\},\ F(l,r,d)\},$$

$$F(l,r,d) = \max_{\substack{1 \le j < \min\{l+r,\ d\} \\ \max\{0,\ j-r\} \le k \le \min\{j,\ l\}}} \left\{ 1 + j + \sum_{i=0}^{j-1} T\begin{bmatrix} G(k-0,\ i,\ l-i-1,\ r-j+i-1), \\ G(k-1,\ i,\ l-i-2,\ r-j+i-0), \\ d-j+i \end{bmatrix} \right\}$$

$$G(u,v,x,y) = \begin{cases} x & \text{if } u > v \\ y & \text{if } u < v \\ \max\{x,y\} & \text{if } u = v \end{cases},$$

where $l \ge 0$, $r \ge 0$, and $d \ge 1$ are the heights of the first, second, and third trees respectively and $l+r \ge 1$.

Arguments: Figure 11 gives the intuition for the construction. We have not yet tried to worked through the details of a argument. Parts of the equations have been verified by exhaustive search. □

Note that we want to know the maximal number of cache lines N for a given sum h of the two multicasts latencies in SLC. Using regression analysis we can show that

Construction for Pointer Reversing

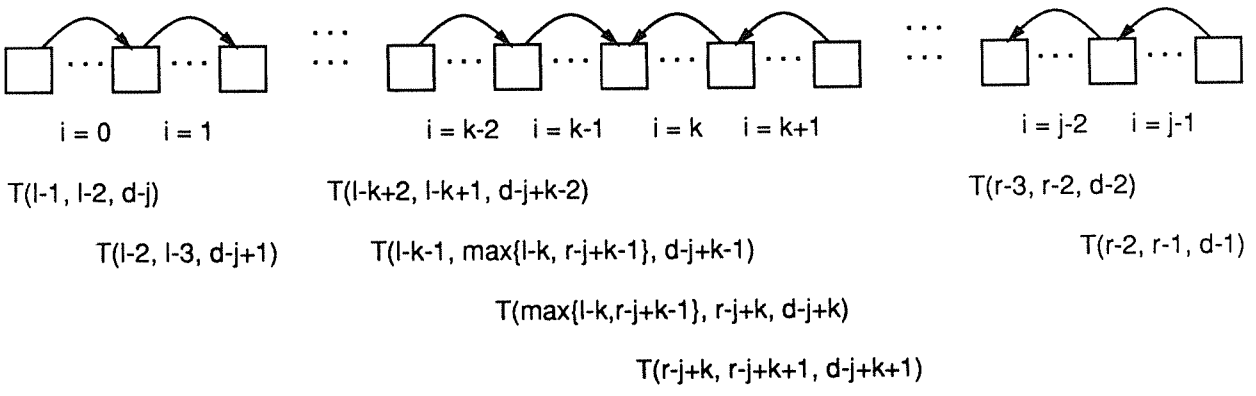


Figure 11

$$N(h) = \max\{T(l, 0, d): l + d = h\} \approx 0.5\,(1.4)^h.$$

See figure 10. Linear regression analysis for $1 \le \log_2(N) < 31$ gives $h(N) \approx 2.0 \log_2(N) + 1.9$.

Equation 5.1 Assuming at most one temporary pointer per cache line and that information flow is unidirectional, $T(r,d)$ is the maximum number of cache lines such that data can be distributed in d message delays after a pointer between the endpoints exists after r message delays, where

$$T(r,d) = T(r-1, d-1) + T(r-2, d-1) + 1,\ T(0,0) = T(x, 0) = T(1,x) = 0.$$

Arguments: Construct the temporary pointers for the list as follows (see figure 12). Partition the list into M segments, where $M = \min\{r-2, d-1\}$ and the right endpoint of one segment is the same as the left endpoint of another. Then create a temporary pointer across each segment from the left endpoint to the right. Recursively construct the temporary pointers for each list with new rules for r and d, as follows. Since data reaches the z^{th} segment in z message delays, the data must be distributed throughout that segment in $d - z$ message delays. Since recursive doubling for the whole list must complete in r message delays, the z^{th} segment must finish recursive doubling in $r - z - 1$ message delays. Also note that recursive doubling can not complete in less than 2 message delays (request and response) and that data can not be forwarded in less than 1 message delay. These observations lead to the following formula.

$$T(r,d) = M + 1 + \sum_{z=1}^{M} T(r-z-1, d-z),\ r \ge 2,\ d \ge 1,\ T(0,0) = T(x, 0) = T(1,x) = 0,\ x \ge 1.$$

$M + 1$ represents the number of overlapping endpoints of these segments, sequentially connected by temporary pointers for segments that are longer than two. Each term of the summation represents the length of the temporary pointer that spans the z^{th} segment (length zero if no temporary pointer). Since this construction satisfies the constraints of r and d, it remains to show that it is maximal and that it is equivalent to the result to be proved.

Three observations show that $T(r,d)$ is maximal. First, M is the largest number of segments that are possible for the given values of r and d. Second, decreasing the length of any segment would not allow an increase in the length of any other segment because the constraints are independent. Third, decreasing the

First Construction for One Multicast with Recursive Doubling

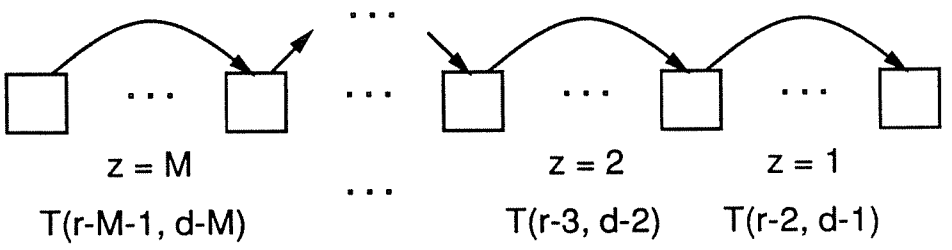


Figure 12

number of segments decreases the total length of the list because removal of the last i segments does not relax the constraints (on r or d) for the other $M-i$ segments. Therefore, $T(r,d)$ is maximal.

Finally, subtracting $T(r,d)$ from $T(r+1, d+1)$ yields the result to be proved. □

Since it can be shown that $\sum_{i=0}^{j} \binom{i}{n} = T(n+j, n) - T(n+j, n-1)$, $j \le n$ and since there is no known closed-form solution for partial sums of Pascal's triangle, we believe that $T(r,d)$ does not have a closed-form solution.

Equation 5.2

$$N(h) = K_{13}\left[\frac{D_3^{2/3} + 2^{2/3}}{D_3^{1/3}\, 2^{1/3}\sqrt{3}}\right]^h - 4 + K_{14}\left[\frac{D_3^{2/3}\,\bar{\omega} + 2^{2/3}\,\omega}{D_3^{1/3}\, 2^{1/3}\sqrt{3}}\right]^h + K_{15}\left[\frac{D_3^{2/3}\,\omega + 2^{2/3}\,\bar{\omega}}{D_3^{1/3}\, 2^{1/3}\sqrt{3}}\right]^h,$$

where

$$K_{13} = 1 + \frac{E_3 + F_3}{276},\quad K_{14} = 1 + \frac{E_3\,\omega + F_3\,\bar{\omega}}{276},\quad K_{15} = 1 + \frac{E_3\,\bar{\omega} + F_3\,\omega}{276},$$

$$E_3 = D_3^{2/3}\, 2^{1/3}\left[207 - 19\sqrt{3}\sqrt{23}\right],\quad F_3 = D_3^{1/3}\, 2^{2/3}\left[92\sqrt{3} - 18\sqrt{23}\right],\quad D_3 = \sqrt{23} + 3\sqrt{3},$$

and $\omega = \dfrac{-1+\sqrt{-3}}{2}$ is a primitive cube root of unity.

Arguments: Construct the temporary pointers for the list as follows (see figure 13). Partition the list into $h-2$ segments, where the right endpoint of one segment is the same as the left endpoint of another. Then create a temporary pointer across each segment from the left endpoint to the right. Recursively construct the temporary pointers for each segment with rules for $T(r,d)$ as follows. First, recursive doubling must complete in $r = z+1$ message delays so that data reaches the z^{th} segment in $z+1$ message delays. Second, the data must be distributed throughout the z^{th} segment in $d = h-(z+1)$ message delays so that the latency is h message delays. These rules lead to the following formula.

Second Construction for One Multicast with Recursive Doubling

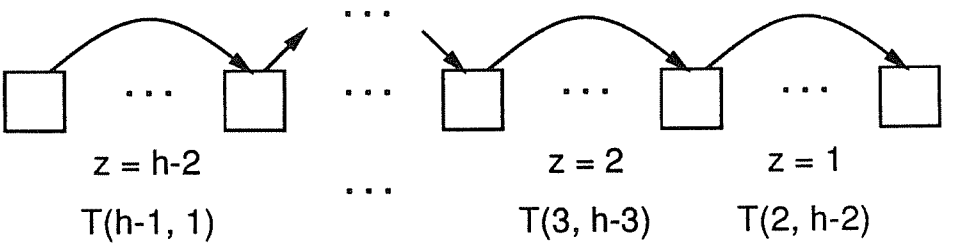


Figure 13

$$N(h) = h - 1 + \sum_{z=1}^{h-2} T(z+1, h-z-1),\ h \geq 1.$$

The $h-1$ term represents the number of overlapping endpoints of these segments, sequentially connected by temporary pointers for segments that are longer than two. Each term of the summation represents the length of the temporary pointer that spans the z^{th} segment (length zero if no temporary pointer). Since this construction satisfies the constraints of h, it remains to show that the formula is equivalent to the result to be proved and that it is maximal for given h.

First, show equivalence. Using the formula for $N(h)$ and the recurrence relation for $T(r,d)$, we derive a recurrence relation for $N(h)$.

$$N(h) = h - 1 + \sum_{z=1}^{h-2} T(z+1, h-z-1) = h - 1 + \sum_{z=2}^{h-1} T(z, h-z).$$

$$N(h+1) = h + \sum_{z=2}^{h} T(z, h+1-z),\ N(h+3) = h + 2 + \sum_{z=2}^{h+2} T(z, h+3-z).$$

$$N(h+3) = h + 2 + \sum_{z=2}^{h+2} T(z-1, h+2-z) + \sum_{z=2}^{h+2} T(z-2, h+2-z) + (h+1),$$

$$N(h+3) = 2h + 3 + \sum_{z=1}^{h+1} T(z, h+1-z) + \sum_{z=0}^{h} T(z, h-z),$$

$$N(h+3) = h + (h-1) + 4 + \sum_{z=2}^{h} T(z, h+1-z) + \sum_{z=2}^{h-1} T(z, h-z).$$

$N(h+3) = N(h+1) + N(h) + 4$, where $N(1) = 0$, $N(2) = 1$, and $N(3) = 3$.

Standard methods [PuBr85] yield the result to be proved.

Three observations show that $T(r,d)$ is maximal. First, $h-2$ is the largest number of segments that are possible for the given value of h. This is because it takes two message delays for data to reach the first segment (one request and one response) and then the remaining message delays to reach the remaining endpoints at the rate of one message delay per endpoint. Second, decreasing the length of any segment would not allow an increase in the length of any other segment because the constraints are independent. Third, we *believe* that decreasing the number of segments could not increase the total length of the list. The exact formula is verified by exhaustive search for $h < 12$ and $N \leq 72$. Therefore, $N(h)$ is correct as stated. Note that the contributions of two roots become negligible for large h. □

See figure 14. Also, $h(N) = 2.4650 \log_2(N) - 3.8423 \pm \frac{14.9755}{N - 4.2111},\ N \geq 5.$

Equation 5.3 The longest temporary pointer that can be created in $h \geq 1$ message delays by recursive doubling is

$$N(h) = \left[\frac{1+\sqrt{5}}{2}\right]^h - 1 + \left[\frac{1-\sqrt{5}}{2}\right]^h \approx \left[\frac{1+\sqrt{5}}{2}\right]^h - 1.$$

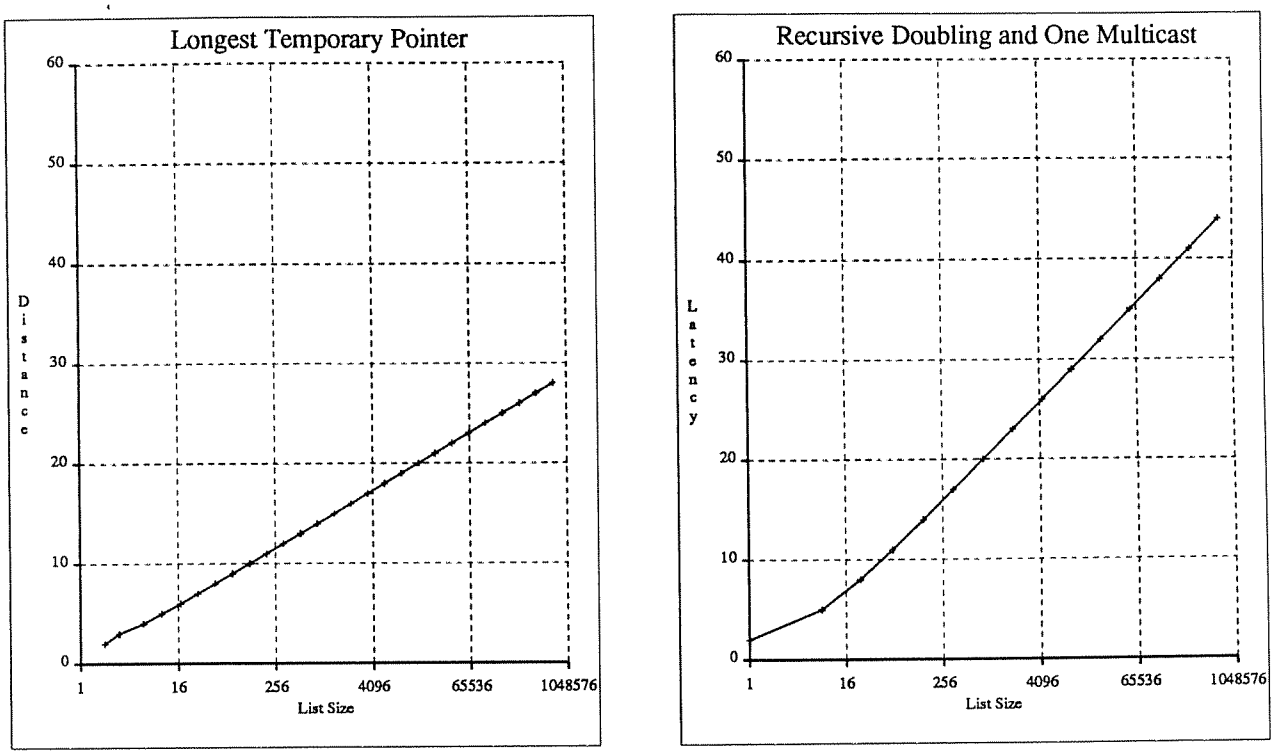


Figure 14

Arguments: (sketch) The construction is shown in figure 15. Also recall figure 3c.

$$N(h) = +\sum_{i=2}^{h-2} N(i),\ h \geq 4,\ N(3)=3,\ N(2)=2,\ N(1)=0.$$

$$N(h)-N(h-1)-N(h-2)=1,\ N(2)=2,\ N(1)=0.$$

Construction for Longest Temporary Pointer

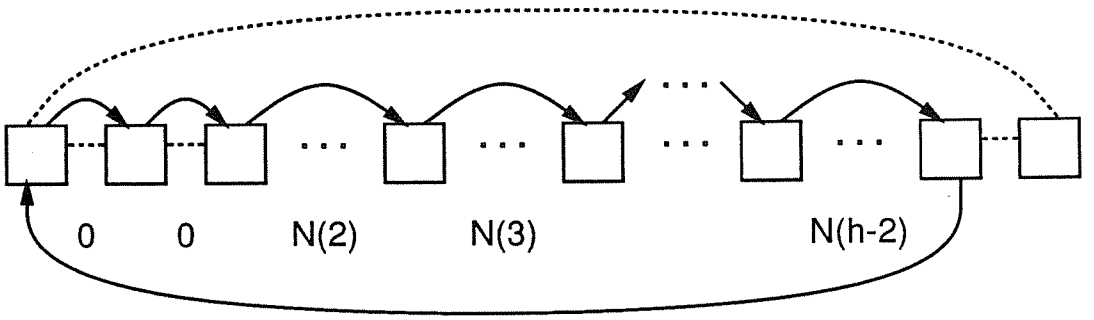


Figure 15

Note that the contribution of one root becomes negligible for large h. □

See figure 14. Also, $N(h) = (1.6180)^h - 1 + (-0.6180)^h$ and $h(N) = 1.4404 \log_2(N) + 0 \pm \frac{3.3624}{N - 1.6180}$, $N \geq 2$.