Hierarchical Extensions to SCI

James R. Goodman
Stefanos Kaxiras

Technical Report #1235

July 1994

Hierarchical Extensions to SCI[1]

James R. Goodman, Stefanos Kaxiras
University of Wisconsin Madison

Computer Sciences Department
University of Wisconsin - Madison
1210 W. Dayton Street
Madison, WI 53706

July 12, 1994

1 This work is supported in part by National Science Foundation grant #CCR-9207971

Abstract

The cache coherence scheme of the Scalable Coherent Interface (SCI) offers performance for some operations degrading linearly with degree of sharing. For large scale sharing, we need schemes that offer logarithmic time reading and writing of shared data for acceptable performance. Therefore, we need to move from SCI's sharing lists to sharing trees. The currently proposed Kiloprocessor Extensions to SCI define tree protocols that create, maintain and invalidate binary trees without taking into account the underlying topology. However these protocols are quite complex.

We propose a different approach to Kiloprocessor Extensions to SCI. We define k-ary trees that are well mapped on the topology of a system. In this way the k-ary sharing trees offer great geographical locality (neighbors in the tree are also physical neighbors). The resulting protocols are simple and in some cases their performance for reading and writing shared data is superior to the previous protocols. We present our protocols on two example topologies. The first is the well known k-ary n-cube. We introduce the second topology which is a type of Omega topology, constructed of rings. In order to implement our new schemes we decouple data from directory information. In this way we can cache directory information on various nodes without the corresponding data. Regarding the distributed directory information, we define it in terms of a cache tag with multiple pointers. The actual implementation of such a tag can be in a form of a linked list (or other structure) so we can have dynamic pointer allocation. Set associative caches can be used for fast dynamic allocation and deallocation of pointers.

Investigation of the new approach resulted in several variants of the protocols. We present most of the variants here, without comparing them in depth, in terms of performance or cost.

Contents

1. Motivation

The current proposal for Kiloprocessor Extensions to Scalable Coherent Interface [1], which we will refer to as STEM [2], has achieved its goal of a logarithmic-time algorithm to build, maintain and invalidate a sharing binary tree without taking into account the topology of the interconnect network. The complexity of the algorithm however, is very high. It requires complex transactions that generate a lot of traffic. Most significantly this traffic is non-local and consumes large portions of system bandwidth. The complexity of the algorithm impacts negatively on the marketability of SCI to high performance computing manufacturers, not to mention academic environments. Clearly if we want to do better, that is, attain comparable or even higher performance with considerably less complexity, a new approach is needed.

The starting point for STEM for a binary sharing tree is that an $O(\log_2 N)$ algorithm is needed for logarithmic time reading and writing shared data. However, this algorithm was proposed under the assumption that the latency of any message is unit latency (i.e. all messages have the same latency, regardless of the distance they travel). This is not true in a real system. When we take into account the increase of message latencies as a function of system size, it is not clear if the logarithmic nature of the algorithm can be translated into logarithmic performance, regardless of topology and system size.

Combining [3] has been proposed to reduce the bandwidth requirements of the STEM algorithm and possibly capture some geographical locality. Combining takes two requests that happen to be at the same time in a queue, and generates a single new request and a response to one of the original requests. Combining is not guaranteed to work with any load in the network. In fact it may only work in the presence of congestion in the network, i.e. with high loads[2].

We started from the assumption that in order to achieve high performance we need to be able to near-optimally map any sharing tree on top of any reasonable topology[3]. In this way, neighbors in the logical sharing tree will also be physical neighbors. Sending a message from one node to the next (following a pointer), means that the message only travels a short distance with very low latency, consuming little bandwidth. Such a tree is impossible with STEM, simply because its structure is dependent on the timing of the requests. In STEM we have temporal locality in the tree. Requesting nodes, in close proximity in time, are neighbors in the tree. What we would like, is to exploit geographical locality in the tree. By geographical locality we mean that nodes in physical proximity are neighbors in the tree, regardless of the timing of their requests. In this work we present schemes that create trees that map well on the underlying topology. In our protocols messages are exchanged only between nodes on the same ring with very low latencies. Furthermore, we guarantee reduction of traffic, unlike the case where combining is used.

---

2 Admittedly this is the case when combining is mostly needed.

3 Of course there are topologies on which mapping a sharing tree is meaningless, useless or nearly impossible (i.e. single rings or completely connected systems). However large high performance systems are likely to have topologies, where requests, going towards a memory, and data distributed back to nodes, naturally form trees.

2. Hierarchical Extensions to SCI

We propose building a tree that closely matches the underlying topology. The tree is made of linear SCI-like sharing lists. Each such list is confined to one physical ring. Although the tree is not going to be optimal in the theoretical sense (some parts are linear) the operations involved (creation, rollout, invalidation) are going to be fast because messages are generally confined to a single ring.

Our approach in building such trees is a form of hierarchical caching. Nodes in the same ring will send their requests, for a memory line, through the same switch node which we call *agent* node. If at this agent we have a copy of the line, we can satisfy these requests locally. The requesting nodes will form a small linear list. The head of this list will point to the agent, and not the home node of the line. The agent, along with other nodes in the next ring towards the home node, will also send their requests to their common agent and they will form their own small list. This process continues, until we reach the ring of the home node.

In order to be able to take full advantage of the wide variety of topologies that can be used to build SCI systems we propose *k-ary* logical sharing tree of lists (in contrast to STEM that specifies a binary tree). The degree *k* is dependent on the topology used. In general we specify a degree *k*, if a node is connected to *k+1* rings[4].

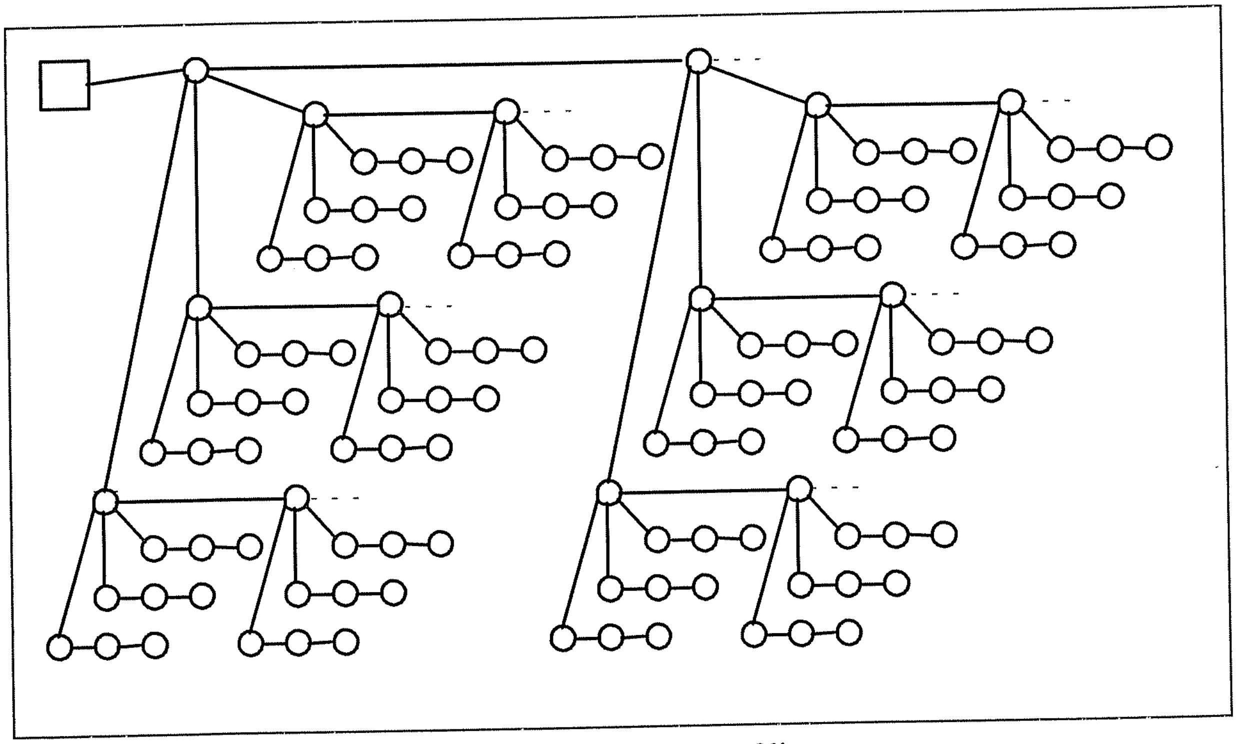

Fig. 1 Logical k-ary tree of lists

[4] Actually we define a degree *k* when the rings connected to a node can be partitioned to *k+1* sets, so that all rings in the same set contain the same nodes. For the sake of clarity, we will only talk about one ring per set.

An example of a logical tree is shown in figure 1. In this figure memory is represented by the square in the top left corner, cache lines in nodes are represented by circles. The horizontal strings of cache lines are the linear lists. Several of the nodes in the above example have three children lists. A linear list and its parent are contained in one physical ring. The nodes that have child-lists, are the agents.

We say that all nodes in a list belong to a level of hierarchy. The nodes of the list that is connected to memory, are in the highest level of hierarchy. The nodes that cannot have children are in the lowest level. The nodes of the children lists of a node in level *L* are in level *L-1* or lower[5]. In figure 1, the tree has three levels of hierarchy and the lists in the lowest level are shown to have three members. Figure 2 shows, in the left, an agent in a high level list and its three child-lists. On the right figure 2 shows a mapping of the lists onto rings.

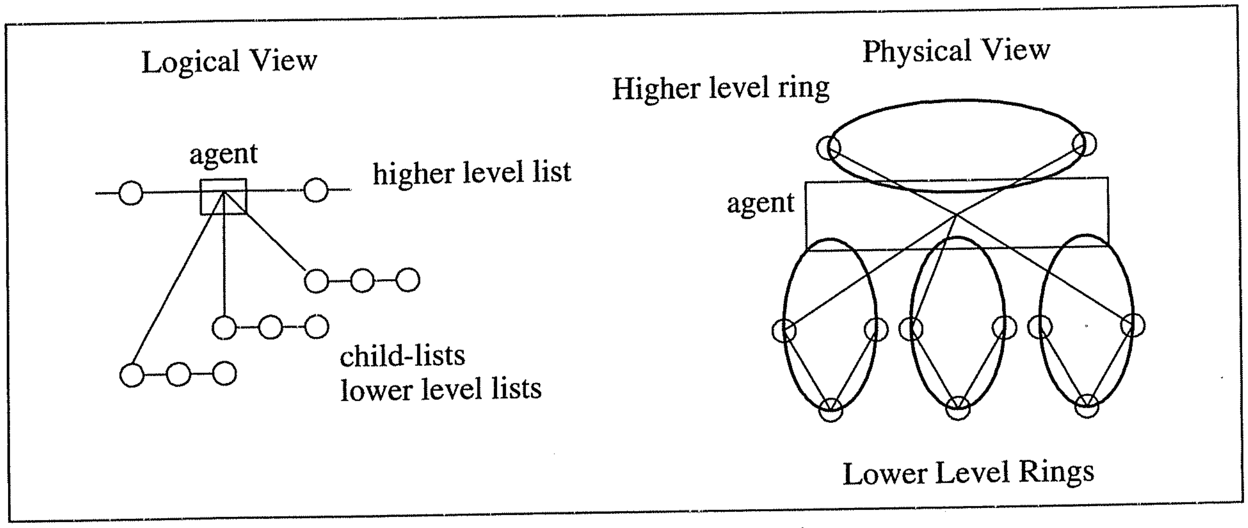

Figure 2. Mapping of lists onto rings.

A simplified description of the algorithm for creating such a tree is given below; a more detailed version is given in a later section. A node reading an uncached line sends a special request towards the home node of the remote line. At the first agent, where this request is taken off a ring, it is intercepted. The intercept generates a lookup in the SCI cache of the agent. If the lookup results in a hit, the requesting node is instructed to prepend to the appropriate child-list and a copy of the cache line is returned (either by the agent or by the previous head of the child-list). If the lookup results in a miss, the requesting node is instructed to prepend to the appropriate child-list and wait for the data. The agent then sends its own request for the line towards the home node[6]. This new request will be treated in the same way at the next agent. Eventually the last agent will request the line from the home node (if it does not have a cached copy) and pass it back to the waiting nodes.

5 See the k-ary n-cube example, where a child can be several levels lower than the parent.

6 The agent does not forward intercepted requests but rather it replaces all of them by its own request.

In summary the main points of our proposal are :

- We build the logical k-ary tree having as a blueprint the underlying topology
- The k-ary tree is made of linear lists. A linear list is confined inside a single ring.
- The cache lines that are members of the same linear list have a common agent on the ring that by default routes requests for the cached address to other rings.
- Such agents have a cached line that is part of the tree and it can point to a number of child lists.

We are also investigating a more radical departure from the base SCI protocol that offers higher performance. The scheme depends on the observation that entire lists created under the proposed scheme are generally contained within a single ring. This fact, and the expectation that a modest number of nodes will be attached to a single ring, suggests the possibility of storing directory information related to a single list not in the form of doubly-linked list, but as a bitmap. Storing the information as a bitmap opens up a range of alternative algorithms, many using broadcast, and in particular, permits a pruning cache implementation in the style of that proposed by Scott and Goodman [9].

3. Examples

In trying to come up with examples of topologies we skipped the obvious topologies where our scheme would be an obvious fit. These include trees, snowflakes, dense snowflakes and star topologies [4], the lens [5], and various other topologies. Although these are interesting topologies, we think that they are not mainstream and some of them also give an "unfair" advantage to our proposals by being so hierarchical in nature.

To illustrate our approach we chose two topologies because (1) we believe they are likely candidates for implementation and (2) it is not trivial to see how our schemes fit them. The first is k-ary n-cube topology made of rings, suggested in the SCI standard (p.12). We introduce the second topology, which we call Omega/Flip. This topology is a refinement of the butterfly topologies suggested also in the SCI standard (p.11). We believe that Omega/Flip topologies are a good fit for SCI but this is an appropriate discussion for a different document.

We found that our schemes compare favorably to STEM, while being much less complex. In several cases, our schemes in the two example topologies outperform STEM. More details about performance will be given in the appropriate sections.

3.1. k-ary n-cube

In this example the routing of requests is fixed in order to have deadlock free routing [6]. For example, in a k-ary 3-cube the request messages are routed first along x, then y and finally z dimensions[7]. At the place where requests switch to get into the next dimension they are

[7] Responses to requests are not required to return in the opposite way (i.e. routed first in z, then in y, and finally in x dimensions), but they too, can follow the same routing as the requests. It is up to the designer of a system to decide the routing. However, in our schemes, requests and replies travel only in one dimension because they stop at the first agent.

intercepted. A new request is generated and continues in the next dimension. The response to an intercepted request travels back in the dimension the request used.

In figure 3 we show a 4-ary 3-cube which is equivalent to a 3-D Torrus[8]. This topology has four nodes per ring and three rings per node. The total number of nodes is 64. Without loss of generality we assume that all the nodes access a line in some node, labeled "Home node."

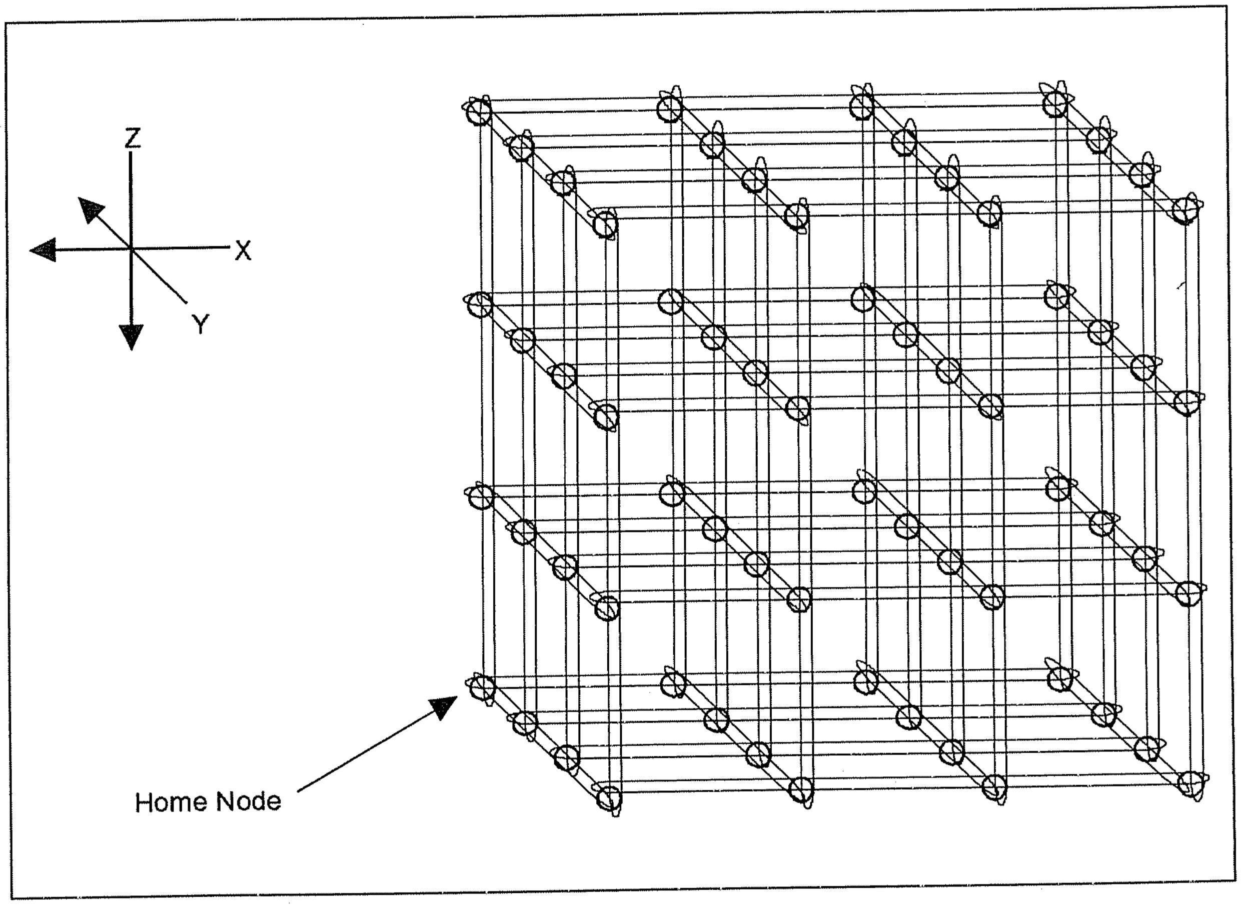

Figure 3. 4-ary 3-cube

In the sharing trees of this topology, not all nodes have the same number of child lists. The higher the hierarchical level of a node, the more child lists it has. For this example, nodes in the lowest level have none, in the next level they have one, and in the top level they have two. In general, in a k-ary n-cube, the logical tree will have *n* hierarchical levels. At the highest level (the list that is directly connected to memory) the nodes will have *n-1* child-lists. The child-lists do not, necessarily belong to the same hierarchical levels. Moving one level down, the number of child-lists decreases by one, until the lowest level, where the nodes do not have any child-lists. In figure 4 we show how a k-ary tree is mapped on the 4-ary 3-cube. In the second representation of the tree, in figure 4, we draw it as a tree of lists. We have numbered a few nodes to correlate the two representations.

[8] Note that CRAY T3D uses a 3-D Torrus

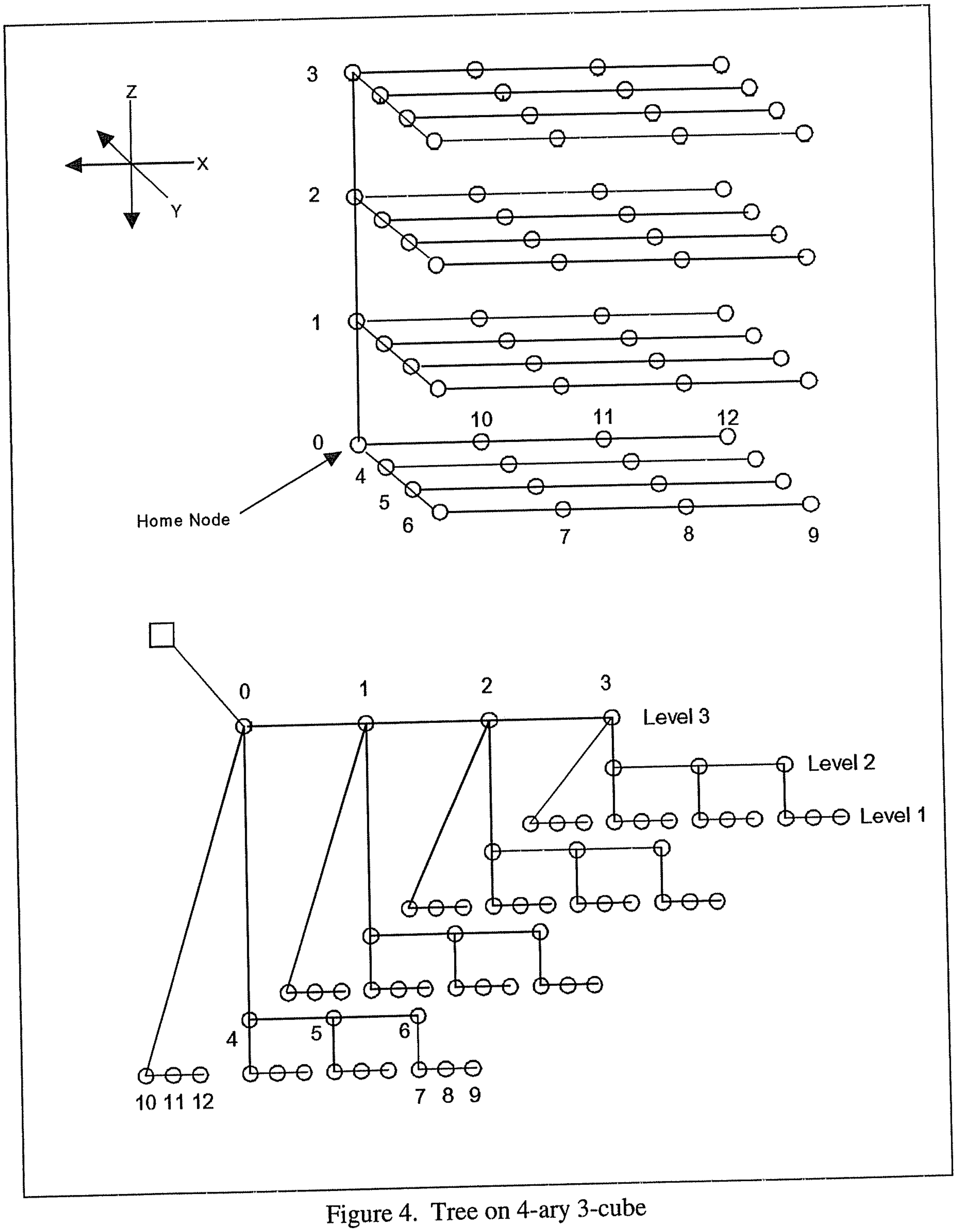

Figure 4. Tree on 4-ary 3-cube

The height (longest path) of the logical tree is from node 0 to the far right node in Level 1. In a k-ary n-cube a tree's maximum height is given by the formula : $(k-1)\times n$. It is easy to derive that the best balanced STEM tree has a height of $(\ell og_2(\frac{k^{(n)}}{2}))\times 2 = 2\times n\times \ell og_2(k) - 2$. Comparing these two formulas, for a small range of *n,* our tree is shorter for $k\leq 5$. In table 1 we present the system size for a few values of *n* for which our tree is shorter.

n	k	System Size
2	4	16
3	5	125
4	5	625
5	5	3,125
6	5	15,625

Table 1.

For cases where our tree is longer than STEM we believe our scheme will still out perform STEM because of the shorter paths of individual links.

3.2 OMEGA/FLIP Topologies

In his thesis Ross Johnson suggested building butterfly topologies with rings [2]. A disadvantage of the topologies he suggested, is that the rings have a varying number of nodes. We propose two variations of these topologies, the Omega [7] and the Flip. The important characteristic of these two topologies is that they are built of rings of the same size. The ring-based Omega topology, has *R* rings per node, *S* stages, and *N* nodes per stage. The nodes of last stage are connected to the nodes of the first stage, making the topology a cylinder. The rings run once around the cylinder, and they contain *S* nodes. The Flip is the reverse of the Omega: a counterclockwise Omega is a clockwise Flip. The Omega/Flip is made of one Omega and one Flip network superimposed[9], as shown in figure 5. The relations that define the Omega topologies we suggest are:

$$TotalSystemSize = S\times N = S\times R^{S-1} = (log_R(N) + 1)\times N$$

$$S = log_R(N)+1 \Leftrightarrow R^{S-1} = N$$

T*otalSystemSize* is the total number of nodes. The independent variables are *S* and *R*. Variable *N* is constrained to have specific integer values, by the above relations.

The reason we use Omega/Flip instead of either the Omega or the Flip is that it minimizes the total distance, a request and its response travel. One can argue that we double the degree of each node for our scheme to work well, but this is a central premise of this work. We need to consider

[9] One can view an Omega/Flip simply as a bidirectional Omega.

both the protocol and the topology for high performance, and we should not design the one independent of the other[10]. The Omega/Flip also benefits the performance of STEM, since it reduces the average distance in the network by half.

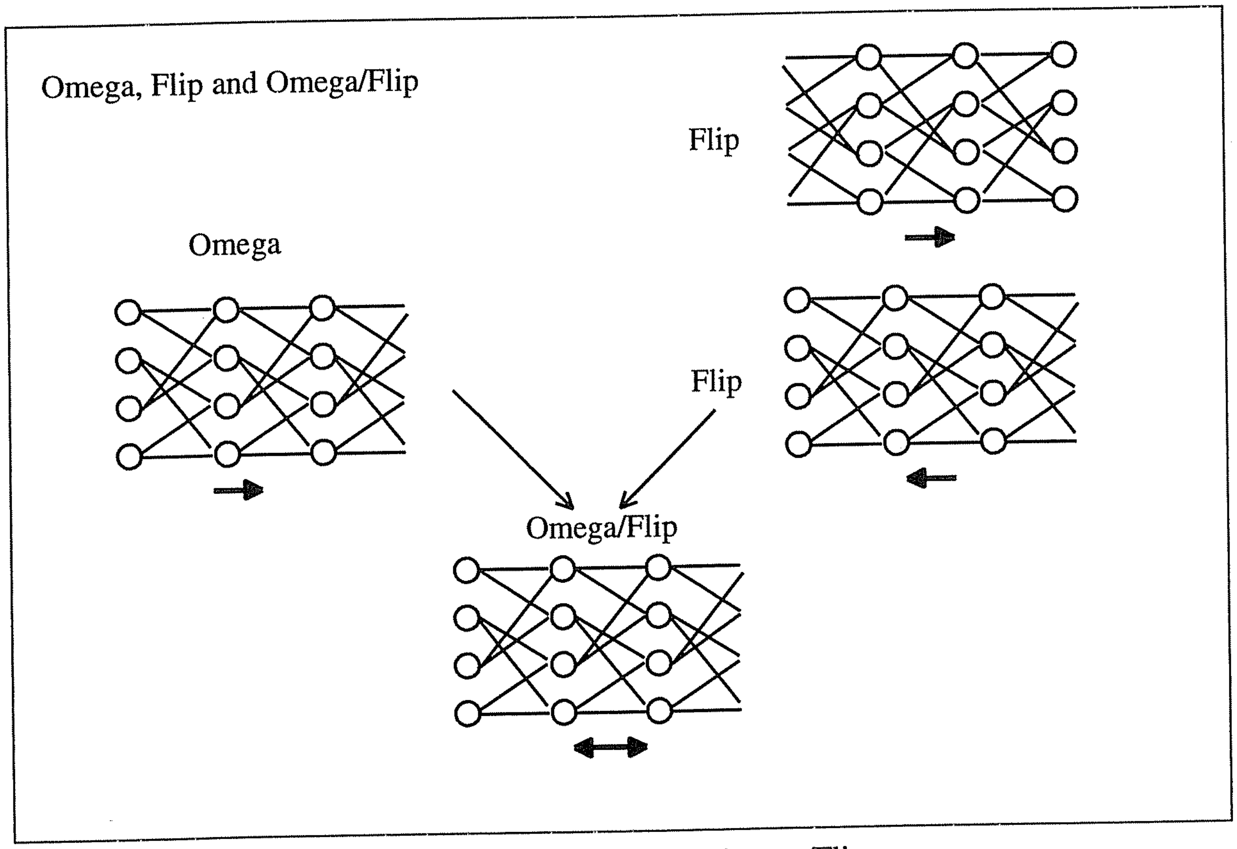

Figure 5. Omega, Flip and Omega/Flip

Figure 6 shows a 4 stage, 4 ring per node (2 for the Omega and 2 for the Flip), Omega/Flip network. The structure of the our trees in this topology is depicted in the two trees below the network. These two trees are the worst and best case, in terms of height (longest path). In Omega/Flip networks have as many hierarchical levels as stages. For *S* stages the height is given by the formula:

$$(2 \times S) - 1 \leq \text{LongestPath} \leq \frac{S \times (S+1)}{2}$$

For the best balanced STEM tree we have a height of:

$$2 \times [\log_2(S \times R^{S-1}) - 2] = 2 \times (S-1) \times \log_2(S \times R) - 2$$

[10] In effect, what we are saying is that, there are appropriate and inappropriate topologies for our schemes. What we are trying to achieve, is to define a protocol that fits good topologies, which are likely candidates for large-scale SCI systems.

In figures 7 through 11 we plot the heights of the best (min) and worst (max) tree of our proposal and the height of the corresponding STEM tree (stem). Each figure corresponds to a network with a specific number of rings per node. The description for each figure gives the number of rings for an Omega or Flip and inside the parentheses the number of rings for an Omega/Flip. The *x* axis is system size. (The discrete points in the *x* axis correspond different values for the number of stages.) The *y* axis represents the height of the trees. The lines labeled *stem* represent the height of balanced STEM trees. The lines labeled *min* represent the best trees of our proposal and the lines labeled *max* represent the worst.

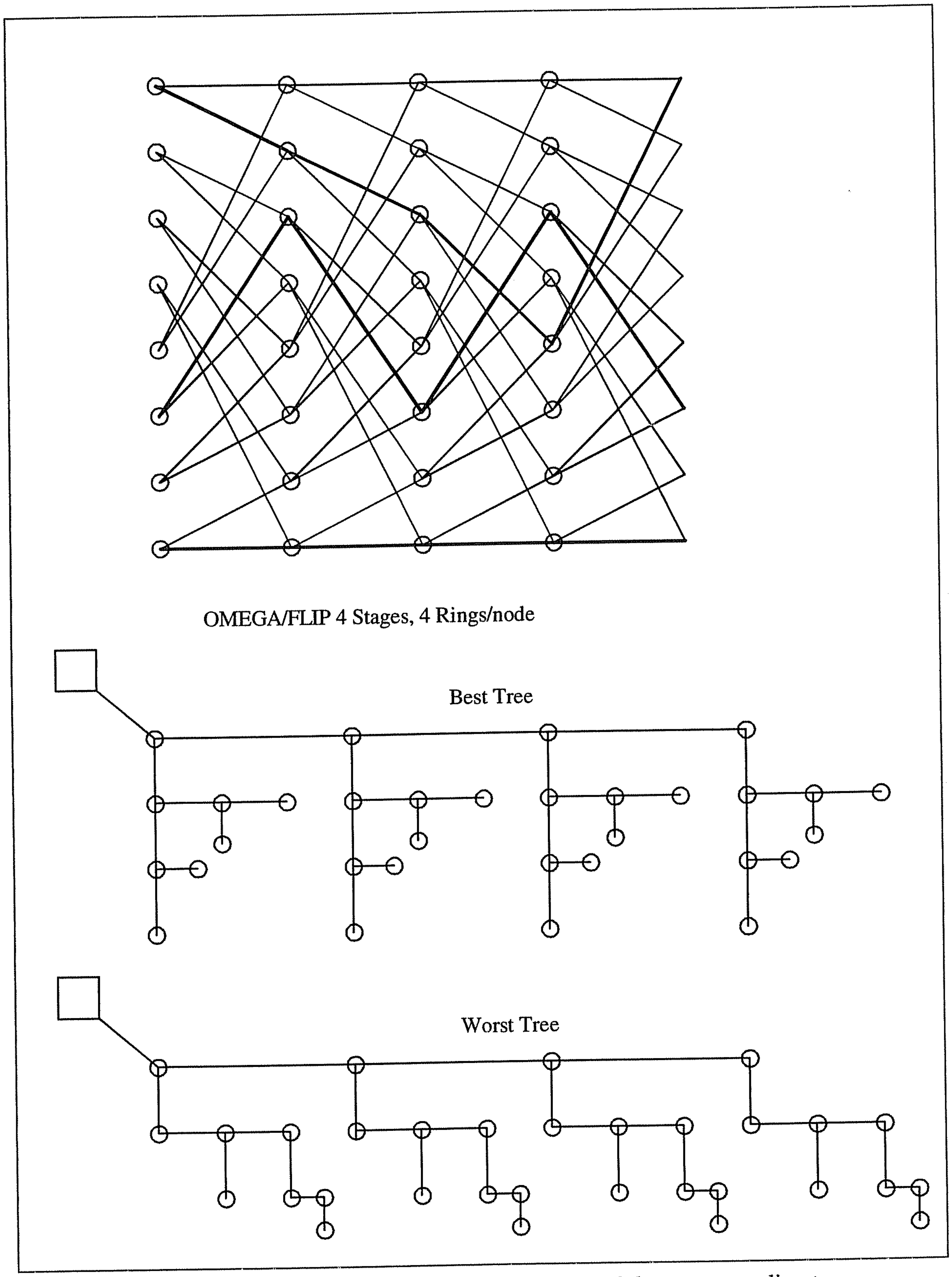

Figure 6. Omega/Flip (2 Stages, 4 rings per node) and the corresponding trees.

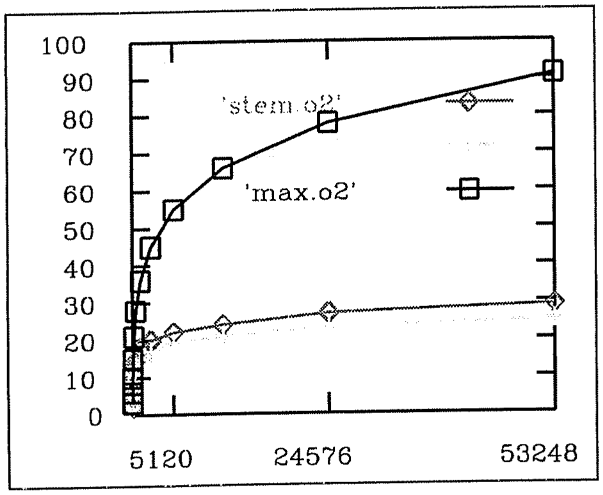

Figure 7. Omega/Flip 2(4) rings per node.

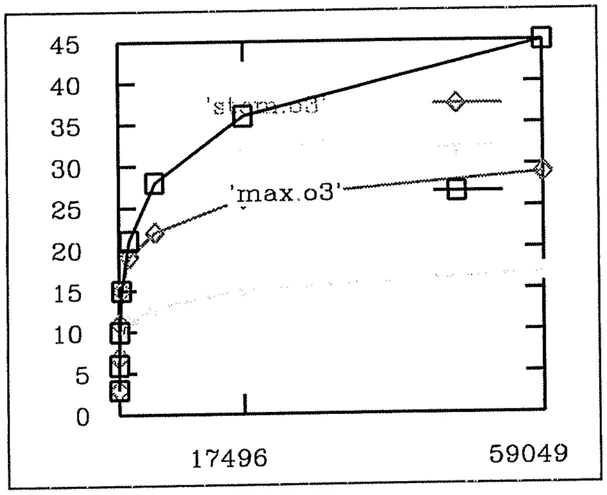

Figure 8. Omega/Flip 3(6) rings per node.

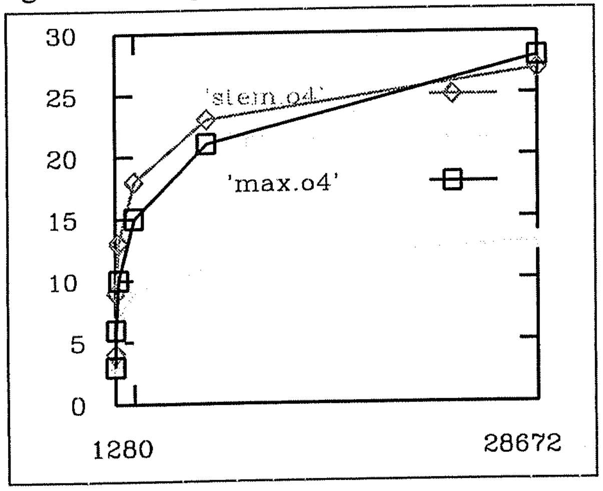

Figure 9. Omega/Flip 4(8) rings per node.

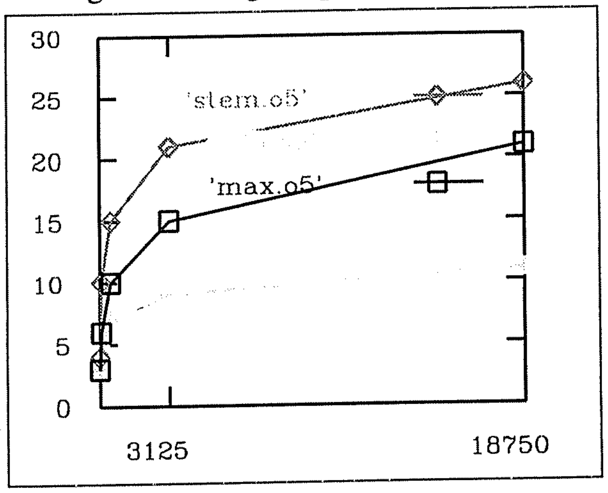

Figure 10. Omega/Flip 5(10) rings per node.

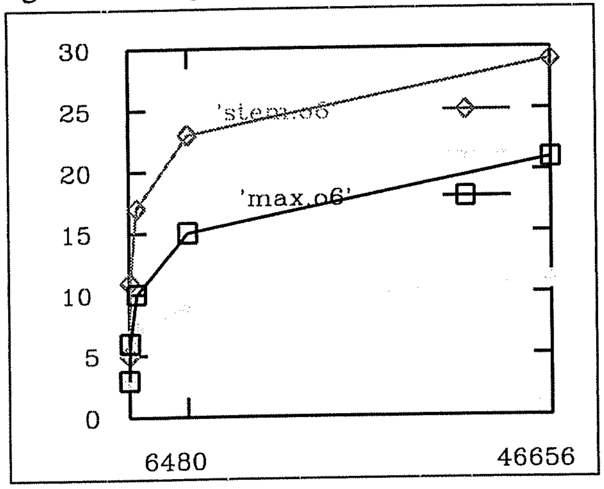

Figure 11. Omega/Flip 6(12) rings per node.

4. Tag Storage

Two kinds of information are contained in the tag information of a cache memory. First is the data necessary to identify uniquely the address of the data in the cache, and its validity. This information is required for any cache, even a uniprocessor. Second is data that is related to maintaining the consistency of the data, which we will refer to generically as "directory information." In a snooping cache, this directory information may be as little as the state of the line, though a separate tag memory is often maintained to permit simultaneous access from the processor and the bus. In a more general, directory-based scheme, this directory information also includes indications about what other nodes in the system have copies of the data. The basic SCI protocol, for example, includes forward and backward pointers, indicating up to two other nodes that have copies of the line. Though this data is usually associated with a cache, it is in fact directory information. The concept of distributed directory information is contained both in snooping protocols and in SCI. In both cases, critical information about where copies of the line reside are distributed along with the cached copies.

In general, directory information can be distributed around the system, including nodes where it is neither cached nor present in memory. In particular, it is not necessary that a valid copy of the cached data be present at a node containing directory information. At such a node, the problem then is to store the information in a way that it can be accessed and updated as needed. Depending on the scheme used, it may be desirable that the amount of directory information stored vary over time. The problem of dynamically allocating space for directory information was nicely solved by Simoni and Horowitz [8] for a conventional directory scheme, where space is allocated at a centralized location. We propose that such techniques might also be applied at other, critical points in the system, where directory information might be strategically placed to minimize the distance necessary to acquire the information. Thus the problem is one of allocating storage for directory information dynamically, and independent of cached copies, which might or might not be present.

A plethora of techniques might be used to store directory information. Among those techniques are to maintain a linked list, as suggested by Simoni and Horowitz. Another technique is to store the data associatively, using conventional cache techniques of set-associativity, but eliminating the cache data. Finally, it may be useful to store information about nearby nodes that contain data using boolean variables to indicate which nodes contain data. Techniques similar to those proposed for pruning caches [9] can then be applied to store the critical directory information. In general, there are many possibilities.

In this section we discuss various aspects of tag storage. We first present a simple implementation of the tags needed for k-ary trees. We then discuss how, by using multiple types of tags, we decouple the data and the corresponding directory information of the tag. We continue with alternative implementations of tags with multiple pointers in the form of linked lists or trees and finally we examine how, by taking into account geographical locality, we can minimize storage requirements with *short* pointers and bit maps.

4.1 Single Tag with Multiple Pointers

We defined previously, the degree *k* of the *k*-ary trees to be one less than the number of rings *R*, connected to a node. Since each node belongs to a list in a hierarchical level and can have up to *R - 1* child lists, we require at minimum *R + 1* pointers : a *forwid*, a *backid*, and *R - 1 childid* pointers.

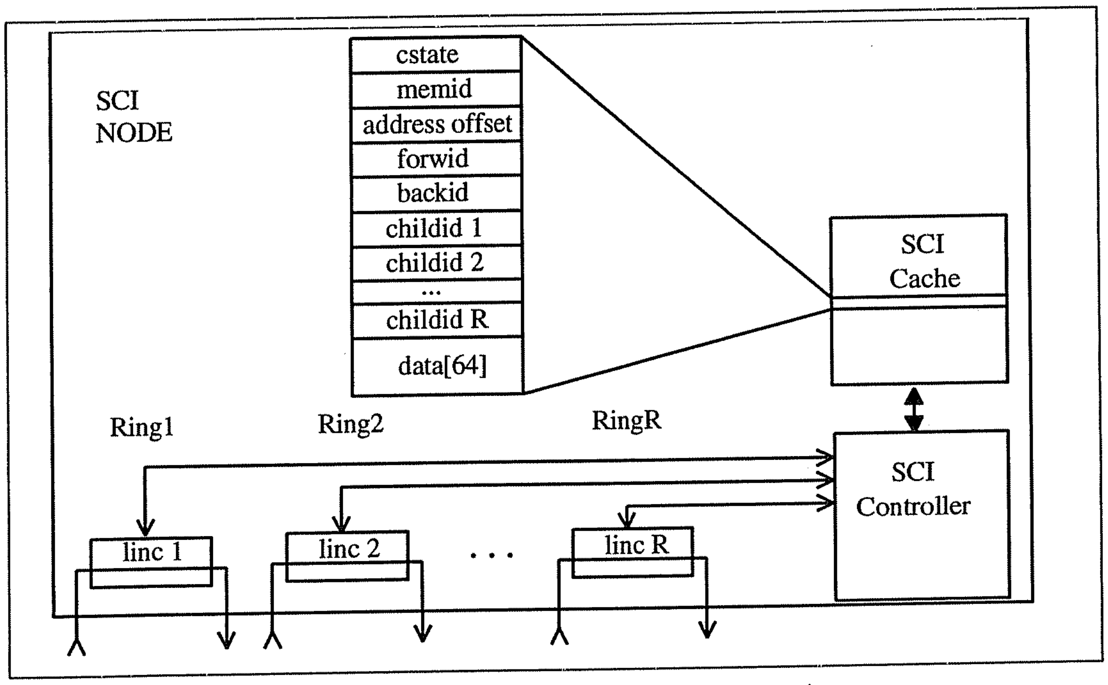

Figure 12. Pointers for simple implementation.

A simple minded implementation with *R* + 2 pointers is shown in figure 12. Each child pointer handles the child list inside the corresponding Ring_n (accessed by linc_n). Since both *forwid* and *backid* will be used to connect to the high-level list in one ring, it follows that the corresponding child pointer for this ring will be left unused. In this way we avoid doing any dynamic binding between the pointers and the rings. We can distinguish messages that refer to *forwid* and *backid* and we know which *childid* pointer we will use, by knowing the ring that carried the message.

A more cost effective implementation is shown in figure 13. Here we exploit the fact that no more than *(R+1)* pointers are ever concurrently active. Thus, we only need the minimum number of pointers required *(R+1)* plus a small extra pointer for binding. In order to do dynamic binding, we do not define either the *forwid* or *backid* pointer. Instead, any *childid* can assume the role of the *forwid* (*backid*). A new small pointer for binding, *forwid_ptr* (*backid_ptr*), designates the child pointer that currently assumes the role of the *forwid* (*backid*) pointer. The storage requirements for *forwid_ptr* (*backid_ptr*) are very small compared to any other pointer. This is because it can only take values between *1* and *R*.

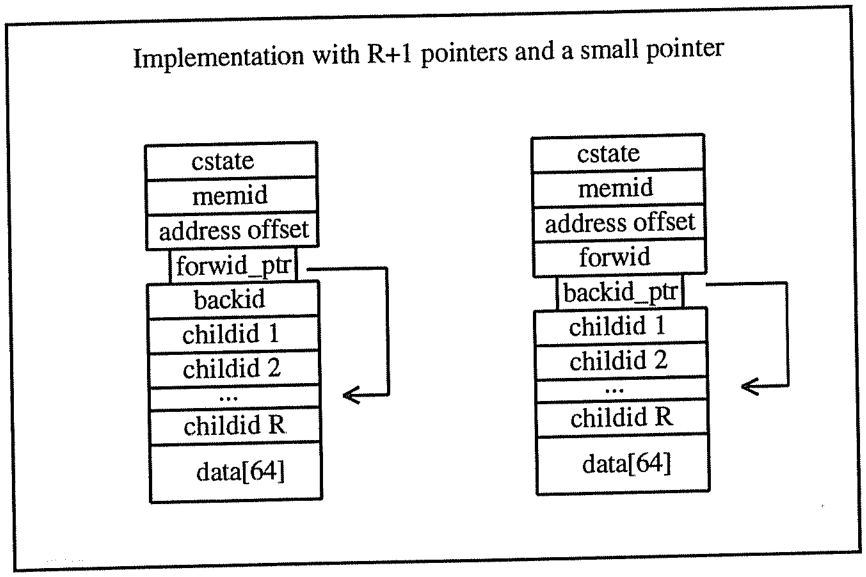

Figure 13. *R+1* pointers

4.2 Multiple Types of Tags

It is clear from the two examples given in previous sections that in hierarchical trees, an agent node might be in the tree without the agent's processor accessing the cache line. The agent has a cached copy of the line just for the benefit of its children. This may create many conflict misses because local processor accesses might conflict with the tree cache line. This situation would result in many tree rollouts for the agent nodes. Tree rollouts are generally very expensive and we would like to avoid them. To avoid this we allow tags to be cached in agents without the corresponding data. Thus we are able to have pointers in an agent without competing with the agent's processor for data storage. In order to cover the possible cases, we define four types of tags summarized in table 2.

Types	Description	Pointers	Data
Type 1 SCI	Standard SCI	2	Yes; accessed locally
Type 2 Tree No Data (TND)	Tree	multiple	No
Type 3 Tree Optional Data (TOD)	Tree	multiple	Yes; can be discarded
Type 4 Tree Data (TD)	Tree	multiple	Yes; accessed locally

Table 2. Types of Tags

When a local access results in choosing a victim line with its associated tag we take different action depending on the type of the tag. Table 3 summarizes the actions in this situation. The goal is to keep tree tags (with multiple pointers) cached in an agent despite a conflict for the data line. When we eventually have to choose a tree tag to replace, because of the tag storage replacement algorithm, we will have a tree rollout as described in a later section. Notice that because a Type 2

TND tag does not have an associated data line it is never chosen for replacement due to local accesses. It can only be chosen by the replacement algorithm for the tag storage. Additionally we do not allow tags of Type 2 TND and Type 3 TOD to exist without any children.

	Data line Victim	Action
Type 1 SCI		Replace (SCI rollout)
Type 2 TND	Never	None
Type 3 TOD		Convert to Type 2 TND
Type 4 TD		Convert to Type 2 TND

Table 3. Replacement of data line and actions on tags

Finally notice that when we use multiple types of tags there is not 1-to-1 correspondence between the tags and the data lines. In Appendix II we give some further guidelines for replacement of different types of tags.

4.3. Multiple Tags

In order to implement k-ary trees we have described tags with multiple child-pointers. We can also implement the tags as multiple tag entries in an agent. The central idea is to have multiple tag entries (instead of one) to implement a k-ary tree. This idea is described by R. Simoni and M. Horowitz [1]. The multiple tag entries (instead of one with many pointers) can be linked together and form a linear list. Alternatively they can be unrelated (not linked) in which case they can be searched and found individually.

4.3.1. Multiple Linked Tag Entries

There is a whole range of implementations with varying number of pointers per tag entry. One end of the spectrum is to have all necessary pointers to implement k-ary trees in only one tag. Moving away from this end, we can have tag entries with smaller number of pointers so we would need to have two or more of them, linked together, to implement k-ary trees. Another option is to have tags with different number of pointers in a linked list (Figure 14 A and B). For example the first tag entry can have three pointers and the rest only two. Two interesting points in this spectrum are the case with three pointers per entry, which is a STEM requirement, (Figure 14 C) and the case of just two pointers per entry, which is a base SCI requirement (Figure 14 D). Three pointers per tag entry are enough for a doubly linked list of tags where each tag has a child pointer. Two pointers per tag entry are sufficient for a singly linked list of tags where again each tag has a child pointer. Of course we are not constrained in linked lists and we can easily implement the tags as trees. One simple example is shown in figure 14 E. In order to access all the tags we have to find only the first tag entry and then follow the pointers. This means that the SCI address has to be kept only for the first tag entry in order to be matched. Traversing these internal lists is not expected to be a performance problem because a) they are internal to a node, b) the maximum length is not great (the constant k is usually small) and c) their average length is usually very small.

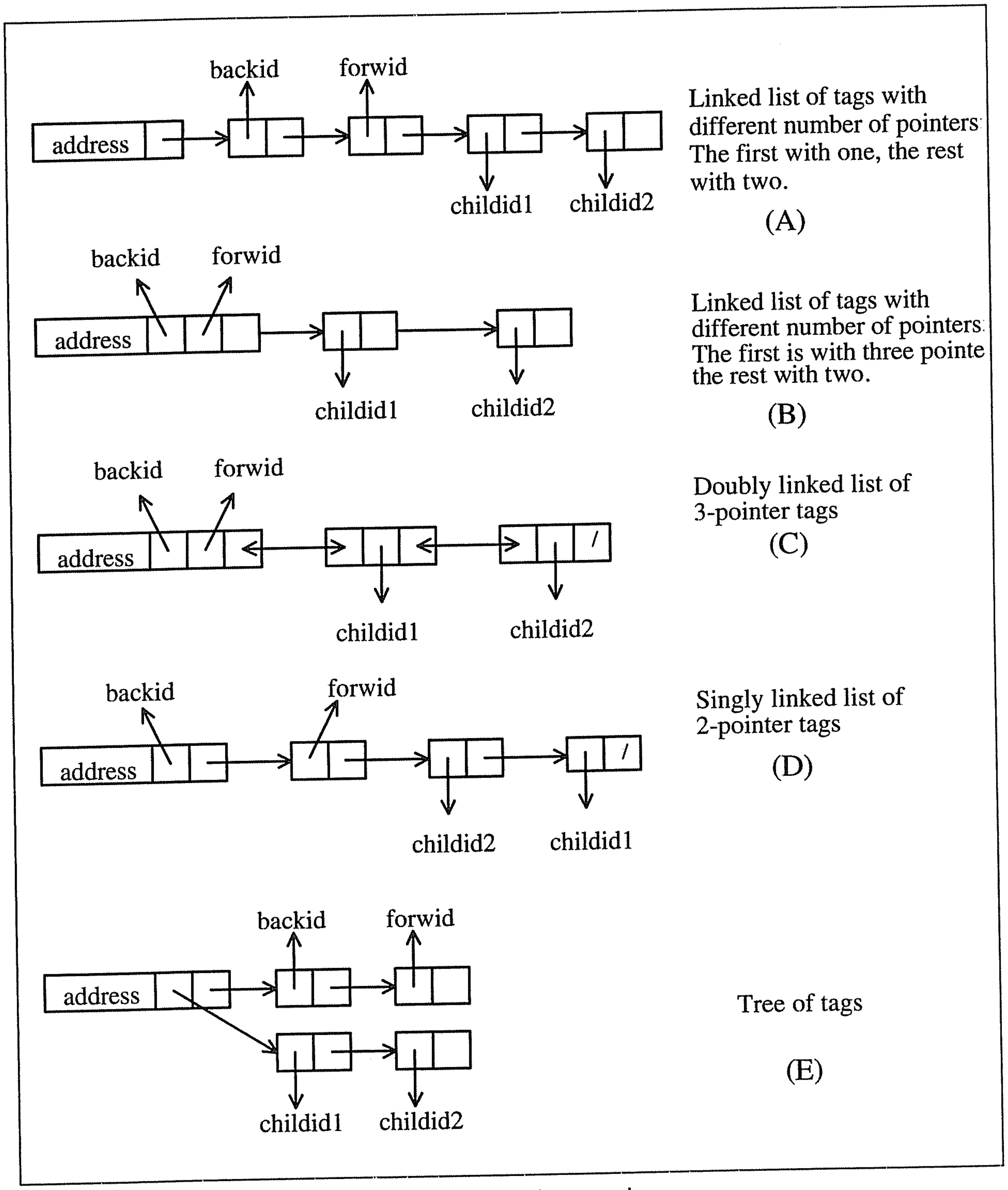
backid
forwid
address
childid1
childid2
Linked list of tags with different number of pointers: The first with one, the rest with two.
(A)
backid
forwid
address
childid1
childid2
Linked list of tags with different number of pointers: The first is with three pointe the rest with two.
(B)
backid
forwid
address
childid1
childid2
Doubly linked list of 3-pointer tags
(C)
backid
forwid
address
childid2
childid1
Singly linked list of 2-pointer tags
(D)
backid
forwid
address
childid1
childid2
Tree of tags
(E)

Figure 14. Linked tag entries.

4.3.2. Multiple Tag Entries

Multiple tag entries can also be used without linking them together as long as we can access all of them. This means that the SCI address has to be kept for each individual tag entry for matching purposes. Again there is a spectrum of implementations depending on the number of pointers per tag entry. Standard SCI tags with two pointers can be used. One of the tags would implement a *forwid* and *backid* pointers to connect to the high level list and each of the other tags can implement one or two child pointers. We can also use just one pointer per entry and allocate enough entries for a *forwid* pointer a *backid* pointer and the required number of child pointers. This in effect is equivalent to providing multiple memory directory entries in an agent that represents the remote memory to its children. To implement such a scheme with multiple unlinked tags we can use a set associative cache with associativity equal to or greater than the maximum number of tag entries needed to implement k-ary trees. For example, to implement 4-ary trees with standard SCI tags (unlinked), at minimum three tags are needed: one to implement the *forwid* and *backid* pointers and two for the four child pointers. A 3-way (or greater) set associative cache could then be used. A set of this cache can hold three tag entries for the same address or three tags corresponding to different addresses (or any combination in between). In figure 15 we give an example of a 5-way set associative cache holding in one set three tags for one address (six pointers in total) and two tags (shaded) of another address. A disadvantage of using unlinked tags is that each one must carry the SCI address so it can be matched and accessed independently.

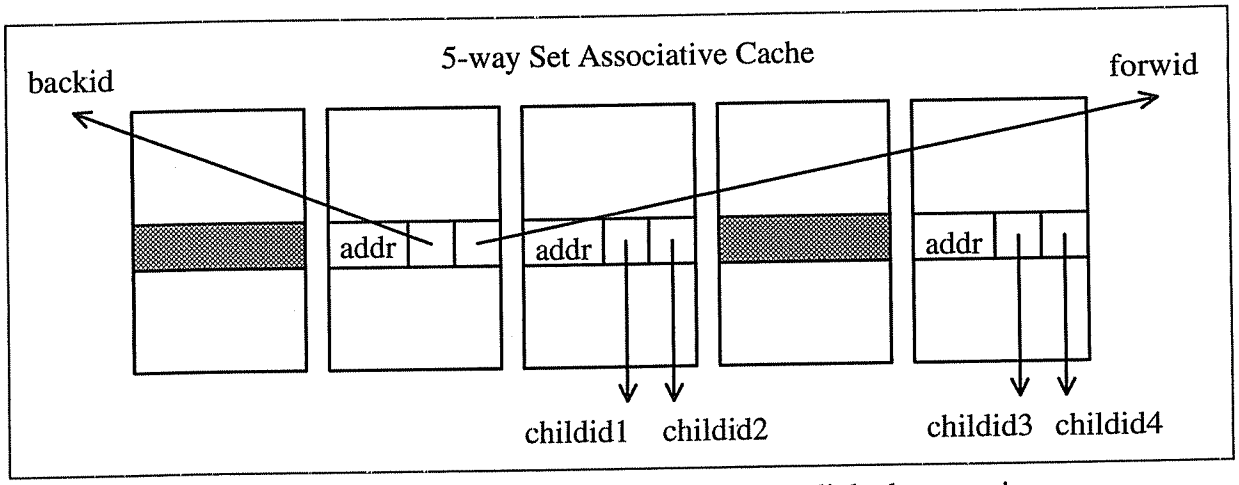

Figure 15. Set associative cache with multiple unlinked tag entries.

4.3.3. Linked Tags with Address

We can also combine the previous two methods so we can have a linked list of tags where each tag carries the SCI address. In this way we can use a set associative cache to implement the linked list. The associativity of the cache must be equal to or greater than the number of tags we need to implement k-ary trees. The set associative cache offers transparent buffer management, automatically allocating and deleting tags according to demand. Interesting cases include tags with three and tags with two pointers as described previously in section 4.3.1. In figure 16 we show an example of a linked list of three 2-pointer tags inside a 5-way set associative cache. The two remaining tag entries of the set can be used for different addresses, possibly for a different linked list. Having the list linked helps in the rollout algorithm as discussed in a subsequent section.

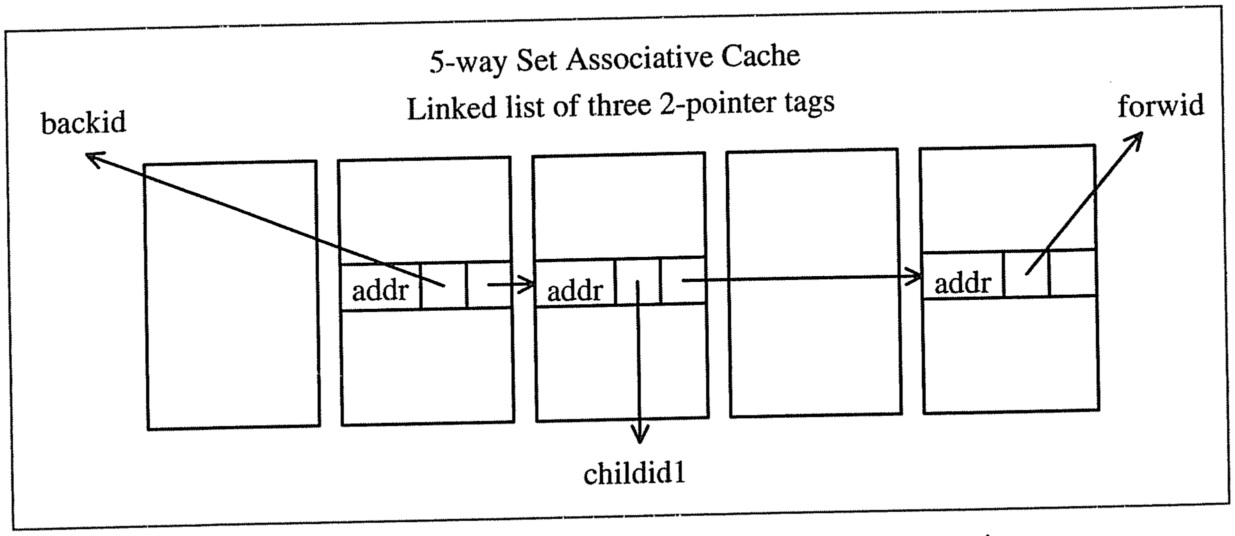
5-way Set Associative Cache
Linked list of three 2-pointer tags
backid
forwid
addr
addr
addr
childid1

Figure 16. Set associative cache with multiple linked tag entries.

In general multiple tags (either linked or unlinked) offer the opportunity to allocate pointer storage on demand instead of always allocating pointer storage for the maximum number of pointers needed. Notice here that in hierarchical trees only a few nodes usually need the maximum number of pointers. Usually, nodes in the lowest level of hierarchy do not have any children and as we go up in the hierarchy the number of children increases up to the maximum in the highest level.

The multiple Types of tags (Type 1 SCI, Type 2 TND, Type 3 TOD, Type 4 TD) that were described previously and the multiple tags in an agent are largely orthogonal. Using multiple tags (linked or unlinked) to achieve the same effect as a tag with many pointers is orthogonal to whether the data is also present in an agent.

4.4. Short Pointers

In our k-ary trees, geographical locality plays an important role. If all the nodes are in their proper position in the tree, we know that pointers do not point very far. In fact a pointer associated with a ring, will only point to nodes on that ring! It is therefore possible, for every node to have aliases for the other nodes on its rings and point to them using their alias id. Such a pointer would require much less storage than a system-wide pointer. We call the pointers, that are the alias ids, *short* pointers, while we call the system-wide pointers, *full* pointers.

In the figures 17 and 18 we assume a hypothetical system that has tens of thousands of nodes. Each node is connected to three rings and each ring has four nodes. The example shows node 0 as the box in the left side. The rings are the ellipses connected to the node 0. The small circles represent other nodes on the rings. The numbers near each circle are the node SCI ids. We assume that the high level list is on ring 1 (the *forwid* and *backid* pointers point to nodes on this ring). It is obvious that, to implement a tag with full pointers we need $(4 \times 16) + 2 = 66$ bits (three rings, four pointers plus 2 extra bits for a *forwid_ptr* or *backid_ptr*) while we need only $(4 \times 2) + 2 = 10$ bits with short pointers. This is $\frac{1}{3}$ of the storage needed just for the *forwid* and *backid* of the basic SCI protocol! In this fictional example the short pointers can be up to 7 bits long and still fit in the space of the standard *forwid* and *backid.*

The only extra hardware needed to support short pointers, are the Alias Tables. For each ring connected to a node, we need an Alias Table. Each table must have as many entries as the nodes on the corresponding ring[11]. Note, there are only a few such tables per node whereas there are many pointers per cache line.

Although in the above example we talked about a hypothetical system we now prove that for the k-ary n-cube topology the storage for short pointers is always less than or equal to the storage for a full *forwid* and a full *backid*:

A k-ary n-cube has k^n nodes. A full *forwid* and full *backid* require $(2 \times \lceil (\log_2(k^{(n)})) \rceil)$ bits. For our tree we need n+2 short pointers (n rings per node) of $\lceil (\log_2(k)) \rceil$ bits each (k nodes per ring).
$2 \times \lceil \log_2 k^{(n)} \rceil \ge (n+2) \times \lceil \log_2(k) \rceil \Leftrightarrow 2 \times n \times \lceil \log_2(k) \rceil \ge (n+2) \times \lceil \log_2(k) \rceil \Leftrightarrow 2 \times n \ge n+2 \Leftrightarrow$
$\Leftrightarrow n \ge 2$
If n is less than two, the above result excludes one dimensional cubes (i.e. a single ring), on which our proposal is not applicable anyway. For every other combination of k and n we can implement k-ary trees in just the space allotted for the *forwid* and *backid* of the standard SCI protocol. This is also true for the Omega/Flip topologies (except for some uninteresting cases).

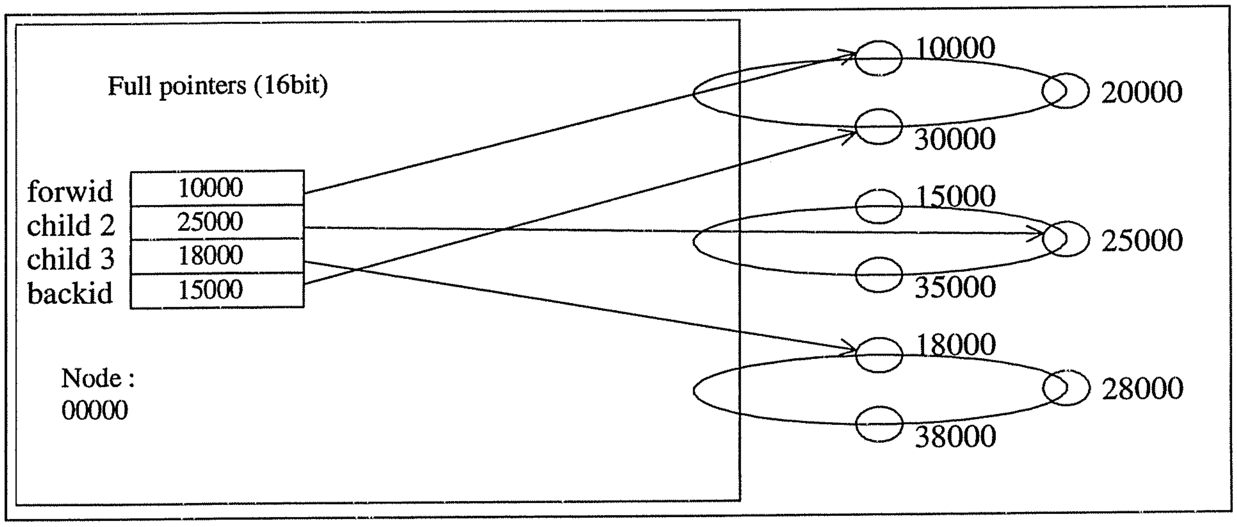

Figure 17. Using full pointers.

[11] We expect this number to be generally small. In a single ring we do not expect more than 16 nodes. For large systems (multiple rings) we expect it to be at most 256.

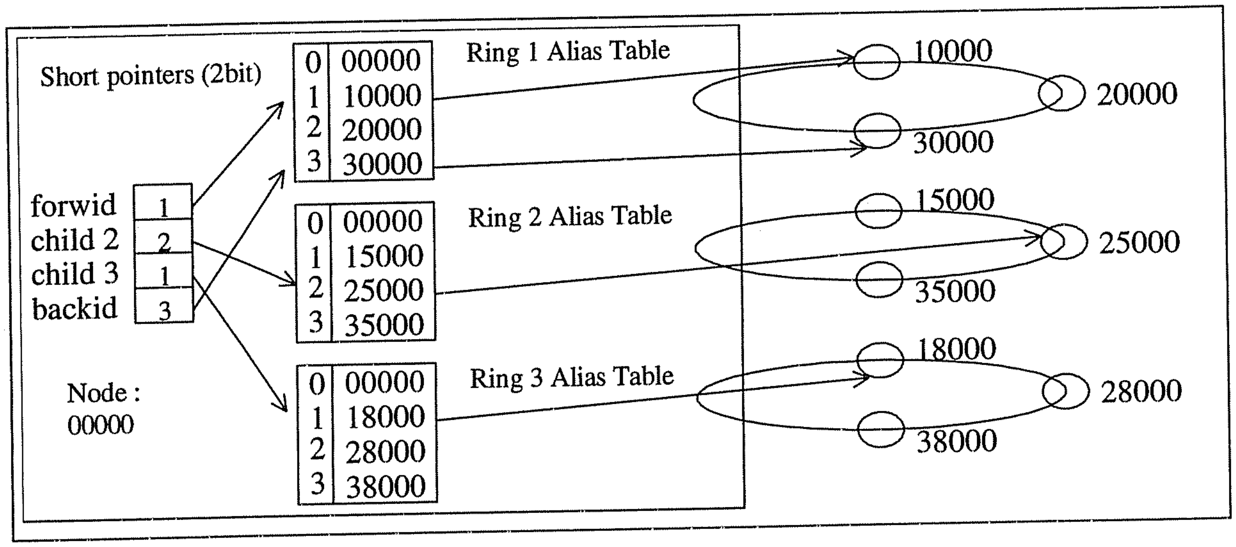

Figure 18. Using short pointers and Alias Tables.

In Appendix I we describe how short pointers can be used in fixed size tags. The main idea is to have multiple short pointers for performance and switch to a few full pointers for flexibility.

4.5. Agents

The agents play a central role in our schemes. They intercept other nodes' requests and cache tree tags and possibly data. In order to use a standard SCI switch in our schemes we require some additional functionality. Specifically the agent (switch node) must be able to:

1. Intercept special requests used for the creation of the trees[12]. The agent must recognize these requests by examining the command field (second symbol) of an SCI request. Instead of routing these requests normally, switching them from ring to ring, the agent delivers them to the local SCI controller.
2. Be able to annotate an intercepted special request with the ring id it came from. The agent delivers the special request to the local SCI controller and also supplies the ring id of the request. In this way the local SCI controller will be able to choose the appropriate pointers of a tree tag to manipulate.

12 See section 5 for a discussion on special *mread* requests.

5. Protocols

Current trends in research for distributed shared-memory parallel machines, favor the use of specialized protocols for different classes of shared data [11]. Several classes of shared data have being identified, including migratory, read-only, etc. [11,12]. This approach to shared memory assumes that the programmer and/or the compiler can identify the type of a datum. Subsequently, the underlying hardware or software layer is instructed to apply the proper protocol for the datum. This approach can provide greater performance and we believe that we should have different kinds of requests that specify construction of linear sharing lists, construction of sharing trees, construction of sparse sharing trees (section 5.1.3), pairwise sharing, and QOLB. We note that STEM already has implicitly assumed that a node must ask for a non-modifiable cache copy for combining to work.

5.1. Construction

In this section, we describe the construction of the k-ary tree. We briefly outline a method to construct a sparse tree when there is not enough sharing to justify the overhead of the full tree.

A node that wishes to read a remote line, that is expected to be widely shared, sends a special *mread_tree* request towards the node that has the data in its local memory (home node). The request is intercepted at the first agent where it is taken off the ring. The intercept causes a local lookup to find information about the requested line. If the lookup results in a hit, the requesting node is instructed to prepend to the appropriate child list[13]. The node will either get the data from the previous head of the child-list or the agent. If the lookup results in a miss, the agent sends its own request for the line towards the home node. The agent does not forward intercepted requests but rather it replaces all of them by its own request. The requesting node is instructed to prepend to the appropriate child list. As soon as the interceptor gets a copy of the line it will pass it to the waiting node.

In the time it takes for an agent to get its own copy of the data, new requests may arrive. Each of the new requesting nodes is instructed to prepend to the appropriate list and wait for the data. We have three options of how to distribute the data to a list where all nodes are waiting for the agent to get the line:

1. As is currently defined in SCI. A node asks the agent (memory in SCI) for the data. The agent (or memory in SCI) instructs the node to prepend to the child-list and ask the data from the old head. The very first node that requested the data is the waiting tail of the child-list. It will eventually get the data from the agent. The agent may keep a temporary pointer to the first requesting node (the storage for the data can be used for temporary pointers).
2. If a requesting node prepends to a child list, where the old head is waiting for the data, then it assumes responsibility to forward the data. When the agent gets the data it will pass it to the waiting heads of its child-lists. The heads will then forward the data towards the

13 The pointer that corresponds to the ring where the request came from.

waiting tails. This distribution happens in the opposite direction of the one described above (standard SCI).

3. Broadcast the line on the ring. This is a broadcast confined in a single ring. All nodes that are waiting will receive the broadcast and consequently the data. Here we may need to define a confirmation protocol to make sure that all waiting nodes actually got the data.

In the case of a miss in the agent, its own request will be treated the same way in the next agent. Eventually the agent just before the home node will get the data from memory and it will pass it to its child lists.

Construction of the tree when all nodes simultaneously read a line, after a global barrier, is quite fast. All the linear lists can be constructed in parallel. Copies of the cache line are distributed down the tree concurrently with list construction.

For the k-ary n-cube example the construction of the full tree takes roughly as much as the maximum of the time to create a linear list of length k and the time to distribute the data throughout the tree. The time it takes to distribute the data (assuming unit time the time it takes a message to traverse a link) is proportional to the height of the tree: $n \times (k-1)$. If we assume two pairs of request-response per node joining the linear list, we can assume a time of $(k \times 4)$ for the construction of a list with k nodes. The time it takes to create the full tree is $\max(4 \times k, n \times (k-1))$. As for the Omega/Flip with N stages, with the same reasoning, we get a minimum time of $(4 \times N)$ and a maximum of $\max[4 \times N, (\frac{N \times (N+1)}{2})]$.

5.1.1. Construction with Multiple Types of Tags

In this section we describe in more detail the construction taking into account the four types of tags. We remind here that the types of tags we have defined: Type 1 SCI, Type 2 TND, Type 3 TOD, Type 4 TD.

When a node requests a line and its request is intercepted in an agent we can have either a miss or a hit. In case of a miss the agent can allocate either a Type 2 TND or a Type 3 TOD tag. When the data returns the agent will forward it to the waiting tails of its newly created child-lists. If the allocated tree tag was of Type 3 TOD the data is also stored in the agent to service subsequent requests more efficiently. Figure 19 A shows the case of a miss in the agent.

In case of a hit the tag in the agent can be of any of the four types. The agent services the request differently depending on the type of the tag. The four cases are described below:

1) Hit on Type 1 SCI tag. In this case the agent has to convert the SCI tag to a tree tag. Since the data is used locally the tree tag will be Type 4 TD. The agent replies immediately to any request by putting the requesting node head of the appropriate child-list and passing it the data. Figure 19 B shows this case.

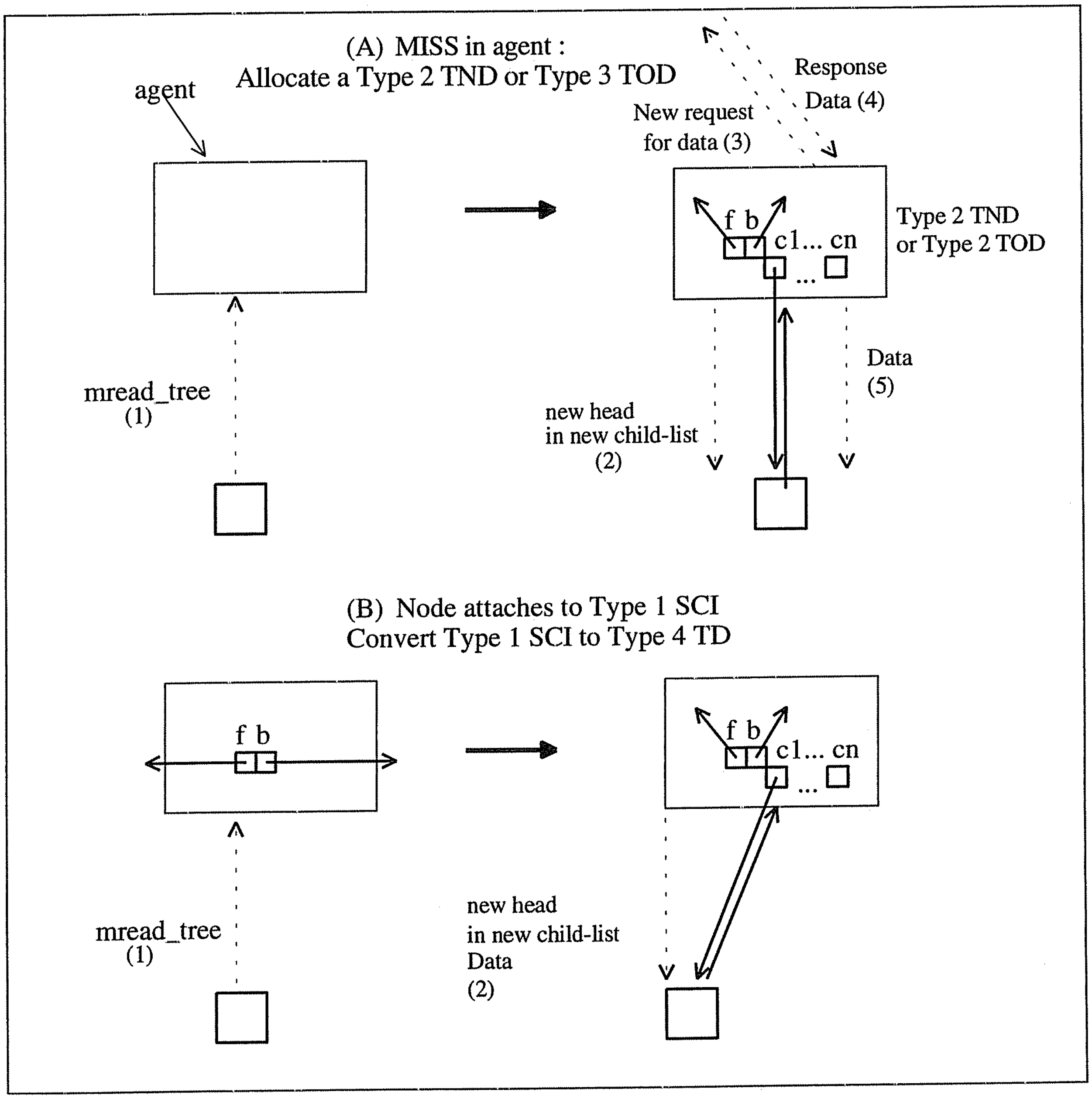
(A) MISS in agent :
Allocate a Type 2 TND or Type 3 TOD
agent
Response Data (4)
New request for data (3)
Type 2 TND or Type 2 TOD
f b
c1... cn
mread_tree (1)
new head in new child-list (2)
Data (5)
(B) Node attaches to Type 1 SCI
Convert Type 1 SCI to Type 4 TD
f b
c1... cn
mread_tree (1)
new head in new child-list Data (2)

Figure 19. Miss in agent and Hit on Type 1 SCI tag.

2) Hit on Type 2 TND tag. Because the tag is already a tree tag there is no need to allocate a new one. However the data are not local so the requests cannot be satisfied with a single response. Since Type 2 TND tags are not allowed to exist without children there is at least one child-list where the data can be found. The agent instructs the requesting node to prepend to the appropriate child list. If the requesting node prepends to an existing child-list then it gets the data from the old head of the child-list. If the requesting node becomes the first head on a new child-list the agent will acquire the data from one of its other child-lists and supply it to the waiting head. Figure 20 shows how a node attaches to a Type 2 TND tag.

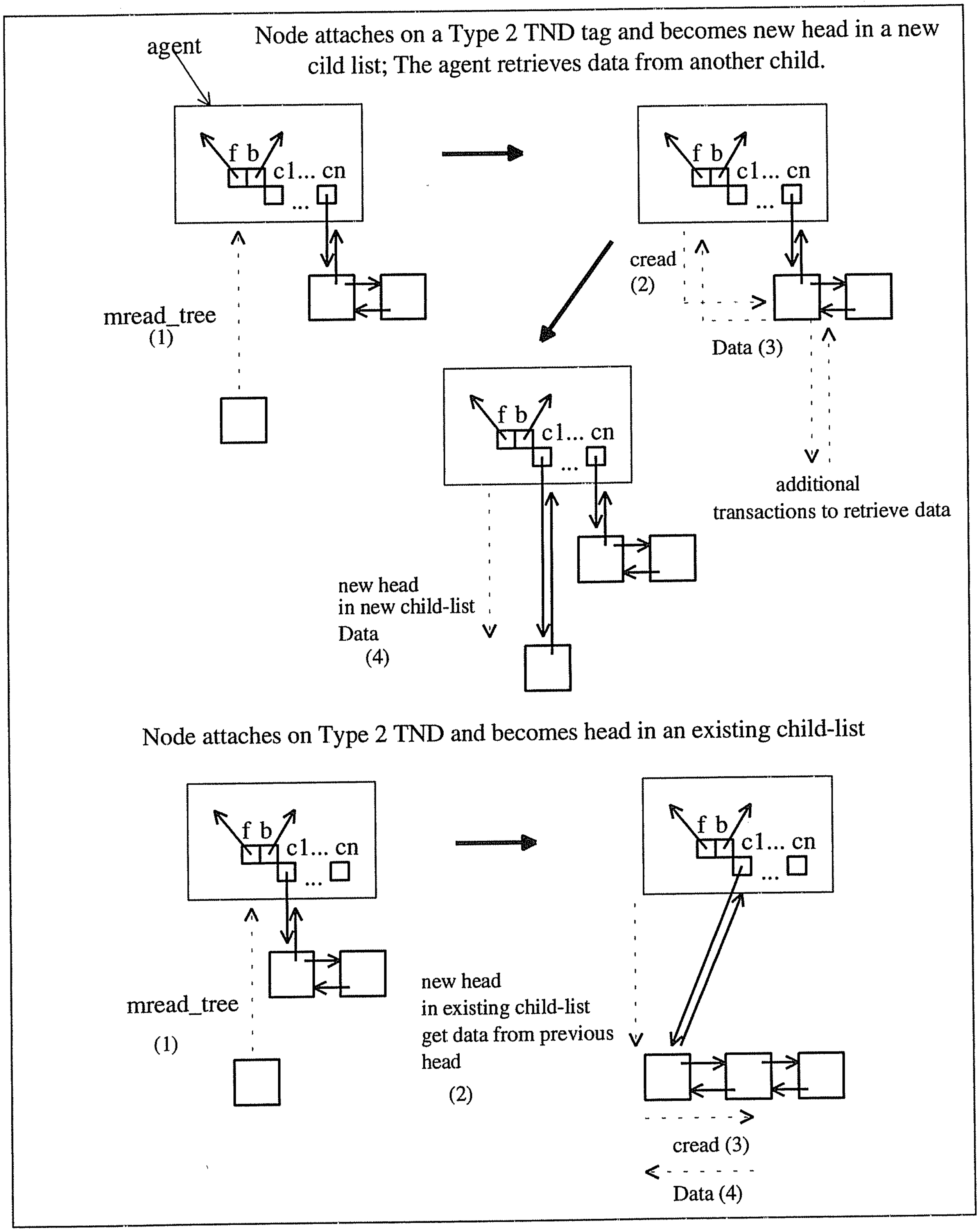
agent
Node attaches on a Type 2 TND tag and becomes new head in a new cild list; The agent retrieves data from another child.
f b
c1... cn
...
mread_tree
(1)
cread
(2)
Data (3)
additional
transactions to retrieve data
new head
in new child-list
Data
(4)
Node attaches on Type 2 TND and becomes head in an existing child-list
mread_tree
(1)
new head
in existing child-list
get data from previous
head
(2)
cread (3)
Data (4)

Figure 20. Hit on Type 2 TND tag in the agent.

3) Hit on Type 3 TOD. In this case the requesting node is made head of the appropriate child-list and it is given the data in a single response. Figure 21 shows how nodes attach to Type 3 TOD tags.

4) Hit on Type 4 TD. The same as a Type 3 TOD hit. Figure 21 shows how nodes attach to Type 4 TD tags.

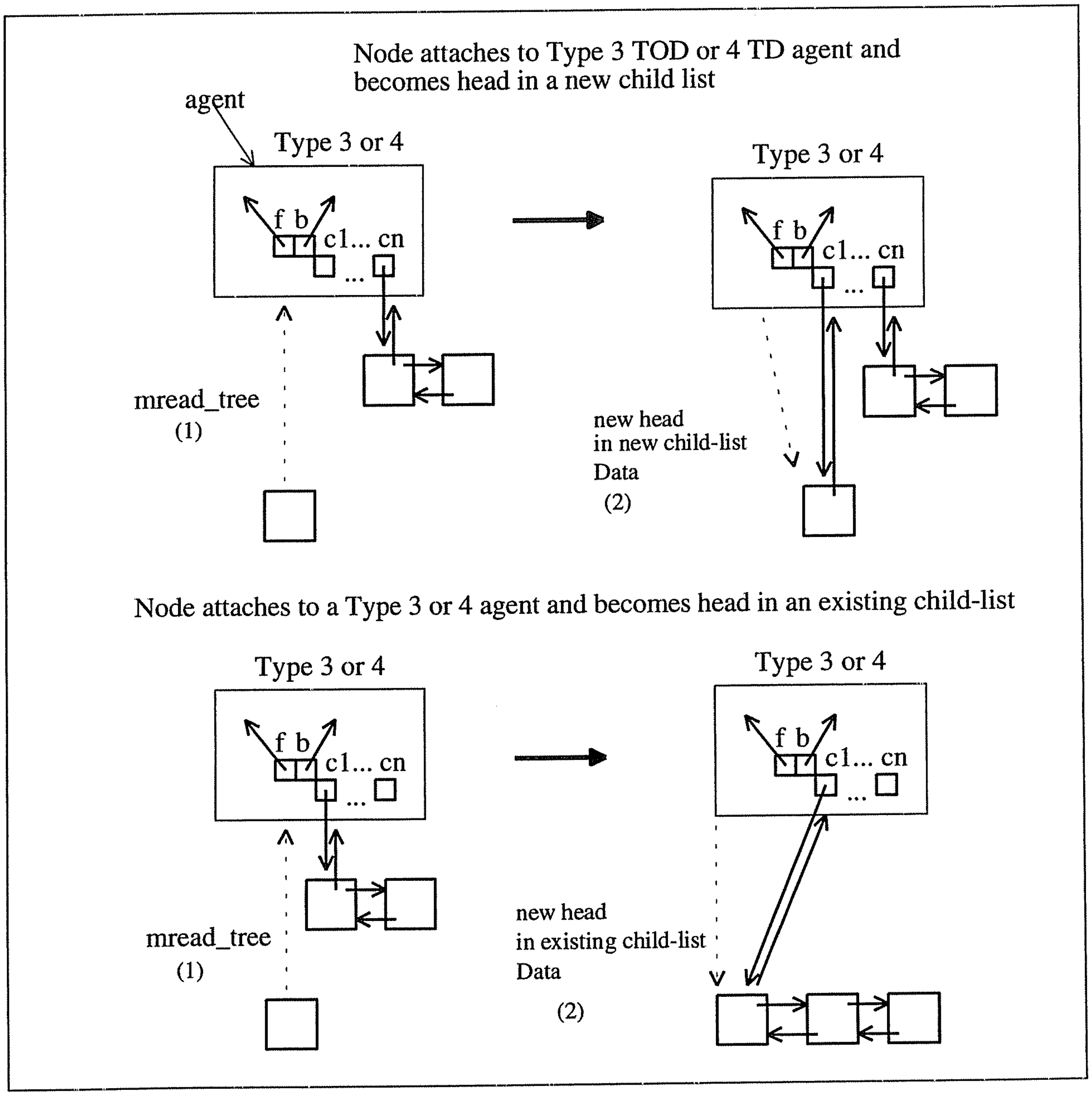

Figure 21. Hit on Type 3 TOD or Type 4 TD tag in the agent.

5.1.2. Construction with Multiple Tags in an Agent

The construction protocols are not affected by the case where the tree tags are implemented as linked or unlinked tag entries in an agent. We describe here how the multiple tags are handled in an agent. In case of a hit in the agent we have to search and find the appropriate tag entry that contains the corresponding pointer. If the tag entry does not exist (for example when the child-list has not been constructed yet) we have to allocate one. In the case of a linked list of tags, we can insert the new tag entry at any place in the list (for example in a k-ary n-cube we would keep the linked list ordered according to the order of dimensions).

A set associative cache can be used to match all tags at once so the search will be fairly quick. Notice that a set associative cache can be used for both linked and unlinked tag entries (in the case of a linked list we would not have to follow the pointers to find all entries). When a new tag must be allocated in a set associative cache we choose a set entry that contains an irrelevant tag.

5.1.3. Construction of Sparse Trees

Sparse trees are trees constructed without invoking every agent on the way to memory thus minimizing the number of superfluous nodes in the tree. Restructuring of the trees is very complex and we do not have a solution to build the tree top-down. We do have, however, a proposal to create the tree slowly, bottom-up.

For sparse tree construction it is not necessary to invoke all the agent nodes on the way to memory. Instead a requesting node will invoke only the agent at the next hierarchical level. If this node accepts a number of additional requests, it will invoke the next agent and so on. When an agent is first invoked by one of its children it will send a new *mread* request to memory. This request, in contrast to the *mread_tree* request, does not generate lookups in other agents. When an agent decides to invoke its own agent it connects to it and forces it to join the tree if it has not already. After the new agent joints the tree the requesting agent disconnects from the list it is in (with an SCI-like rollout) and rejoins in its correct position under its agent. By changing only the *forwid* and *backid* and never the child pointers of the nodes, the tree restructures and eventually it will assume the structure of a full tree.

While constructing sparse trees it is possible for an agent to have a copy of the requested line, but nevertheless ignore a passing request. For this case, we propose a 'hint' cache. The hint cache would reside in the SCI linc hardware (not in the SCI cache controller hardware) and would nudge the node to do a lookup if the address is believed to have been seen recently in a *mread* message.

5.2. Rollout

Rollouts are a concern in tree protocols because they require a replacement in order to keep the structure of the tree intact. In our trees rollouts can be more of a problem because lines may be in the tree structure just for the benefit of other nodes, without being actively accessed by the local processor. In hierarchical caching schemes a property that is sometimes useful is multilevel inclusion. This means, that the contents of a cache are the superset of all the contents, in the *mread* category, of all the caches it serves. Multilevel inclusion is desirable (but not enforced) in our case only for the cache lines that are in tree structures. In this section we propose three approaches to handle rollouts. The first is very simple but destroys parts of the tree, the second is similar to STEM and the third is to patch the tree locally.

5.2.1. Simple Rollout

In the simple cases where a node does not have children the rollout is the same as in the standard SCI. A childless node is only connected to a linear list with its *forwid* and *backid* (Figure 22). A childless node always has the data cached otherwise it would not be in the tree. Remember that childless Type 2 TND and Type 3 TOD are not allowed to exist. Whenever all the children of a Type 2 TND or a Type 3 TOD tag roll out the tag is also required to roll out[14].

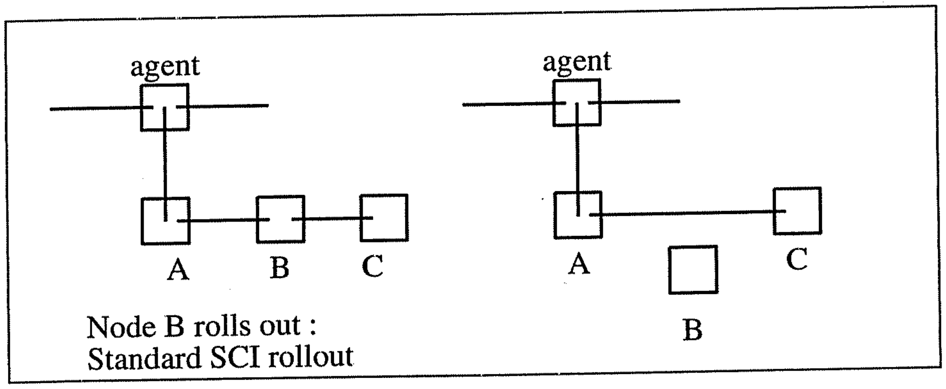

Figure 22. Simple rollout.

However, when using multiple types of tags, agents may not have the data although they take part in the tree. If all nodes that do cache the data roll out the last valid copy of the line might be lost. In order to preserve a valid copy we require that the data be written back by the rolling out nodes when they are alone in their list. This is the same behavior as in the standard SCI where the last node of the linear list writes the data back to memory when it rolls out. For our case when any node rolls out and it is the only node in a child-list, it is required to write the data back to the agent. If the agent has other children or if it is head or in the middle in its list, it ignores the write. When a node has children, it is certain that one of its descendents has the data, because childless Type 2 TND or Type 3 TOD tags are not allowed. On the other hand, if the agent is head or in the middle in its list, then another node in the list will be responsible for the data, specifically the

[14] An option here is not to allow tree tags without children. In this case a Type 4 TD tag without children would also be converted to a SCI tag.

one which will eventually be alone in the list. However if the agent is alone in its list and it does not have any other children but the one that rolls out, it accepts the write. If the tag in the agent is a Type 2 TND or a Type 3 TOD, it will write the data back to its own agent and immediately roll out[15]. Figure 23 depicts the various cases.

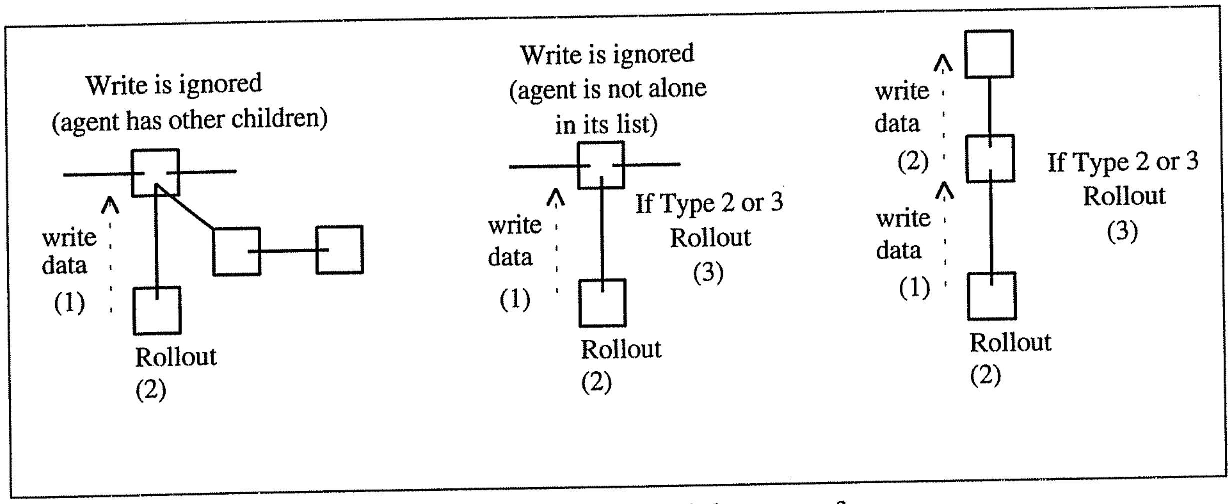

Figure 23. Rollout with multiple types of tags.

5.2.2. Destructive Rollout

The simple solution, borrowed from hierarchical caches, is to maintain multilevel inclusion [10]. When a node rolls out it invalidates all of its descendents. In this case we should try to reduce conflict misses in the caches to a minimum. Tags of Type 2 TND and Type 3 TOD help minimize conflict misses. Notice here, that by using four types of tags we maintain multilevel inclusion only for the tags and not the data.

The above simple scheme has a great appeal because of its simplicity. If we can really keep the conflict misses to a minimum then this might be one the best ways to handle rollouts. One disadvantage is that this rollout method is especially prone to thrashing behavior.

If we implement k-ary trees in an agent as a collection of multiple tags (either linked or unlinked) then the replacement algorithm is likely to delete some of them at time (but usually not all of them). In this case we do not have to invalidate all the child-lists of the agent but only those corresponding to the deleted tags. This gives us the opportunity to "test the waters" when we want to delete tags: We pick a tag to delete and invalidate its child-list in random. If the cache line is active in the children then new requests will arrive quite fast. If the cache line was inactive then we can delete some more tags of this line (until we eventually delete all tags of the line). In this way the replacement algorithm for the tags can choose the tags that correspond to inactive lines.

15 A Type 4 TD tag is not required to roll out as soon as all its children do.

5.2.3. STEM-like Rollout

A different method to handle rollouts is similar to STEM. When a node with children rolls out we search among its descendents for a childless node and swap the two nodes. The STEM-like rollout protocol in more detail is defined as follows. If the node does not have children the rollout is the same as in normal SCI lists. If the node does have children it must find a childless node, swap places with it, and then roll out. In STEM, a node has to walk down the tree, following pointers in order to find a replacement node. For our protocols we are considering two options:

1. The rolling out node sends, in parallel, a message to all its children asking for a childless node. If any of the children is indeed childless it replies immediately. Replies form the rest of the children are then ignored. If none of node's children is childless, they ask their own children for a childless line. This process continues until a childless node is found. We expect this process to find the closest childless node. Even if the node found, is not the closest it will be chosen since it is the first we know about.
2. The second method is the same as STEM. The rolling out node walks down the tree until it either finds a childless agent or reaches the lowest hierarchical level (where it will find a childless node). Since a node can have multiple children there are several options for walking the tree. Some of the options include : a) to choose randomly a child at each node b) to order the children in a specific manner according to the topology (for example in the k-ary n-cube we would choose to follow the child in the least significant dimension) c) to choose the last child that attached to the agent (this requires LRU bits for the pointers or analogous facilities).

When a childless node is found it can be reserved for the subsequent swap. This will protect against problems with concurrent deletions. When the result of the search is reported back to the originating node, it starts the swap process with the replacement node. It instructs the replacement node to roll out and patch the list it is in. It then sends in one message all of its pointers to the replacement node. Finally it notifies its siblings (or parent and sibling) to point to the replacement node and rolls out.

A node that rolls out has to search for a replacement node among its descendents. If a descendent happens to roll out at the same time, it has priority to choose a replacement, over its own set of descendents (as in STEM). If a parent node and one of its children roll out simultaneously, the search of the parent node will result in an negative answer from the child. The replacement node must be found in the rest of the subtree of its other children. If a node gets a negative answer from all its children (i.e. all of them roll out) it has to retry later. Forward process can be guaranteed with retries, since all its children will roll out and the node will eventually succeed.

While, for the STEM tree, rollouts do not affect the performance of subsequent operations (invalidation, write-updates) this is not true for our proposal. Geographical locality in the STEM tree remains as poor as it was. In our tree it is degraded from the optimal case (where pointers point to nodes in the same ring). Furthermore a replacement node cannot act as an agent for the children and would-be children of the replaced node. It can only take its place but it cannot assume its role. Subsequent read requests from would-be children, will not get intercepted in the replacement node but rather they will invoke again the replaced node in the head of the replacement node's list. As the number of rollouts is increasing, as time goes by, both locality and structure of the tree are degraded. This is expected to happen in the cases where the sharing tree exists for a long time. It is, therefore, acceptable for the invalidation of the tree or subsequent write-updates to be slower.

5.2.4. Local Patching of the Tree

Here we present rollout protocols that are not based on finding a childless node to swap places with. Instead they try to patch up the tree *locally,* manipulating only the tree structure around the agent that rolls out. The protocols described below give priority to maintaining geographical locality instead of preserving the structure of the tree.

In the first protocol the agent that rolls out instructs the heads of its child-lists to chain together in a list. It then connects the chain of the heads of the child-lists in its place. In effect it raises the heads of the child-lists to its own hierarchical level. This method preserves the geographical locality of the tree (as possible), over the quality of its structure. The tree is degraded because each node that rolls out is replaced by a linear list of the heads of its child-lists.

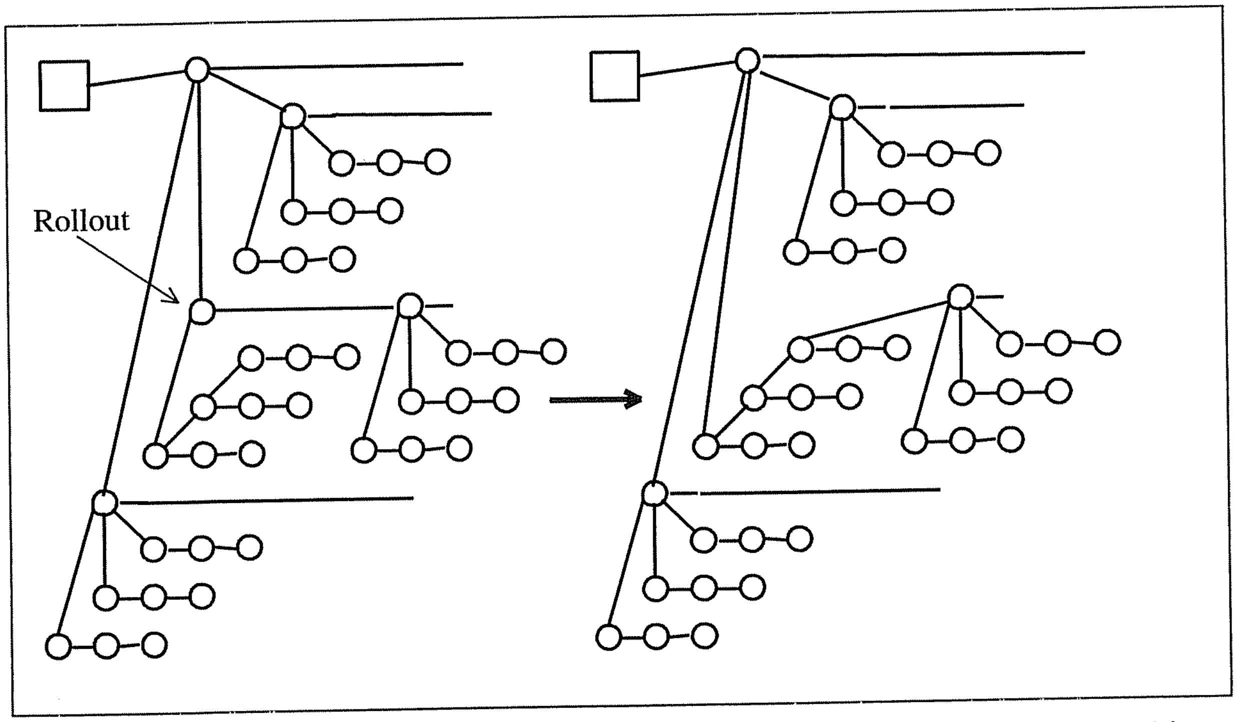

Figure 24. Rollout by chaining the heads of child-lists and raising them to the vacant position.

In figure 24 we show a tree on the left where the labeled agent rolls out. It has already chained its children in a list (compare with other nodes' child-lists). When it rolls out it leaves the chained list in its place and the resulting tree is shown in the right.

Another way to chain the children together is to concatenate all the child-lists together[16]. The agent sends messages to the heads of the child-lists that are distributed towards the tails. The child-lists are chained head to tail. The resulting linear list is then raised to the position of the agent. Figure 25 presents this method. There is the opportunity here, to use the STEM protocol on the resulting linear list and convert it to a tree. Furthermore since the agent is in control of the concatenation of the child-lists it can label the nodes in such a way so the subsequent tree merges will be performed (near) optimally.

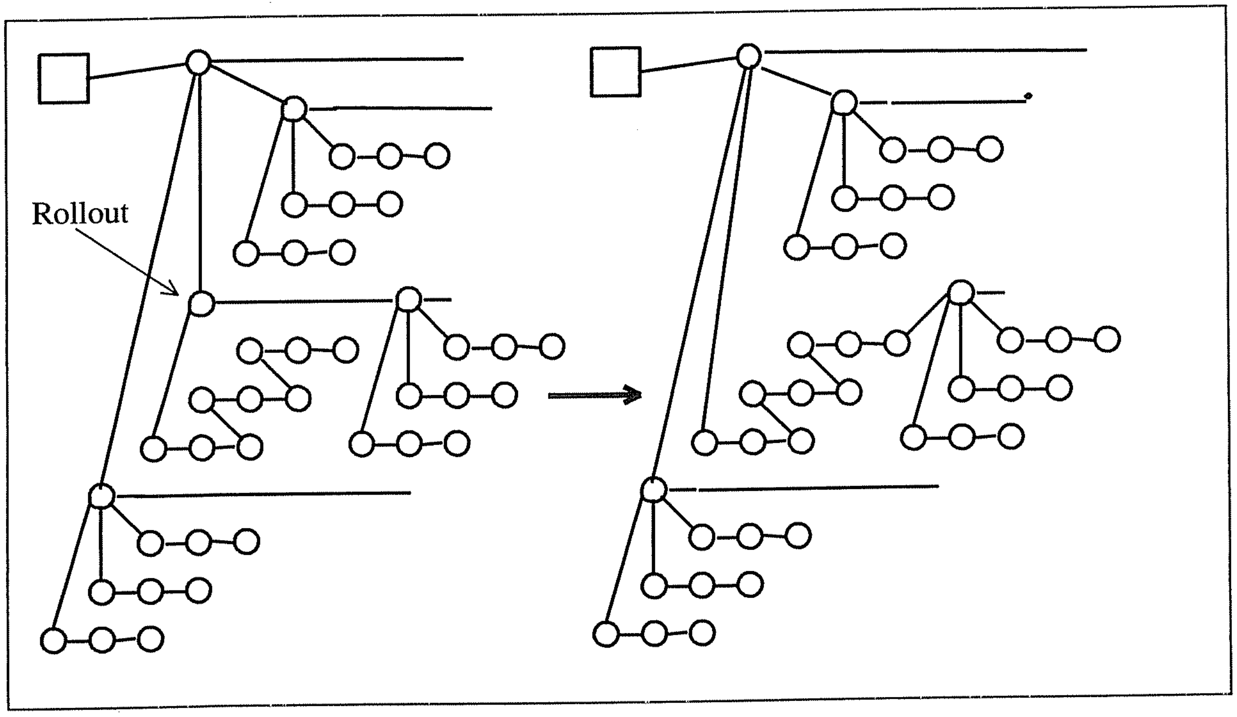

Figure 25. Chaining the child-lists together.

In the rollout protocols we have described so far (including STEM-like rollout) there is the problem of an agent rolling out and later rejoining the tree. In these situations the relation of the agent and its children is lost at the point of the rollout. If the agent rejoins the tree it will do so in another position in the linear list of the appropriate hierarchical level. New requesting nodes that would otherwise become heads in the old child-lists, will create new child-lists under the agent.

Multiple tags in a doubly linked list in an agent are especially well suited for local tree patching. The chaining happens gradually as tags are deleted from the agent. When a tag entry is deleted we use the linked list pointers to point to the head of the child list. At the head of the child-list we may need to allocate another tag entry to point to an *internal* tag in the agent. Eventually all of the linked list will be deleted from the agent. At this point the chained children will be raised to the agent's position. Figure 26 depicts the gradual rollout of an agent (figure 26).

[16] Yet another way to chain the children is presented in Appendix I in the section of Short/3Full pointers.

Since the chaining and the deletion of the tags happens gradually we can take advantage of the remaining tags in the agent. Not until all of the tags are deleted from the agent we have the problem of new requests forming new child-lists as explained above.

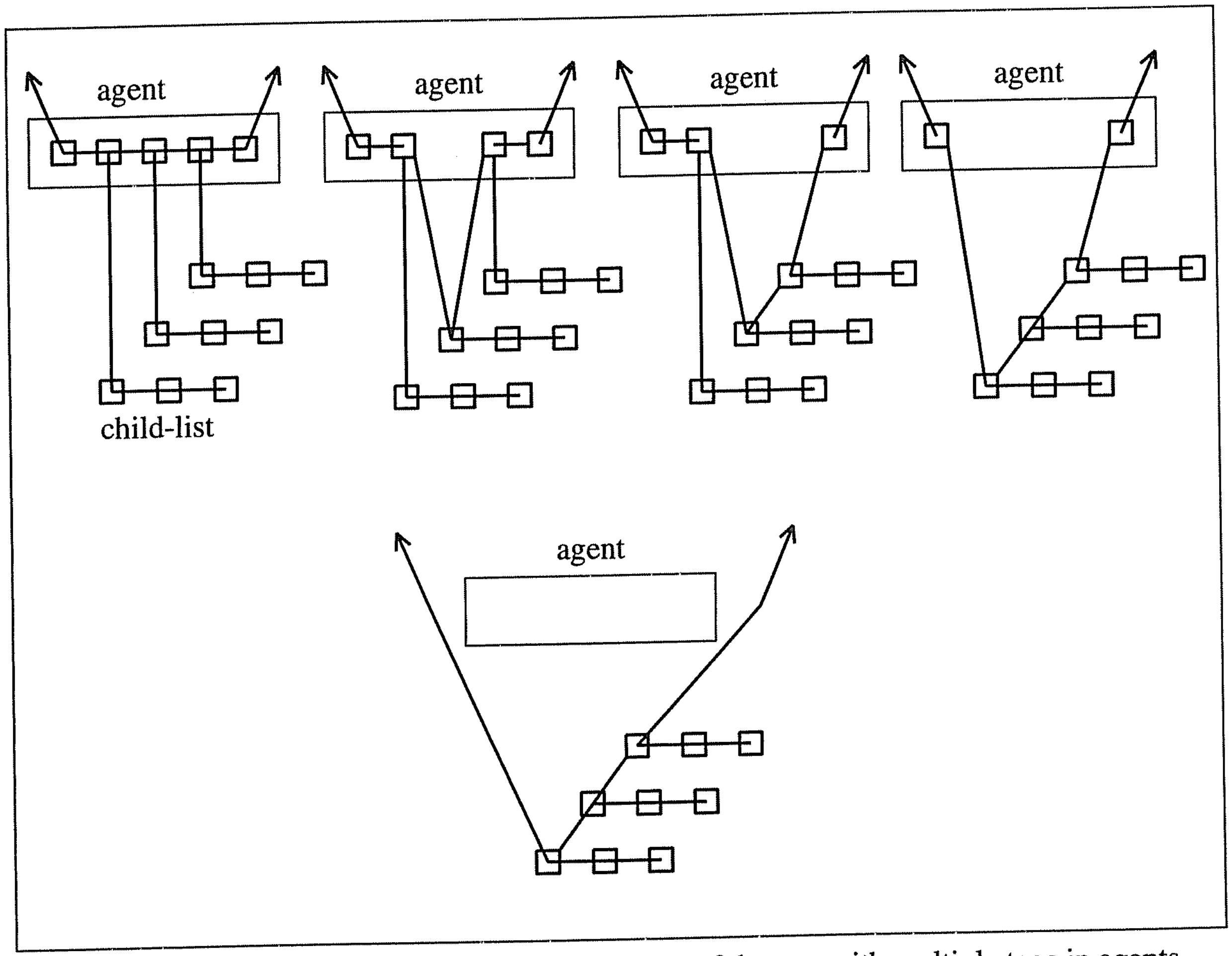

Figure 26. Chaining of children and local patching of the tree with multiple tags in agents.

5.2.5 Bitmaps and Rollout

The bitmap scheme mentioned earlier could be used to store in an agent all the siblings for each of its child lists. The pruning cache concept is based on the observation that if the bitmap information is lost, the tree can be reconstructed conservatively, that is, by including all possible siblings in the child list. While nodes not in the tree may therefore be inadvertently inserted into the tree, they need not contain data, so no harm is created by adding them (except that the tree is unnecessarily larger than it need be, and extra traffic may be generated in destroying it). This approach suggests a fifth type of tag: Absent. This virtual tag is not stored at all, but is implicit in situations where a node is informed by its parent that it is in a list.

One of the benefits of the bitmap scheme is that a node can be rolled out for free. If it is simply deleted, it automatically becomes a Type 5 (virtual) tag, and nothing further need be done. Note that this feature meets a basic requirement of the bitmapped scheme: all lists must be contained within a single ring. This scheme therefore always forms the perfect tree, and avoids the problems of patching ill-formed trees.

5.2.6. Recursive Rollout

In this rollout protocol the node that rolls out asks one of its children to take its place. The empty place left by the child will be filled the child's child. When a node asks its child to take its position the child will ask its own child to do the same. This continues recursively until we reach the lowest hierarchical level where a child will not have any children. This protocol still preserves some of the locality but it affects a lot of nodes. It is also more complex.

5.3 Destruction (Invalidation)

Destruction (invalidation) of the tree is defined similarly to STEM. When a node wants to write a line, it first becomes the head of the top level list. Invalidation messages are then forwarded down the tree in parallel. At each stage a node can accept the invalidation message (with a response to the node that sent it) assuming the responsibility to forward it. Alternatively, if the node is childless, it can return its *forwid* pointer to the sender. In the latter case the sender repeats the invalidation message to the node pointed by the returned *forwid* (as in the base SCI protocol). When the invalidation messages reach the tails of the lists, acknowledgment messages are sent the reverse way. The tails invalidate themselves and send their acknowledgment back. Nodes wait[17] for the acknowledgment of all the invalidation messages they forwarded, invalidate themselves, and return their own acknowledgment.

Invalidation is fast because all the messages are exchanged locally, between nodes within the same ring. Even if the longest path of our tree is larger than that of the STEM tree, invalidation in many cases is faster because of the low message latency. In the k-ary n-cube topologies the longest path of our tree is $(k-1)\times n$ while the longest path of the balanced STEM trees is $(2\times \log_2(k^{(n)}-1) = 2\times n\times \log_2(k)-2$. In our trees the messages travel within the rings so their average distance is $\frac{k}{2}$. For STEM, assuming a randomly mapped tree, the messages travel $\frac{n\times(k-1)}{2}$ links, which is the average distance in the k-ary n-cube. The invalidation of the tree in both cases has twice as many steps as the longest path of the tree because it happens in two phases. At each step we have two latencies: 1) a lookup in a cache and 2) the latency to forward the invalidation to the next nodes (or to return the acknowledgments). Assuming that the latency for one message to traverse one physical link is the unit latency and a lookup is ten times more expensive we get the following functions for the latency of invalidation (*KXS* is the function for our schemes while *STEM* is the function for STEM):

17 Waiting for a number of acknowledgements can be implemented with a counter, associated with each line. The counter is initialized to the number of invalidation messages that were sent and is decremented with each acknowledgement. When the counter reaches zero the node received all its acknowledgments. Alternatively when an acknowledgment is received, we set the corresponding pointer to an invalid value (i.e. the id of the receiver). The node tests if all the pointers are invalid (with the exception of *backid*). When this condition is true, the node has received the proper number of acknowledgments.

$$\mathrm{KXS}(k,n) = 2 \times \{(\frac{k}{2} + 10) \times [n \times (k-1)]\}$$
$$\mathrm{STEM}(k,n) = 2 \times \{(\frac{n \times (k-1)}{2} \times 10) \times [2 \times n \times \log_2(k) - 2]\}$$

In figures 27 through 31 we plot the latency functions for various values of *n* and *k*. Each figure shows the results for different n (dimensions). The x axis is system size. The y axis is the time in unit latencies for the invalidation of the tree.

As for the Omega/Flip (N stages, R rings per node) networks of we note that for most cases the longest path of the tree is less than that of the best balanced STEM tree.

We note here that confining lists to a single ring potentially offers very significant reductions both in message traffic and destruction latency for a scheme that can exploit broadcast effectively. The bitmap scheme, in particular, can invalidate an entire tree very efficiently by employing recursive broadcast, once per ring.

5.4 StoreUpdate

5.4.1. StoreUpdate by the Head of Tree

StoreUpdate in the tree, is defined similarly to STEM. A node wishing to update all other entries in the sharing tree, leaves the tree (performing a rollout as described in the previous section). It then prepends to the top level list. Being the head, directly connected to memory, it can initiate a StoreUpdate. The StoreUpdate happens in two phases. The first is the initiation phase where the new data propagate throughout the tree in the same way as the invalidation messages. A node receiving a StoreUpdate from its parent or left sibling forwards it in parallel to all its children and right sibling. Alternatively a childless node can simply return its *forwid* to its parent, giving it the responsibility to forward the StoreUpdate to the next node. When the data reach the tails of the lowest level lists, the tails return confirmation of their update. Nodes in the tree wait for confirmation from all their children and right sibling, before they return their own confirmation messages to their parent.

5.4.2 StoreUpdate with Bimodal Behavior

A method of doing StoreUpdate that is currently discussed in Kiloprocessor Extensions to SCI working group is for a node to have bimodal (or schizophrenic) behavior. This behavior allows a node to be in its proper position in the tree and also temporarily be head in the highest level list. A node in this position has exclusive access to the tree and can perform a StoreUpdate as described above.

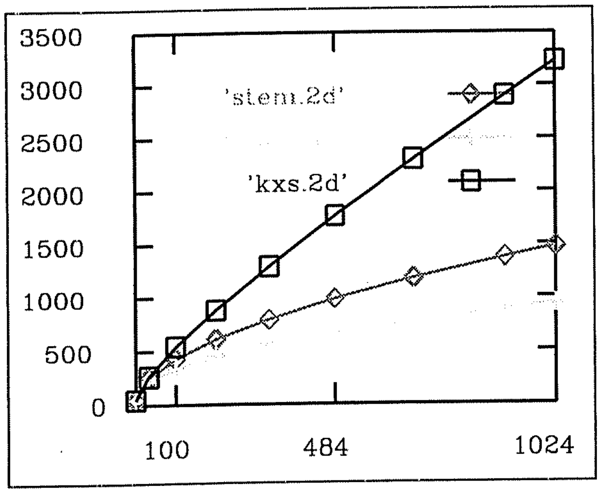

Figure 27. k-ary 2-cube.

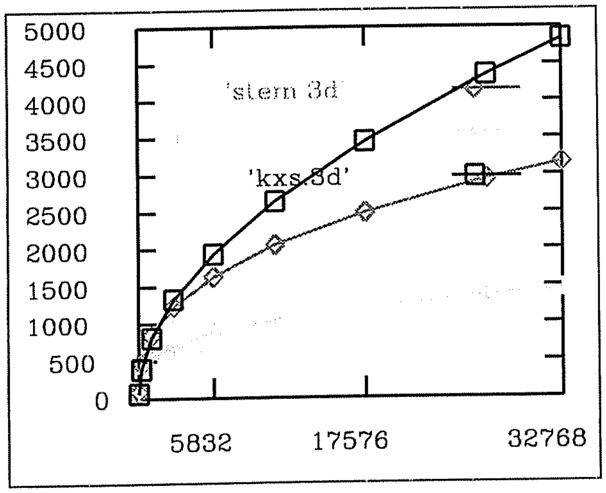

Figure 28. k-ary 3-cube.

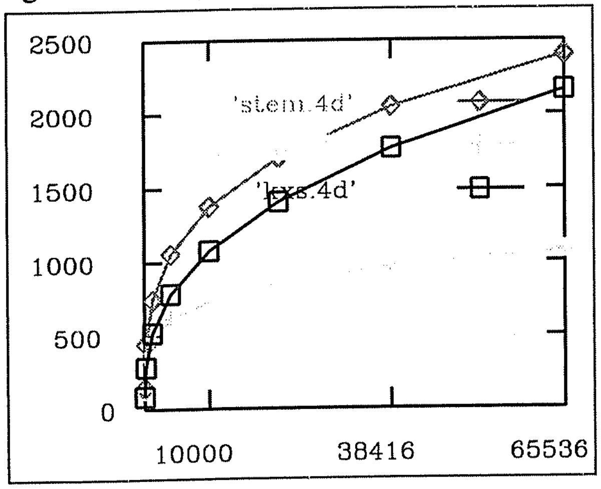

Figure 29. k-ary 4-cube.

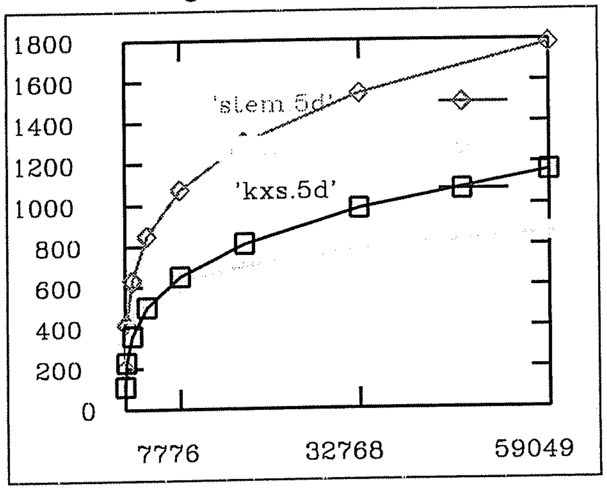

Figure 30. k-ary 5-cube.

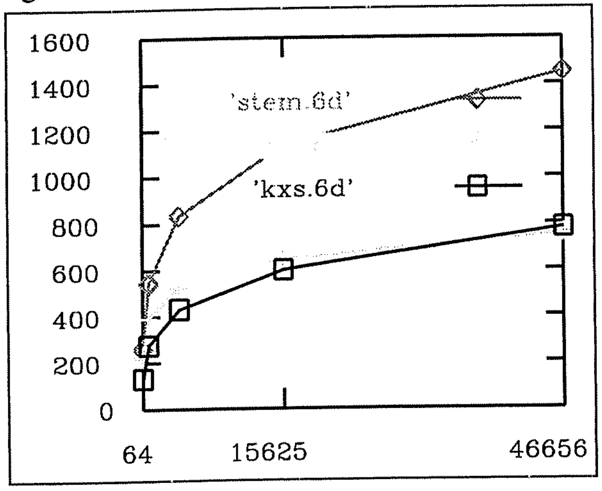

Figure 31. k-ary 6-cube.

APPENDIX I

1. Short Pointer Conversions

In this appendix we present two methods of switching from many short pointers to three or four full pointers. In a system that uses fixed size tags and the minimization of storage requirements is important we may use small tags that contain all necessary pointers to implement k-ary trees as long as they are short. However, in order to use short pointers, that cannot cross rings, it is imperative that each node is in its proper position[18] in the tree. A node must only point to its true parent or to its true children. This severely restricts our options for sparse tree construction, rollout and StoreUpdate protocols, as they are discussed in previous sections. It seems that having only short pointers we are unable to do anything more than create the full tree and invalidate it.

Thus we propose an implementation, in which we use short pointers for performance as long as a line is in its proper position, but we convert to a smaller number of full pointers if need be. Short pointers can be stored in the same bits as long pointers, allowing either a few long pointers or more short pointers. Since we have the option to use full pointers we can define protocols where a node can find itself at any place in the tree. Here, we will show how we can convert from short to full pointers and how this affects the structure of a k-ary tree. The basic idea, behind converting from short to full pointers, is to chain the child lists together and handle them with a single pointer. To chain the child-lists together, the parent sends messages to the heads of the child-lists and instructs them to connect to each other. In turn, the heads may chain their children together, forcing them to convert to full pointers too.

We propose two variations of this implementation. One requires more storage and has higher performance and the other less storage with lower performance.

1. Four full pointers or $2+R$ short pointers. The four full pointers allow us to chain the child lists together with out significantly degrading the structure of the tree. The four full pointers are: *forwid* and *backid*, which handle the high level list; a *childid* pointer, which points to the chained child lists of the node; and a *chainid* pointer that connects the node's sibling lists in a chain when requested by their parent.
2. Three full pointers or $2+R$ short pointers. This is similar to the above but we do not specify a *chainid* pointer. Again we chain the child lists together and we point to them with the *childid* pointer. In the cases where we are short of full pointers, we connect the rest of list in the same hierarchical level, at the end of the chained child lists. This scheme is more complex and it has lower performance than the previous scheme.

1.1. Short/4 Full Pointers

For the first case the tags will be as shown in figure 32. In figure 33 we give a simple example of a node converting to full pointers. The node in the top left corner instructs its children to chain together. It sends a message to the third child instructing it to connect to the second child. This

18 Proper position is when a node is a member of the correct list. Position in the list doe not matter.

child will convert itself to full pointers. It will then replace its *backid* pointer (that previously pointed to the parent) with a pointer to the second child. Concurrently, the top node sends a message to the second child. This child will also convert to full pointers. It will then use its *backid* to point to the first child, the *chainid* to point back to the third child and its *forwid* pointer to hold its own list. Similarly, the first child will connect to the second with its *chainid* and use its *backid* to point to the parent. Figure 33, shows the resulting pointers when both the parent and the heads of its child-lists chain with other sibling lists.

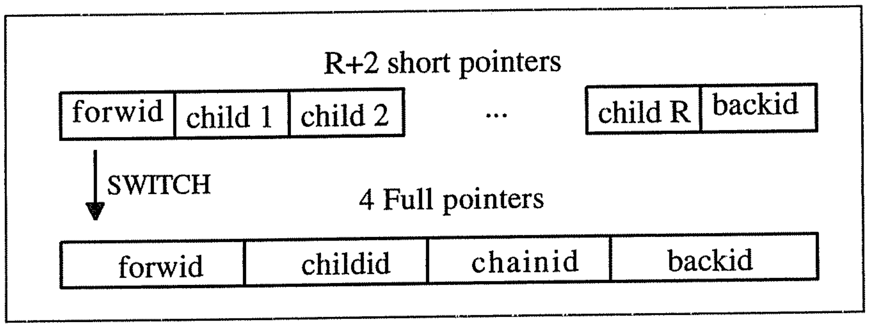

Figure 32. Storage for Short/4-full pointers.

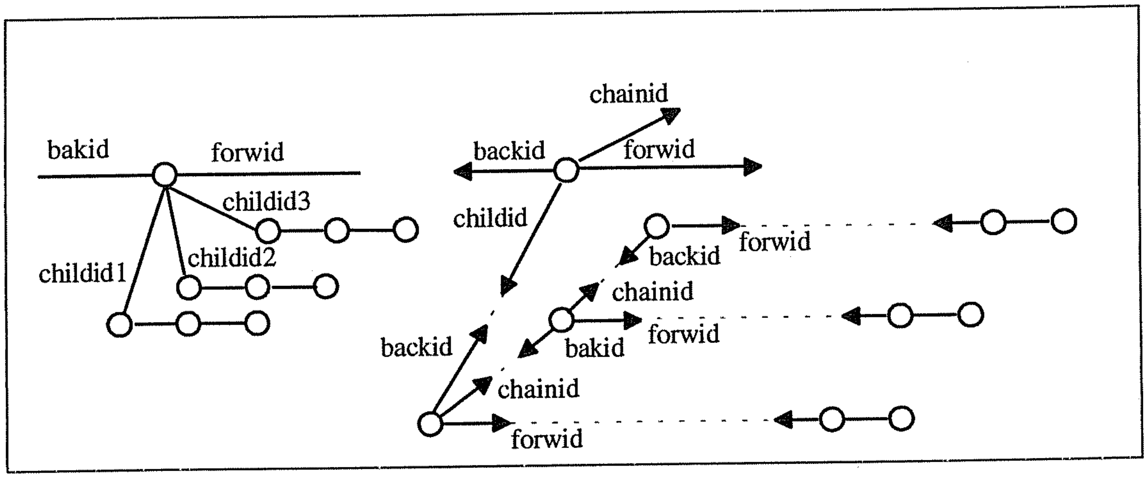

Figure 33. Converting from short to four full pointers.

The structure of the tree of figure 1 when the two top nodes convert to full pointers is shown in figure 34. The conversion of the two top most nodes to full pointers results in a cascade of conversions that reaches the lowest level of the tree. However the conversions to full pointers do not extend beyond non-head nodes.

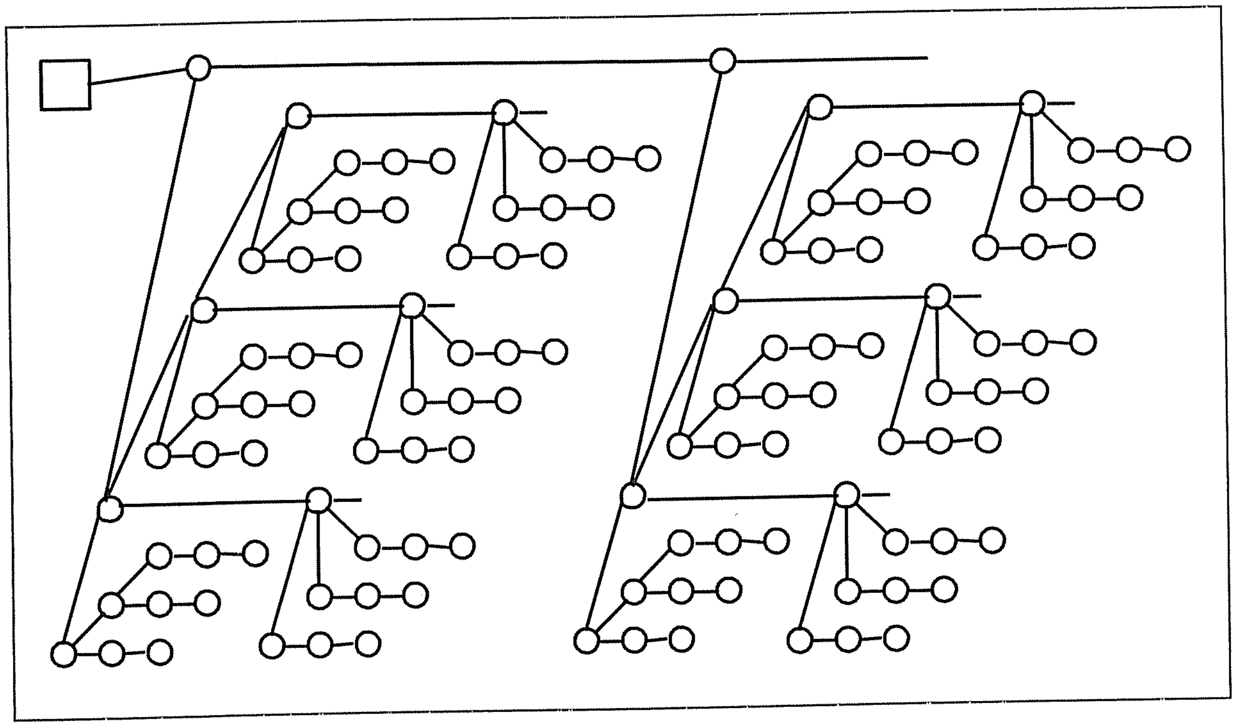
Figure 34. Converting from short to four full pointers.

It is obvious that converting from short to full pointers we change the structure of the tree. If all nodes convert to full pointers then we no more have a k-ary tree but rather a 3-ary tree. The conversion can increase the longest path in a tree, at most, by $(R-1)\times(L-1)$ where R is the number of child lists and L is the number of hierarchical levels of the tree.

Whether this increase to the longest path of the tree is acceptable or not, depends in the structure of the tree that differs according to the topology. The trees in k-ary n-cube have $L = n$ hierarchical levels, and at most $(n-1) = R$ child lists per node. The longest path is $(k-1)\times n$. The trees in the k-ary n-cube do not have the same number of child lists in every level. In the highest level, the number of children is $(n-1)$ and decreases by one until the lowest level. In this case we add to the longest path $\frac{(n-1)\times n}{2}$. The longest path after a switch from short to full pointers is $[n\times(k-1)]+\frac{(n-1)\times n}{2}=n\times(k-1+\frac{n-1}{2})$.

The trees in an Omega/Flip topology of N stages have a longest path of $\frac{N\times(N+1)}{2}$. Switching from short to full pointers results in a longest path of $\frac{N\times(N+1)}{2}+(N-1)^2$.

1.2. Short/3-full Pointers

When we only specify three full pointers there will be cases where we cannot hold on to all the pointers that are needed. In figure 35 we show the case. A node needs two pointers (*backid* and *forwid*) to hold onto the high level list, one pointer to hold its child list (*childid*) and one pointer to chain with its siblings (*chainid*). In the case where we only have three full pointers we use the *backid* and *forwid* for the chain list and the *childid* to hold everything else. This means that we not only chain together the children but we append at the end of the chain list, the rest of the high level list.

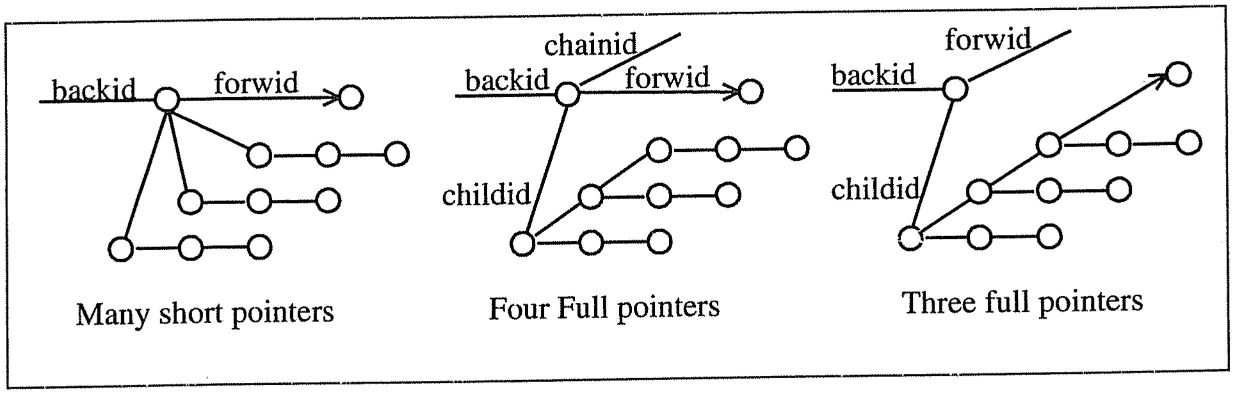

Figure 35. Converting from short to four full and three full pointers.

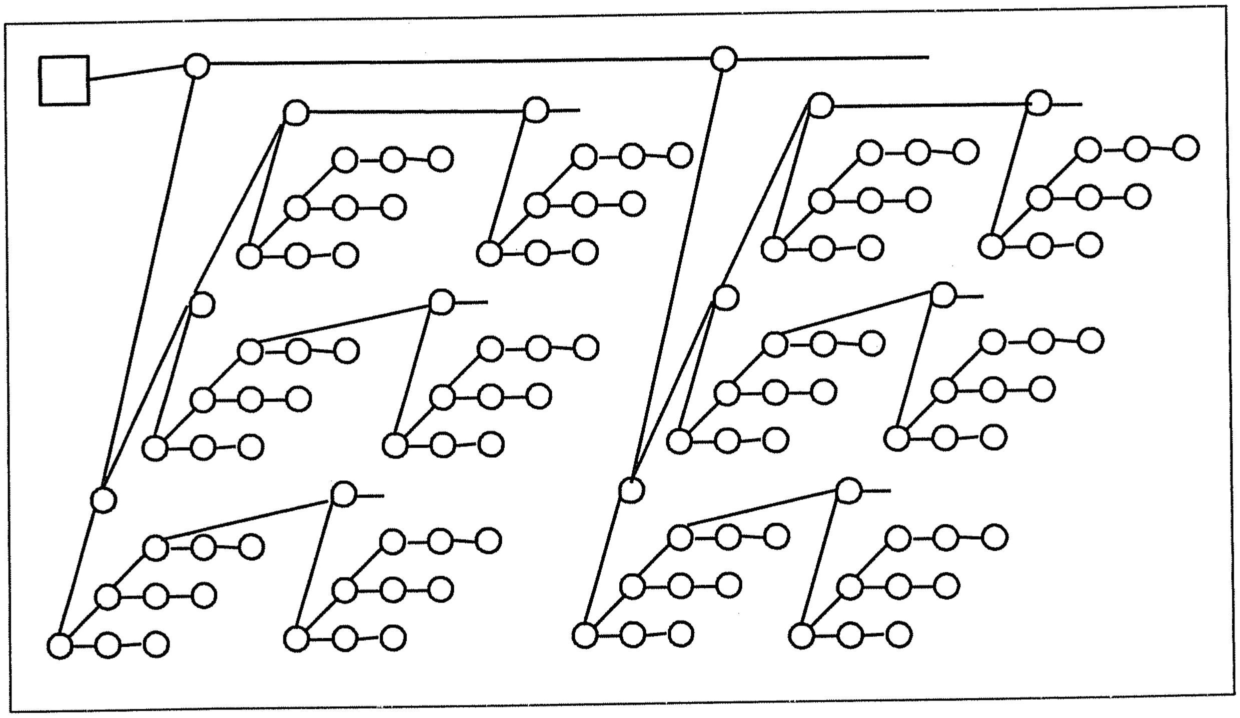

Figure 36. Structure of the tree with three full pointers.

When we use short pointers the structure of the tree will be the same as in figure 1. When we use full pointers the tree will assume the structure of figure 36. There is a clear trade off of storage versus performance here. Using four pointers results in better structured trees but it requires 33% more storage.

1.3. Rollout with Short and Short/Full Pointers

When we use short pointers the only option we have for rollouts is to invalidate all descendents (along with their linear lists). We simply cannot replace nodes in the tree. The combined simplicity of short pointers and destructive rollouts is very attractive. If it turns out that we can minimize conflict misses so that rollouts will not present a serious performance problem, this is a good, yet very simple solution for an SCI tree protocol. However, these methods will always be subject to thrashing behavior and one can always write a program that will have very high conflict miss rates. When we use the combination of Short/Full pointers we can use any protocol described previously as long as we switch to full pointers.

For the STEM-like rollout protocol when a node wishes to roll out it first converts to full pointers. This will force its descendents, that are the heads of the child-lists, to convert to full pointers all the way down to the lowest level. This is convenient, since the search for replacement considers only the nodes that have converted to full pointers. The replacement node, having full pointers, will be ready to swap. We can even combine the search for replacement and the conversion to full pointers as they expand in the same way down the tree.

For the local patching of the tree the chaining of the children is the same as described here. The conversion from short to full results in the same chaining we would have for the rollout.

1.4. Invalidation

The protocols for invalidation are not affected by the use of short pointers or the conversions. However the performance of the invalidation changes as the tree converts to full pointers. We believe that the performance is still in acceptable levels even if all the tree converts to full pointers.

APPENDIX II

Guidelines for the use of Multiple Types of Tags

In this appendix we give some guidelines for the use of multiple types of tags. We present actions taken on a data conflict (the tag is a Data Victim) and actions taken on a tag conflict (Tag Victim)

Data Victim

Processor request for data (read or write) results in choosing a victim line that has an associated tag. Table 4 describes the actions taken for the tag:

	Data Victim	Action
Type 1 SCI		Replace
Type 2 TND	Never	None
Type 3 TOD		Convert to Type 2; Allocate new entry
Type 4 TD		Convert to Type 2; Allocate new entry

Table 4.

Tag victim

The tag is chosen for replacement due to one of the following reasons: 1) A new tag entry has to be allocated because of a Data Victim conversion (see Table 4) 2) A new child request results in allocation for a Type 2 TND (or Type 3 TOD) tag. For the two cases we describe strategies for replacement of the tags in tables 5 and 6. In table 5 a new entry has to be allocated because of a Data Victim. Incoming Type denotes the new tag type. Columns list the priority (1 highest, 4 lowest) and action suggested.

Choose :	Incoming Type 1 SCI	Incoming Type 2 TND	Incoming Type 3 TOD	Incoming Type 4 TD
Type 1 SCI	1; Invalidate data item; SCI Rollout	Never	Never	1; Invalidate data item; SCI Rollout
Type 2 TND	2; Tree Rollout	Never	Never	2; Tree Rollout
Type 3 TOD	3; Invalidate data item; Tree Rollout	Never	Never	3; Tree Rollout
Type 4 TD	4; Invalidate data item; Tree Rollout	Never	Never	4; Invalidate data item; Tree rollout

Table 5.

In table 6 a new entry has to be allocated due to new child request. The new entry is a Type 2 TND tag (which is converted to a Type 3 TND in three cases).

Choose :	Incoming Type 2 TND
Type 1 SCI	3; Invalidate data item; SCI Rollout; Convert to type 3; Load data into cache
Type 2 TND	1; Tree Rollout
Type 3 TOD	2; Invalidate data item; Tree Rollout; Convert to type 3; Load data into cache
Type 4 TD	4; Invalidate data item; Tree Rollout; Convert to type 3; Load data into cache

Table 6.

Acknowledgements

We wish to thank David V. James, Ross E. Johnson and Stein Gjessing for their suggestions, contributions and constructive comments on this work.

References

[1] IEEE Standard for Scalable Coherent Interface (SCI) 1596-1992, IEEE 1993.

[2] Ross E. Johnson. "Extending the Scalable Coherent Interface for large-Scale Shard-Memory Multiprocessors," PhD Thesis, University of Wisconsin-Madison, 1993.

[3] Gregory F. Pfister and V. Alan Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks", Proceedings of the 1985 International Conference on Parallel Processing, pp. 790-797, August 20-23 1985.

[4] Marvin H. Solomon and Raphael A. Finkel, "Processor interconnection strategies," IEEE Transactions on Computers, Vol. C-29, pp. 360-371, May 1980.

[5] Raphael A. Finkel and Marvin H. Solomon, "The lens interconnection strategy," IEEE Transactions on Computers, Vol. C-30, no. 12, December 1981.

[6] William J. Dally and Charles L. Seitz, "Deadlock-Free Routing in Multiprocessor Interconnection Networks," IEEE Transactions on Computers, Vol. C-36, no. 5, pp. 547-553, May 1987.

[7] Duncan H. Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Transactions on Computers, Vol. 24, no. 12, pp. 1145-1155, December 1975.

[8] R. Simoni and M. Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. Proceedings of the International Symposium on Shared Memory Multiprocessing, 1991.

[9] Steve L. Scott and James R. Goodman, "Performance of Pruning-Cache Directories for Large-Scale Multiprocessors," IEEE Transaction on Parallel and Distributed Systems, Vol. 4, no. 5, May 1993.

[10] J. L. Baer and W. H. Wang. "Architectural choices for multi level cache hierarchies," Proc. 16th International Conference on Parallel Processing, pp. 258-261, 1987.

[11] John Carter, John Bennet and Willy Zwaenopoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," Proceedings of the Conference on the Principles and Practices of Parallel Programming, 1990.

[12] Wolf-Dietrich Weber and Anoop Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors," ASPLOS III, pp. 243-256, 1989.