Request Combining in Multiprocessors with Arbitrary Interconnection Networks

Alvin R. Lebeck and Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton street
Madison, WI 53706
<alvy,sohi>@cs.wisc.edu

To appear in IEEE Transactions on Parallel and Distributed Systems

Abstract

Several techniques have been proposed to allow parallel access to a shared memory location by combining requests; they have one or more of the following attributes: requirement for *a priori* knowledge of the requests to combine, restrictions on the routing of messages in the network, or the use of sophisticated interconnection network nodes. We present a new method of combining requests that does not have the above requirements. We obtain this new method for request combining by developing a classification scheme for the existing methods of request combining. This classification scheme is facilitated by separating the request combining process into a two part operation: (i) determining the *combining set* which is the set of requests that participate in a combined access, and (ii) distributing the results of the combined access to the members of the combining set. The classification of combining strategies is based upon which system component, processor elements or interconnection network, performs each of these tasks. Our approach, which uses the interconnection network to establish the combining set, and the processor elements to distribute the results, lies in an unexplored area of the design space. We also present simulation results to assess the benefits of the proposed approach.

1. Introduction

Arvind and Iannucci state that the design of a large-scale, shared memory multiprocessor must address two basic issues [2]:

(1) *it must tolerate long latencies for memory requests,*

(2) *it must achieve unconstrained, yet synchronized, access to shared data.*

While several techniques, for example caches and prefetching [24], and low level context switching [25], have been proposed to tolerate the latency of memory requests, heretofore the only known methods of allowing unconstrained, yet synchronized, access to shared data are implementations of *request combining*. The earliest published proposal for request combining was in the CHoPP system [28], where several read requests to a common memory location are combined in the interconnection network and are satisfied with only a single access of the memory location.

When two read requests destined for the same memory location meet at a node of the network, the source of one of the requests is saved and only one read request is forwarded to memory. When the response of the read request arrives at the node where combining took place, two responses are sent back toward the processors.

The idea of combining read requests, or *read combining* in CHoPP was extended in the NYU Ultracomputer to allow several types of requests to combine [6]. The Ultracomputer uses the *Fetch&*Φ primitive, where Φ is any associative and commutative operator. An enhanced interconnection network with the topology of an Omega network is proposed to perform combining on the *Fetch&*Φ primitive.

The Ultracomputer style of request combining is illustrated in Figure 1. When a *Fetch&*$\Phi(X, e)$ request meets a *Fetch&*$\Phi(X, j)$ request at a network node, *combining* takes place: e is saved in a *wait buffer*, an ALU in the node computes $e\Phi j$, and the request *Fetch&*$\Phi(X, e\Phi j)$ is forwarded to memory. When a response V is received from memory for the *Fetch&*$\Phi(X, e\Phi j)$ request, *decombining* takes place: V is forwarded as a response to the *Fetch&*$\Phi(X, e)$ request, $e\Phi V$ is forwarded as a response to the *Fetch&*$\Phi(X, j)$ request, and e is removed from the wait buffer. Correct operation is guaranteed if the combining of requests satisfies the serialization principle: the final state of the system must be consistent with the servicing of all requests in some (unspecified) serial order [6].

There are three distinct features of the Ultracomputer style of request combining:

(1) Requests are combined on the forward trip through the network.

(2) State is saved in the network when requests are combined.

(3) Requests are decombined on the return trip through the network.

Combining in a network node first requires a comparison to determine if two requests are combinable: the two requests must be *Fetch&*Φ requests to the same memory location. This requires the use of comparators in the network nodes. When two requests are combined, each request is removed from the queue, the value of e is saved in the wait buffer, the operation $e\Phi j$ is carried out in the ALU, and the request *Fetch&*$\Phi(X, e\Phi j)$ is placed in the queue to forward to memory. The wait buffer must be large enough to hold as many values as there are requests that can be combined at this node, else combining will not take place. On the return trip, each returning request searches the wait buffer and, if decombining must take place, appropriate actions are initiated. This implies that the return path must be identical to the forward path for decombining to take place, or the return path must have at least one node in common with the forward path—the node where the combining state is stored. Almasi and Gottlieb [1]

give several examples of how such *hardware combining* can eliminate serial bottlenecks.

Several alternative proposals for request combining have appeared in the literature [8, 11, 19, 21, 29, 32]. The primary focus of these efforts is on reducing the cost of the combining network. This is accomplished either by altering the topology of the combining network or by requiring the system software to reduce the amount of contention for shared data.

This paper has two purposes. The first is to develop a taxonomy that can be used to categorize combining methods proposed to date. This allows us to enumerate the issues involved, and to make a comparison of known techniques for combining requests. The second is to propose a new approach to request combining, one which can be used with arbitrary interconnection topologies.

We develop a taxonomy in section 2, and classify existing methods using this taxonomy. We observe that one area of the design space, which we call Interconnect-Processor combining, has not been explored for arbitrary interconnection networks. We explore this in section 3, where we present a new scheme for request combining. The potential of the new combining scheme is evaluated in section 4. Section 5 summarizes the paper and suggests directions for further research.

2. A Taxonomy for Request Combining

2.1. Parallel Prefix Computation and Request Combining

Kruskal, Rudolph, and Snir [14] observed that request combining is very similar to the problem of *parallel prefix computation* [15]. Given the elements $x_1, x_2, ..., x_n$ a prefix computation produces the results:

$$\begin{Bmatrix} r_1 = & x_1 \\ r_n = & r_{n-1} \Phi x_n \end{Bmatrix}$$

where Φ is any associative operator. Computing the results in parallel is termed a parallel prefix computation [15].

To examine the similarity between request combining and parallel prefix consider an example in which four processors add a constant, C, to a shared variable X and receive the previous value of X. Assume the processors simultaneously execute the atomic operation *Fetch&Add*(X,C). The values returned to the processors are X, $X+C$, $X+2C$, and $X+3C$. Regardless of the order that requests are serviced, memory has the final value of $X+4C$. This is simply the set of results $\{r_1, r_2, r_3, r_4, r_5\}$ produced by a prefix computation on the set of elements $\{X,C,C,C,C\}$ with

the addition operator (+).

Based on this observation, we see that arbitrary request combining is a two part operation. The first part, or task of the combining method, is to determine the set of requests that are destined for the same memory location and need to be combined. We call this set of requests the *combining set*. The second task is to distribute the results of the combined access to the appropriate processors by performing a prefix computation on the combining set.

A prefix computation network, such as the one proposed for *scan* primitives by Blelloch [3], can be used to distribute the results of the combined access. In such a network, state is saved on the forward trip through the prefix computation network, and the results are distributed on the return trip, very similar to the Ultracomputer approach towards combining. However, the use of a prefix computation network requires *a priori* knowledge of the combining set. Blelloch proposed the *scan* primitives for a Single Instruction Multiple Data (SIMD) paradigm where the elements on which to perform the prefix computation, in our case the combining set, are stored in an array. The array is distributed across the processors and participation in the prefix computation is based on the processors' *activation* status. Therefore, to use a prefix computation network to distribute the results of a combined access, the combining set must be established prior to insertion in the prefix computation network. Alternatively, in the Ultracomputer approach towards combining, the combining set is determined dynamically in the interconnect by comparing the addresses of requests on the forward trip through the network. The results are then distributed to the appropriate processors on the return trip through the network.

The two techniques use different system components to establish the combining set: the Ultracomputer uses the interconnect, whereas Blelloch uses *a priori* knowledge in the processor elements. We can therefore obtain a taxonomy for request combining by specifying which system component, processor elements or interconnect, performs each of the tasks involved in combining requests.

2.2. Classification of Existing Request Combining Strategies

Based upon how the combining set is determined, and how the results of the prefix computation are computed and distributed, the design space for request combining can be divided into four regions (Figure 2). Interconnect-Interconnect Combining (IIC) covers schemes in which the interconnection network determines the combining set and distributes the results. In Processor-Interconnect Combining (PIC) the processors establish the combining set, and the interconnection network distributes the results. Schemes in which the processors perform both tasks are

classified as Processor-Processor Combining (PPC). Finally, Interconnect-Processor Combining (IPC) indicates that the interconnection network determines the combining set, while the processors distribute the results. In the following subsections we discuss existing methods of request combining according to which region of the design space they belong to.

2.2.1. Interconnect-Interconnect Combining (IIC)

The CHoPP [28] and the NYU Ultracomputer [6] methods of request combining are instances of IIC: the interconnection network determines the combining set and distributes the results. The IBM RP3 [21] researchers proposed the basic ideas of the Ultracomputer method of combining for their implementation. However, the RP3 has two interconnection networks, one network which combines requests and one that services non-combining requests. A distinction is made between non-combinable and potentially combinable requests (typically synchronization requests), and the interconnect dynamically determines the combining set of the potentially combinable requests.

Two alternative techniques for IIC are presented by Tzeng [29] and Hsu and Yew [11]. Tzeng separates the interconnect into a routing section and a combining section. It is assumed that requests which may combine are distinguished from non-combining requests by examination of the opcode. Such requests are directed to the combining section of the network. The combining section of the network then determines the combining set. Hsu and Yew propose a single stage shuffle-exchange combining network in addition to a non-combining network. These two proposals are similar to the IBM RP3 method of combining, but the difference in topology of the combining network reduces the hardware cost of the networks. However, the basic technique of request combining in all three schemes, the IBM RP3, Hsu and Yew, and Tzeng, is the same as the technique used in the Ultracomputer.

An interesting example of IIC read combining can be found in schemes with hierarchical cache/memory structures, such as Cache-Only Memory Architecture (COMA) machines [7, 27], or Non-Uniform Memory Access (NUMA) machines with hierarchical caches [18, 23, 30]. Here read combining can be implemented by using a technique similar to the CHoPP method of read combining. A read miss of a cache block at one level of the hierarchy causes a request to be propagated to the next higher level in the hierarchy. Subsequent read misses of the same cache block at the same level in the hierarchy cause state to be saved; this state allows the data to be forwarded to the appropriate requesting processors when the response arrives from the higher level of the hierarchy.

2.2.2. Processor-Interconnect Combining (PIC)

An alternative approach to reducing the cost of the combining network is to determine the combining set prior to inserting the elements into the network. This is the approach taken by schemes that fall under the classification of PIC in the design space.

As with schemes that perform IIC, schemes performing PIC use the interconnection network to distribute the results of combined requests. However, the processor elements, and not the interconnection network, determine the combining set. Blelloch's prefix computation network [3], discussed in Section 2.1, and the control network of the Thinking Machines CM-5 [17] fall into this category.

Another form of PIC in a SIMD paradigm is proposed by Lipovski and Vaughan [19]. This implementation uses a modified carry-lookahead circuit to implement a prefix computation network which distributes the results. The combining set is determined by which processing elements are currently active. The prefix computation network may be extended for operation in a Multiple Instruction Multiple Data (MIMD) paradigm, though the authors do not explicitly state how this might be done.

An alternative technique for combining requests in a MIMD paradigm, proposed by Harrison [8], uses a synchronous prefix computation network. All requests at the same stage in the network combine. Therefore, the entire combining set must be inserted into the network in the same time slot. This is accomplished by broadcasting information about the combinable locations to the processors. Based on this information each processor determines the correct time slot for its request.

In addition to being restricted to use *a priori* knowledge of the combining set, the methods discussed in this section make use of a (parallel) prefix computation network to distribute results. Such networks are conceptually very similar to the Ultracomputer combining network: state must be saved on the forward trip through the network and requests are decombined on the return trip. Recall that this requires sophisticated interconnect nodes and restricts the routes of the return messages. These potentially undesirable features (associated with the result distribution) can be eliminated if we use the processor elements, to perform the (parallel) prefix operation and distribute the results.

2.2.3. Processor-Processor Combining (PPC)

In schemes classified as IIC or PIC, the interconnect, as viewed from a shared memory location, forms a tree. The memory module is the root of the tree and the processors are the leaves of the tree. The nodes of a combining tree are realized by the implicit storage in the interconnect nodes, *i.e.*, wait buffers. Another alternative is to use explicit storage in memory to construct the combining tree. This method of request combining called *software combining* in the literature [5, 31, 32], is classified as PPC since the processors bear full responsibility for the combining of requests: the processors establish the combining set and distribute the results and there are no demands of the network at all.

In software combining, one shared location is divided into L locations which constitute the storage for the nodes of the combining tree. Requests are combined as the processors traverse the combining tree. The result is that $\frac{S}{L}$ processors access each of the L locations, rather than S processors accessing the single location. However, the L locations (nodes of the combining tree) must be distributed across the memory modules in order to alleviate excessive contention for a single memory module.

Yew, Tzeng, and Lawrie show how software combining can be used for barrier operations [32]. Goodman, Vernon and Woest [5] and Johnson [12] extend the work of Yew, Tzeng, and Lawrie to carry out arbitrary *Fetch&*Φ operations with a software combining tree. Tang and Yew also provide several algorithms for traversing a combining tree where the type of memory access determines which algorithm is chosen (*e.g.*, barrier synchronization, semaphore, read combining) [31].

A consequence of implementing the combining tree with explicit memory locations is flexibility in the type of memory access. In addition to variable types of memory accesses, software combining permits the use of networks with arbitrary topologies and relatively unsophisticated nodes.

A requirement of implementing a software combining tree for access to a shared location is that the shared location must be known prior to program execution. Furthermore, if the latency of the combined access is to be minimized, the combining tree must be balanced; this requires *a priori* knowledge of the number of requests that may combine [31]. Moreover, since the combining tree is created based on the maximum number of requests that may combine, the latency to complete the combining operation is influenced by this maximum number: if only one request is accessing the shared location, it must traverse the entire combining tree. For example, in a balanced

(software combining) tree of height *H*, the single request must perform *H* memory accesses, each of which must traverse the interconnection network.

If the processors are responsible for establishing the combining set, they require *a priori* knowledge. If the burden of determining the combining set is placed back on the interconnect, the need for *a priori* knowledge is eliminated.

2.2.4. Interconnect-Processor Combining (IPC)

Heretofore, there are no proposed methods (with the exception of some special cases which we discuss in section 3) of request combining that use an arbitrary interconnection network to determine the combining set and the processor elements to distribute the results. In order to decide if such a scheme is worthy of investigation, the next section compares the issues of implementing combining under each of the classifications.

2.3. Issues in Combining Requests

There are several aspects to request combining that we have touched upon in the previous discussion that we reiterate to motivate IPC. They are:

- *A priori* knowledge of the combining set.
- Restrictions placed on the routes of messages.
- Sophistication of the interconnection network.
- Latency of the combining operation.

The need for *a priori* knowledge of the combining set requires the programmer to specify this information. Restrictions placed on the routing of messages limits the choices for the interconnection network topology. Sophistication of the interconnection network impacts both the cost and performance of the system: a high degree of sophistication might increase the design time and may either increase the latency for non-combining requests or require the addition of a second network. The latency of the combining operation is the time from when a processor generates a combinable request to the time when the result of the request is received.

In the next two sections we look at how each of these issues is affected by which system component establishes the combining set and which component distributes the results. Table 1 summarizes the following discussion.

2.3.1. Determining the Combining Set

The processor elements require *a priori* knowledge of the combinable locations in order to establish the combining set. For example, the nodes of a software combining tree [31] are defined during algorithm design. In contrast the interconnect determines the combining set dynamically by comparing destination addresses of messages. The consequence of introducing comparators is a small increase in the sophistication of the nodes of the interconnect, which may result in a slight increase in the latency to complete the combining operation.

2.3.2. Result Computation and Distribution

Placing the responsibility of result distribution on the interconnect has two disadvantages. The first is an increase in the sophistication of the interconnect as a result of wait buffers, and decombining logic in each interconnect node. The second, and perhaps more important disadvantage, is that the route which a return message may travel is restricted because it must visit the nodes where state was saved on the forward trip through the network.

The primary advantage of using the processor elements to distribute the results of the combining operation is that no restrictions are placed on the routes that messages may travel (no requirement to visit a particular node). Another advantage is that the sophistication of the interconnect nodes is not increased (no need for wait buffers). However, the sophistication of the processor elements (more accurately the processor-network interface) may increase somewhat to handle the protocol needed to distribute results.

The latency of the combining operation, measured in the number of steps needed to carry out the operation, is a disadvantage of using the processors to distribute results. In Section 3 we show that the latency of distributing the results is logarithmic with respect to the number of requests in the combining set, assuming the system does not have broadcast capability.

Based on the above discussion we feel further investigation of IPC, the as yet unexplored area of the design space, is worthwhile. Such schemes would use the interconnect to determine the combining set and use the processor elements to distribute the results. The above discussion also points out the following potential advantages of IPC combining: (i) no *a priori* knowledge of the combining set is required since the interconnect dynamically determines the combining set, (ii) no restrictions on the routes of messages since the processor elements distribute the results, and (iii) the nodes of the interconnection network require only a small amount of sophistication. The potential drawback of such a scheme, as compared to IIC, is the latency of the combining operation.

3. Interconnect-Processor Combining

We now consider implementations of request combining that fall into the unexplored region of the design space, Interconnect-Processor Combining (IPC). We initially consider two flavors of combinable operations: a restricted form of *Fetch&Add* (*F&A*), or *Fetch&Increment* [9, 26], and the general *F&A* operation. In *Fetch&Increment*, or simply *F&I*, all participants add the same, constant value. In the general *F&A*, each participant could be adding a different value.

The rationale for the simpler *F&I* operation is the following: if, in the process of determining the combining set, it is also possible for a participant to determine its overall position in the combining set, i.e., its position in the serial order, then each participant can compute its value without further interactions with other participants. The following example illustrates this point.

Figure 3 shows a system with six processors ($P0$–$P5$) connected to a shared (synchronization) bus. Each processor is assigned one channel in the ‘‘bus’’ (the channel could be a wire in an electronic bus [26] or a specific frequency in an optical bus [9]). A given processor can read all channels, but can write to only its channel.

A processor generates a combinable request and broadcasts its intentions on the bus, by putting a "1" on its channel. All other processors that wish to participate in the access write a "1" on their respective channels. At this point all processors can, by monitoring all the channels on the bus, determine the number and the identity of the processors which are going to participate in the combined access. The combining set is established by determining the participating processors; the ordering in the combining set is statically defined by priorities assigned to the channels.

Suppose four processors $\{P0, P1, P3, P5\}$ would like to perform a *F&I* operation (increment by a constant C) on memory location X. At the point that the four processors have indicated their intentions to access X, the priority chain $\{S0, S1, S2, S3, S4, S5\}$ is: $\{110101\}$. Each of the four processors then determines how many processors ahead of it in the priority chain are also participating in the combining operation. For example, $P5$ determines that there are three processors ahead of it in the priority chain. $P3$ sees two, $P1$ sees one, and $P0$ sees zero since it is the highest priority processor participating in the combined access. $P0$ takes responsibility for accessing X from memory. When $P0$ accesses X the other processors also read the value from the memory bus. Each processor is restricted to adding the same constant, C, to the shared location, and therefore each processor may compute its value locally based on the number of participants preceding it in the priority chain. $P0$ receives X, $P1$ computes $X+C$, $P3$ computes $X+2C$, $P5$ computes $X+3C$, and memory receives $X+4C$. One processor (or the memory controller

who could also be monitoring the bus) takes responsibility for computing $X+4C$ and updating memory.

The method described above was proposed independently for an electrical bus by Sohi, Smith, and Goodman [26] and for an optical bus by Hiedelberger, Rathi, and Stone [9]. The ease with which combining can be carried out (in special cases), prompted Freudenthal and Gottlieb to investigate the use of the *Fetch&Increment* operation in place of the more general *F&A* operation [4].

When broadcast is not an option, some other method must be used to determine the combining set and to carry out the prefix operation on the combining set. Unlike networks with broadcast, there is no easy way to merge the creation of the combining set and its ordering in an arbitrary interconnection network. Here, we must continue to separate the creation of the combining set, and the implementation of the prefix operation. Since a prefix operation has to be carried out on the combining set, regardless of whether it is to order the elements (as would be necessary in case of *F&I* operations), or to distribute the results (as would be necessary in case of *F&A* operation), we see no potential implementation advantage of a *F&I* operation over a *F&A* operation in an arbitrary network. Therefore we can continue our further discussion with *F&A* operations.

Our proposal for IPC in an arbitrary network uses the network to create the combining set; we choose to represent the combining set as a linked list, though it is conceivable that other structures could also be used. The processors then use this structure to interact with each other and to carry out the prefix operation. We expand on each of the two functions, setting up the list and distributing results, below.

When a processor submits a *Fetch&*$\Phi(X, V_i)$ request, it generates a message that is sent over the interconnect to a destination containing X. The message consists of at least four fields: an address (X), a value field (V), and two pointers, as shown in Figure 4. The pointers indicate the head and the tail of a list of processors, *i.e.*, a combining set, accessing the memory location specified in the message. Initially the processor generating the request is the only member of the combining set.

When two messages destined for the same memory location meet at a node of the interconnect, the two messages are combined, as illustrated in Figure 5. As a result of the combining, a forward message is sent on to the destination location X, and the head and tail fields are updated to indicate the head and tail of the new combining set. The new combining set is the union of the combining sets of the two combined messages. (If ALUs are present, then the value field of the forward message is updated to reflect the Φ operation on the value fields of the combined messages.) A link message is sent to the processor at the tail of the combining set of the first message, with

instructions to create a link to the processor at the head of the combining set of the second message. (The route taken by the link message depends upon the topology of the network. For example, in an Omega network with uni-directional links, the link message would travel in the forward direction to some arbitrary destination, and reflect off that destination to go back to the appropriate processor). Details of the combining operation at each node are shown in Figure 5. It is important to note that *no state is saved in the network at the point where messages are combined*.

Figure 6 illustrates the above with an example. Assume that processors *A* and *B* generate *F&A* requests to location *X*, as shown in Figure 6. When the two requests are merged, a message is sent to processor *A* indicating that processor *B* is the head of a list of processors that are also participating in the *F&A* operation. At this point processor *B* is the only member of the list being added. Processor *A* now has a pointer to processor *B*; this forms the new linked list representing a combining set for location *X*. The fields of the forward message are the memory location (*X*), the head of the list (*A*), and the tail of the list (*B*). (The values fields have not been shown in the figure to prevent a clutter.)

If we assume that processors *C* and *D* also formed a combining set for location *X*, then we have two lists with two members each and two messages en-route to memory. When these two messages merge, a message is sent to processor *B* indicating that processor *C* is the head of a linked list of processors also participating in the *F&A* operation. This creates a single combining set which is the union of the two original combining sets. The message that is forwarded to memory has its fields set to the head of the first list (*A*) and the tail of the second list (*D*). We now have a single combining set represented by one message going to memory. When memory receives the request, it returns the value stored at location *X* to the head of the list (*A*) and also sends a message to the tail of the list, (*D*), indicating it is responsible for providing the final result to memory. This allows the results of the *F&A* operations to be distributed to the processors, as we shall discuss shortly.

When a *F&A* request reaches the destination *X*, the head and tail fields of the request point to the head and the tail of the combining set. We have several options on how to proceed. If ALUs were present in the network nodes where combining took place, and had been used to update the value field of the forward message (as shown in Figure 5), then the value field of the message reaching *X* contains the (final) result of the prefix operation applied to all the value fields of the members of the combining set (determined by the head and tail pointers), even though the individual processor (or intermediate) results of the prefix operation are not yet known. The old value of memory location *X* can be returned to the head, and used by the prefix operation (described later) to determine the results for

the members of the combining set; memory location *X* could be updated with the final value (old value of *X* plus the value field of the message reaching *X*). Further accesses to the same location, *X*, can proceed *while* the results of the prefix operation on the (first) combining set are being carried out.

If ALUs were not present in the network nodes where combining took place, then the final value of the prefix operation on the combining set can not be determined when the (combined) message reaches memory location *X*. We must wait until the prefix operation on the combining set is complete before memory can receive the up-to-date value. What happens to further requests to location *X* during that time? There are two options. The first option is to simply lock the memory location *X* until the final result of the prefix operation is known (the tail of the combining set is responsible for unlocking *X*). This makes further requests to *X* wait, building a second combining set while waiting for the prefix operation on the first to complete. Locking *X* is fairly easy to do but, as we shall see in section 4, has implications on performance.

The second option is to append the ‘‘new’’ combining set onto the ‘‘old’’ combining set (on which a prefix operation is in progress), thereby creating one combining set, and make the prefix operation robust enough so that it can operate an variable-size combining sets. When appending a ‘‘new’’ combining set onto an ‘‘old’’ combining set, care must be taken to avoid potential race conditions. In particular, the message from the memory to the tail of the old combining set (telling it to create a pointer to the head of the new combining set, thereby continuing the prefix operation) could arrive *after* the prefix operation on the old combining set has been completed, and the (final) result is on its way to memory. The message must be reflected back to memory, which can then supply the up-to-date value to the new combining set, and allow it to start its prefix operation. The message handling protocol must take the possibility of such race conditions into account, and take appropriate action to prevent incorrect operation. While at first glance appending onto a combining set on which a prefix operation is in progress appears to be a good idea, it also has implications on performance as we shall see in section 4.

From the above discussion we can discern the amount of sophistication required in each of the interconnect nodes. The use of comparators is necessary to determine if two messages are destined for the same memory location. A small amount of additional logic is required to construct the new message that is sent back to the processors and to update the tail of the message forwarded to memory. ALUs are needed if we expect to update memory with the final value immediately.

Now we consider how a prefix operation could be carried out on the combining set, which is represented as a linked list. For the purposes of our discussion on result distribution, assume that processors can only use point-to-point messages, reflected off the memory modules if necessary, for communication. The naive method of distributing the results is to start at the head of the list and sequentially move from one node to the next in the list. Although there is no hot-spot when using the naive method to distribute results, the messages are unnecessarily serialized. To eliminate this serialization for the distribution of results we turn to the literature on parallel applications to see how the prefix operation on the combining set can be carried out in parallel.

Several algorithms for performing a parallel prefix computation on a linked list exist in the literature [10, 13, 15, 20]. Most of the algorithms are concerned with the case of having more nodes in the linked list than processors available. In our case the number of nodes in the linked list is equivalent to the number of processors that are participating in the combined access. Therefore, we use the all partial sums algorithm given by Hillis and Steele [10] and shown in Figure 7. The algorithm uses recursive doubling: each iteration of the loop performs half as many operations as the previous iteration until the entire computation is complete. This algorithm has the advantage that the result is computed in O(log$S$) steps, where S is the number of nodes in the list, i.e., the cardinality of the combining set. An example of a partial prefix sum computation for a list with eight nodes is shown in Figure 8. The array *initial[]* is indexed by processor number and contains the number of the next processor in the initial linked list, while the array *forward[]* indicates the processor that is communicated with during the current iteration of the algorithm.

It is important to note that the recursive doubling algorithm is defined for a SIMD machine, and therefore appropriate synchronization must be added for MIMD operation. Figure 9 shows the MIMD version for result distribution in IPC, the reader is referred to[16] for further discussion of the transformation from SIMD to MIMD. Processors also require a limited amount of memory (O(log S)) for storing the pointers to neighboring processors in the combining set.

We have not tried to exhaust the methods of IPC, nor have we addressed all of the issues involved in IPC. There are undoubtedly alternative data structures for maintaining the combining set and associated algorithms for distributing the results. However, we do provide a viable solution that deserves further investigation.

4. Evaluation of IPC

To investigate the overall system performance using IPC, we developed a simulator of a multiprocessor system that performs IPC. An enhanced Omega network establishes the combining sets as described in Section 3, and the processors use the MIMD version of the recursive doubling parallel prefix algorithm to distribute the results of the combined access.

The simulator consists of three distinct parts: processor elements, interconnection network, and memory modules. Table 2 summarizes the parameters of our system. The processor elements are memory reference generators with the additional code required to distribute the results of the combined access. Processors generate a memory reference each cycle with probability *r*, providing the network can accept the request. Of these memory references, *h* percent are directed at a single hot memory location [22, 32]. Each processor may have only one outstanding *F&A* request, but unlimited outstanding uniform requests. However, a processor does not generate any requests when there is an outstanding *F&A*. Yew *et. al.* [32] call this the *limited-variable* access pattern.

The interconnection network used in our simulations is an enhanced Omega network. Each node has bidirectional links[1], and a queue exists between the forward network and the reverse network. Therefore, each switch in the forward network has two inputs and three outputs, whereas each switch in the reverse network has three inputs and two outputs. We assume that each message is a single packet and that no buffering occurs if there is no contention in the network. Two messages combine only if one is buffered and we assume a full comparator is used to determine if an arriving message is combinable with any buffered message. Only pairwise combining is carried out at each node.

We simulated a system with 256 processors connected to 256 memory modules. In all of our simulations we vary *h* from 0-32 percent and *r* from 20-100 percent. Figure 10 shows the average latency and the maximum bandwidth when no combining is performed. As previously shown in [22] and [32], there is a point of saturation after which bandwidth ceases to increase and latency increases.

In our first experiment, we implemented IPC, without ALUs in the network nodes, and with locking the memory until the prefix operation on the current combining set is complete. Figure 11a presents the latencies of all

[1] The use of bidirectional links is a design choice we made, it is still possible to use IPC on networks with separate forward and reverse networks. The messages that link the combining sets together would have to be reflected off the memory modules. However, under these conditions IPC still allows the use of adaptive routing techniques.

requests, and saturation bandwidths, for varying hot rates. In comparing Figs 10 and 11 (note the different scales), we see that the latencies have decreased for small values of *h*, but increased for large values, and that the saturation bandwidth has also decreased for all values of *h*. The reason for this is twofold. Since we lock the memory location, we prevent the second combining set from starting its prefix operation. Consequently, the hot requests in the second combining set are stalled for a long time waiting for the prefix operation on the first combining set to complete. This is illustrated in Figure 12. Fig 11b presents the latency of hot requests, while Fig 12 presents the average amount of time a hot request spends waiting for the lock on memory to be released. Comparing Figs 11b and 12, we can see that the time waiting for the lock to be released can be a significant portion of the overall hot request latency.

The next experiment that we carried out involved appending requests, as they arrive at memory, onto the end of a combining set on which a prefix operation is already in progress. In this case, requests do not have to wait for the prefix operation on the previous combining set to complete, rather they join in to form a larger combining set, and join in the existing prefix operation. Our expectation was that by reducing the waiting time, we would decrease the hot request latency. Unfortunately, our experimental results, presented in Figure 13, indicate otherwise. Fig 13a shows the saturation bandwidth and latency of all requests, and Fig 13b shows the latency of hot requests, for varying hot rates. The reason for the disappointing results for this experiment is that while appending requests to a combining set decreases the time spent by the requests waiting at memory, it actually increases the time to carry out the prefix operation, since the hot requests can get appended one by one, and the prefix operation can't complete as long as the requests are being appended (no participant has its value ready until the entire prefix operation is complete). An extreme case is when the hot requests are appended one by one, and the parallel prefix degenerates to serial prefix, but one in which no processor can use its value until the entire prefix operation is complete. Moreover, since there is no deterministic pattern with which the requests are appended, there is no uniform trend in the results. Based upon the results of this experiment, we feel that appending new arrivals onto an existing combining set, on which a prefix operation is already in progress, is something that should be done with caution. More study is needed in this area.

Our final experiment with IPC uses ALUs in the network nodes. In this case, since the final result of the prefix operation is available before its intermediate results are, there is no need to make succeeding hot requests wait while a prefix operation is in progress on a previous combining set. Prefix operations could be in operation on

multiple combining sets simultaneously. Figure 14a presents the latencies and saturation bandwidths of all requests, and Figure 14b presents the same for hot requests, for varying hot rates. When we compare figures 10 and 14 we see that IPC is quite effective in reducing the degradation due to hot spots. IPC has reduced the latency for all requests, improved overall network bandwidth, as well as reduced the latency for hot requests. Interesting, the latency for hot requests is lower the higher the hot rate (Figure 14b). This is because with a higher hot rate a larger combining set is established, and therefore more of the hot requests can be serviced in parallel. Overall, the proposed method for IPC is not as effective as the Ultracomputer style of combining (equivalent results for the Ultracomputer-style of combining can be found in the paper by Pfister and Norton [22]), however our results suggest that it is an option worth considering.

5. Summary and Conclusions

Unconstrained yet synchronized access to shared memory locations is achieved by combining requests. We formulated a taxonomy for the various techniques of request combining by separating the combining operation into two parts: establishing a combining set and distributing the results. This taxonomy divides the request combining design space into four regions defined by which system component (processor elements or interconnection network) performs each of the tasks of request combining. The classification of the existing implementations of request combining has enabled us to obtain four primary issues that an implementation of request combining must address: the need for *a priori* knowledge of the requests to combine, the complexity of the interconnect nodes, restrictions placed on the routing of messages in the interconnect, and the latency to complete the combined access. We have shown that the current methods of request combining occupy only three of the four regions of the design space and do not address the first three issues satisfactorily. We presented an implementation that lies in the as yet unexplored region, known as Interconnect-Processor Combining (IPC), in which the interconnect establishes the combining set and the processor elements distribute the results. We show that implementations in this area of the design space have the following advantages:

- No need for *a priori* knowledge of the combinable location.
- No restrictions on the routing of messages.
- Low sophistication of interconnect nodes.

We also carried out several experiments to assess the benefits of IPC, and observed that IPC is an effective technique to alleviate the degradation caused by serial access of memory, but it must be used with care. Some implementations of IPC could degrade the latency of hot requests intolerably, thereby reducing network bandwidth, even though they reduce the latency of uniform requests.

While we feel that the flexibility and the performance benefits of IPC make it an attractive design choice, more research is needed in this area before a definitive answer can be obtained. Many more alternatives for IPC need to be investigated, including the choice of the data structure used to maintain the combining set, as well as the algorithm used to carry out the prefix operation on the combining set. (One step in this direction is a recent thesis by Johnson where he investigates the use of IPC to build a tree to implement a scalable cache coherence scheme [12].) More direct comparisons between the different forms of combining, using real application workloads, and different network topologies, also need to be done so that we can get a better picture of the cost-performance benefits of the various techniques for request combining.

References

[1] Almasi, G. S. and Gottlieb, A., *Highly Parallel Computing*. Redwood City, CA: Benjamin/Cummings Publishing Company, Inc., 1989.

[2] Arvind, and Iannucci, R. A., ''A Critique of Multiprocessing von Neumann Style,'' in *Proc. 10th International Symposium on Computer Architecture*, Stockholm, pp. 426-436, 1983.

[3] Blelloch, G. E., ''Scans as Primitive Parallel Operations,'' *IEEE Transactions on Computers*, vol. C38, 11, pp. 1526-1538, November 1989.

[4] Freudenthal, E. and Gottlieb, A., ''Process Coordination with Fetch-and-Increment,'' *Proceedings ASPLOS-IV*, pp. 260-268, April 1991.

[5] Goodman, J. R., Vernon, M. K., and Woest, P. J., ''A Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor,'' in *Proceedings ASPLOS-III*, Boston, MA, pp. 64-73, April 1989.

[6] Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., and Snir, M., ''The NYU Ultracomputer -- Designing a MIMD, Shared Memory Parallel Machine,'' *IEEE Transactions on Computers*, vol. C-32, pp. 175-189, February 1983.

[7] Hagersten, E., Landin, A., and Haridi, S., ''DDM — A Cache-Only Memory Architecture,'' *IEEE Computer*, pp. 44-54, September 1992.

[8] Harrison, M. C., ''Synchronous Combining of Fetch-And-Add Operations,'' *Ultracomputer Note #71*, April 1984.

[9] Heidelberger, P., Rathi, B. D., and Stone, H. S., ''A Device for Performing Efficient Task Distribution with a Bus Connection,'' *IBM Research, Technical Disclosure YO889-0053*, vol. 20, January 1989.

[10] Hillis, W. D. and Steele, G. L., ''Data Parallel Algorithms,'' *CACM*, pp. 1170-1183, December 1986.

[11] Hsu, W. T. and Yew, P.-C., ''An Effective Synchronization Network for Hot-Spot Accesses,'' *ACM Transactions on Computer Systems*, vol. 10, pp. 167-189, August 1992.

[12] Johnson, R. E., ''Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors,'' Ph. D. Thesis, Department of Computer Science (Technical Report #1136), University of Wisconsin-Madison, Madison, WI 53706, February 1993.

[13] Kogge, P. M. and Stone, H. S., ''A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations,'' *IEEE Transactions on Computers*, vol. C-22, pp. 786-793, August 1973.

[14] Kruskal, C. P., Rudolph, L., and Snir, M, ''Efficeint Synchronization on Multiprocessors with Shared Memory,'' *ACM Transactions on Programming Languages and Systems*, vol. 10, 4, pp. 579-601, October 1988.

[15] Ladner, R. E. and Fischer, M. J., ''Parallel Prefix Computation,'' *JACM*, vol. 27, pp. 831-838, October 1980.

[16] Lebeck, A. R., ''Request Combining in Multiprocessors with Arbitrary Interconnection Networks,'' M. S. Thesis, Dept. of Computer Sciences, University of Wisconsin-Madison, Madison, WI, 1991.

[17] Leiserson, C. E., ''The Network Architecture of the Connection Machine CM-5,'' *Proc. ACM Symposium on Parallel Algorithms and Architectures*, July 1992.

[18] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J., ''The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor,'' *Proceedings 17th Annual Symposium on Computer Architecture*, May 1990.

[19] Lipovski, G. J. and Vaughan, P., ''A Fetch-and-Op Implementation for Parallel Computers,'' in *Proc. 15th Annual Symposium on Computer Architecture*, Honolulu, HI, pp. 384-392, June 1988.

[20] Lubachevsky, B. D. and Greenberg, A. G., ''Simple, Efficient Asynchronous Parallel Prefix Algorithms,'' *Proc. 1987 International Conference on Parallel Processing*, pp. 66-69, August 1987.

[21] Pfister, G. F., Brantley, W. C., George, D. A., Harvey, S. L., Kleinfelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A., and Weiss, J., ‘‘The IBM Research Parallel Processor Prototype (RP3): introduction and architecture,’’ *Proceedings 1985 International Conference on Parallel Processing*, pp. 764-771, August 1985.

[22] Pfister, G. F. and Norton, V. A., ‘‘’Hot-Spot’ Contention and Combining in Multistage Interconnection Networks,’’ *IEEE Transactions on Computers*, vol. C-34, pp. 943-948, October 1985.

[23] Scott, S. L., ‘‘Toward the Design of Large-Scale Shared-Memory Multiprocessors,’’ Ph. D. Thesis, Department of Computer Science (Technical Report #1100), University of Wisconsin-Madison, Madison, WI 53706, July 1992.

[24] Smith, A. J., ‘‘Cache Memories,’’ *ACM Computing Surveys*, vol. 14, pp. 473-530, September 1982.

[25] Smith, B., ‘‘Architecture and Applications of the HEP Multiprocessor Computer System,’’ *Proceedings of the Int. Soc. for Opt. Engr*, pp. 241-248, 1982.

[26] Sohi, G. S., Smith, J. E., and Goodman, J. R., ‘‘Restricted Fetch&Φ Operations for Parallel Processing,’’ in *Proc. 3rd International Conference on Supercomputing*, Crete, Greece, pp. 410-416, June 1989.

[27] Stenstrom, Per, Joe, Truman, and Gupta, Anoop, ‘‘Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures,’’ in *Proc. 19th International Symposium on Computer Architecture*, Gold Coast, Australia, pp. 80-91, May 1992.

[28] Sullivan, H. and Cohn, L., ‘‘Shared Memory Computer Method and Apparatus,’’ *U.S. Patent 4,707,781*, November 1987.

[29] Tzeng, N. F., ‘‘Design of a Novel Combining Structure for Shared-Memory Multiprocessors,’’ *Proc. 1989 International Conference on Parallel Processing*, pp. I-1 to I-8, August 1989.

[30] Wilson, A. W., ‘‘Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors,’’ in *Proc. 14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 244-252, June 1987.

[31] Yew, P.-C. and Tang, P., ‘‘Software Combining Algorithms for Distributing Hot-Spot Addressing,’’ *Journal of Parallel and Distributed Computing*, October 1990.

[32] Yew, P.-C., Tzeng, N.-F., and Lawrie, D. H., ‘‘Distributing Hot-Spot Addressing in Large Scale Multiprocessors,’’ *IEEE Transactions on Computers*, vol. C-36, pp. 388-395, April 1987.

List of Tables

Request Combining Issues				
Combining Classification	Determining Combining Set		Result Computation and Distribution	
	a priori Knowledge	Restricted Routes	Interconnect Sophistication	Latency
IIC	No	Yes	High	O(L)
PIC	Yes	Yes	High	O(L)
PPC	Yes	No	NA	O(HL)
IPC	No	No	Low	O(LlogS)

NA = Not Applicable
H = Height of Software Combining Tree
S = Size of Combining Set
L = Latency of Interconnection Network

Table 1: Request Combining Comparison

The existing techniques of request combining do not cover the entire design space and require either *a priori* knowledge or restrict the routes of messages. Techniques that use the interconnection network to distribute results require a high degree of sophistication in the network nodes. Although there are currently no implementations, IPC does not have the above limitations, but has the potential drawback of increased latency to complete the combining operation.

Simulation Parameters and Values	
Parameter	Value
message length	1
uniform request rate (h)	0.2 - 1.00
hot request rate (r)	0.0 - 0.32
outstanding uniform requests	unlimited
outstanding hot requests	1
switch buffers	8
processor buffers	4
memory buffers	4
memory cycle time	1
total # of requests	1,000,000
# of processors	256

Table 2: Simulation Parameters

List of Figures

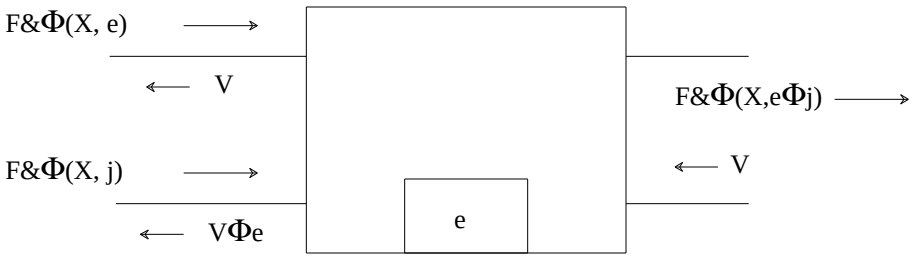

Fig. 1: Request Combining in the Ultracomputer

This figure is adapted from [6]. When two *Fetch&Φ* requests combine, state is saved in a wait buffer until the response returns. This type of combining requires comparators, wait buffers, and an ALU in each network node.

Result Computation and Distribution	**Determines Combining Set**	
	Processors	Interconnect
Processors	Processor-Processor Combining (PPC)	Processor-Interconnect Combining (PIC)
Interconnect	Interconnect-Processor Combining (IPC)	Interconnect-Interconnect Combining (IIC)

Fig. 2: Request Combining Design Space

Request combining is a two part operation; establishing a combining set and distributing the results by performing a prefix computation. The techniques for combining requests are classified based on which system component, processors or interconnect, performs each of the tasks.

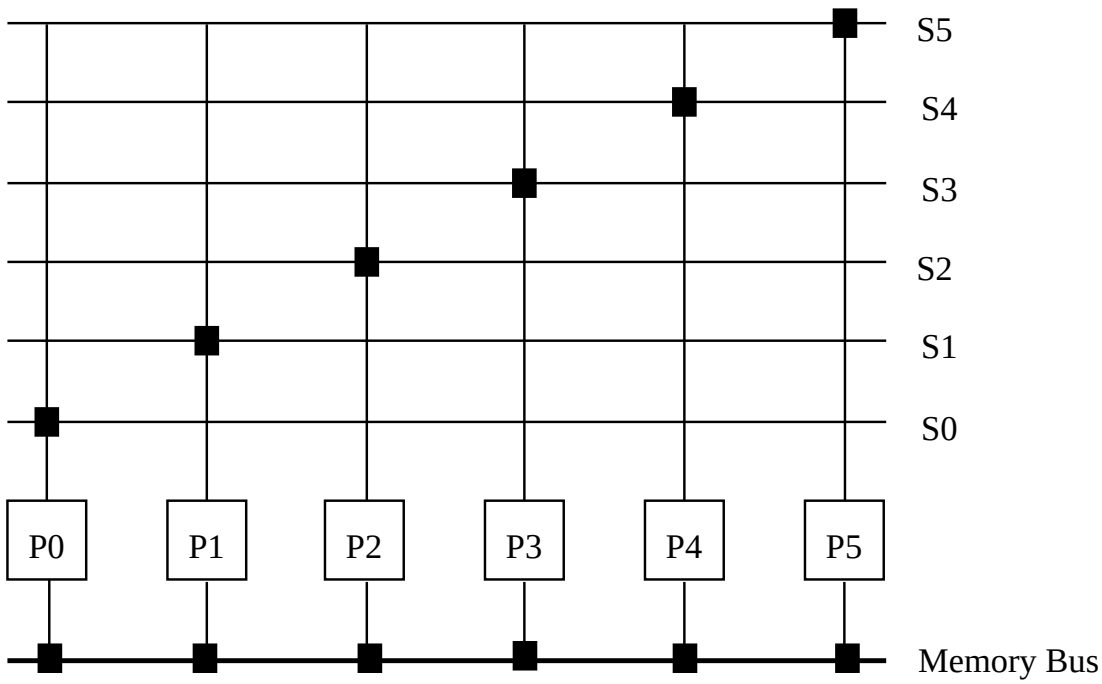

Fig. 3: Priority Chain Implementation

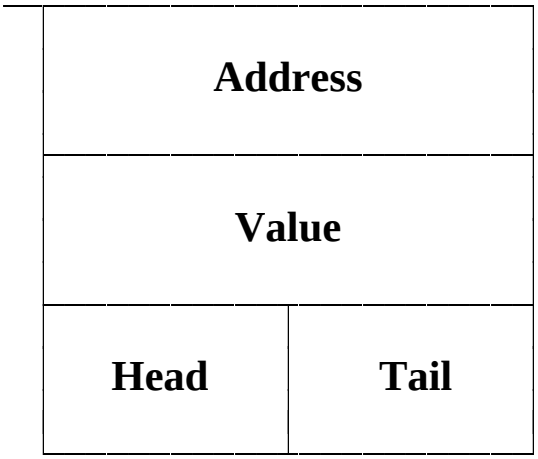

Fig. 4: *Fetch&Φ* Message Format

```
if (m1.address == m2.address && m1.type == Fetch&Φ
  && m2.type == Fetch&Φ)
    link_msg.dest := m1.tail
    link_msg.head := m2.head
    link_msg.type := LINK
    m1.tail := m2.tail
    m1.value := m1.value Φ m2.value    /* if ALUs in nodes */
    SEND_FORWARD(m1);
    SEND_REVERSE(link_msg);
    DELETE(m2)
fi
```

Fig. 5: Merging two messages at an interconnect node

When two combinable requests arrive at a node of the interconnect, the messages are merged together and the respective combining sets linked together.

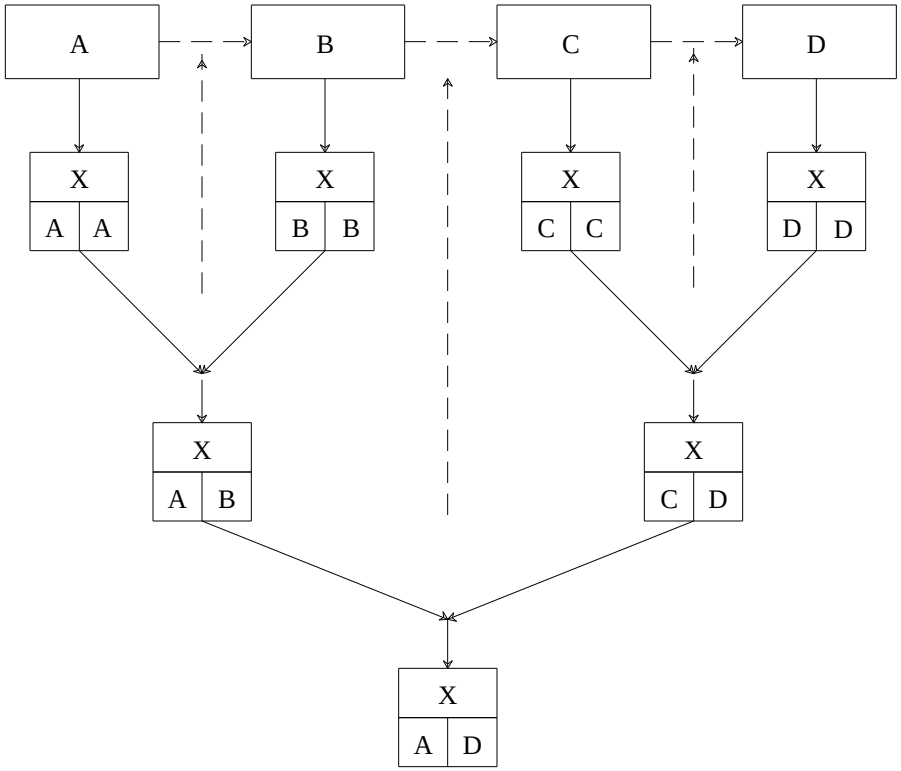

Fig. 6: Establishing the Combining Set as a Linked List

Combining requests involves merging the two messages into one message, which is forwarded to memory. Also, the two lists are linked together by sending a message back to the processors.

```
for all k in parallel do
  forward[k] := initial[k]
  while (forward[k] != null) do
    val[forward[k]] := val[k] + val[forward[k]]
    forward[k] := forward[forward[k]]
  endwhile
endfor
```

Fig. 7: All Partial Sums of a Linked List

This algorithm, adapted from [10], can be used to distribute the results to the members of the combining set.

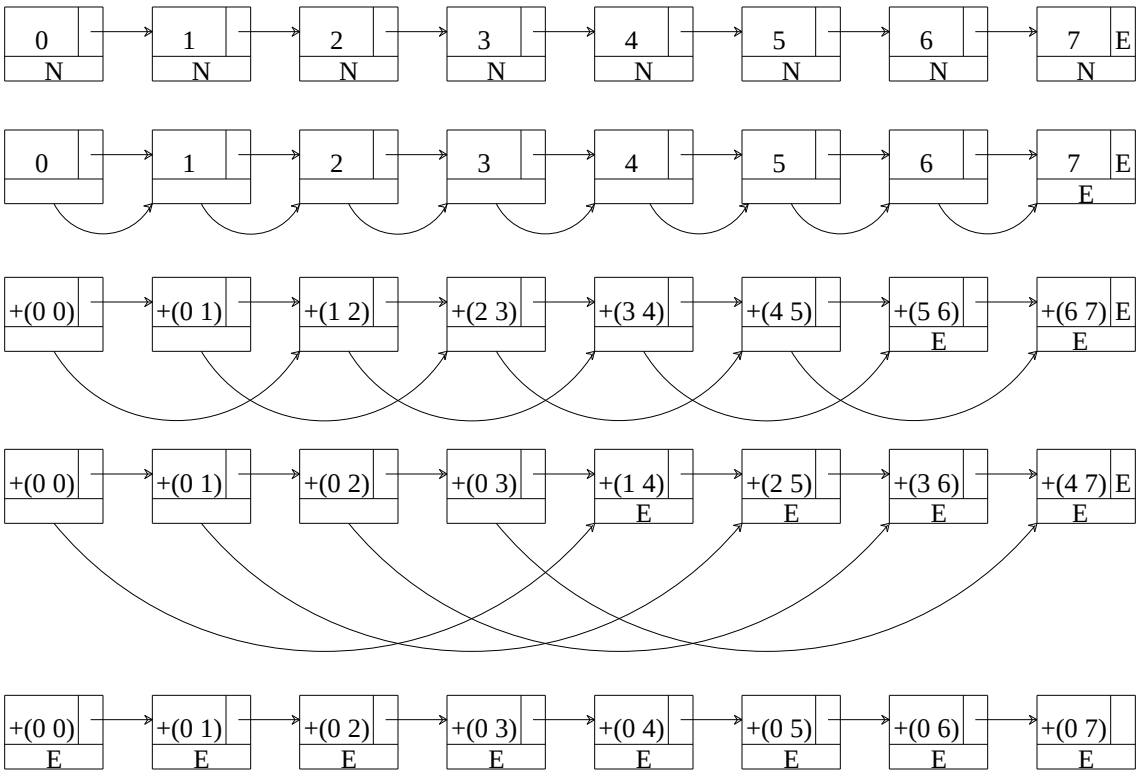

Fig. 8: Partial Prefix Sum Computation

This figure is adapted from Figure 5 in [10] and shows parallel partial prefix sum computation for a list with eight members. The values E and N indicate End and NULL respectively.

```
while chum ≠ null do
    send(chum,myinc_val)
    receive(rev,Φ_val)
    value := Φ_val + value
    myinc_val := Φ_val + myinc_val
    send(rev,chum)
    receive(chum,newchum)
    chum := newchum
endwhile
```

Fig 9: MIMD Distribution of Results

The SIMD version for computing the all partial sums of a linked list can be written in a MIMD fashion that uses only point-to-point synchronization [16].

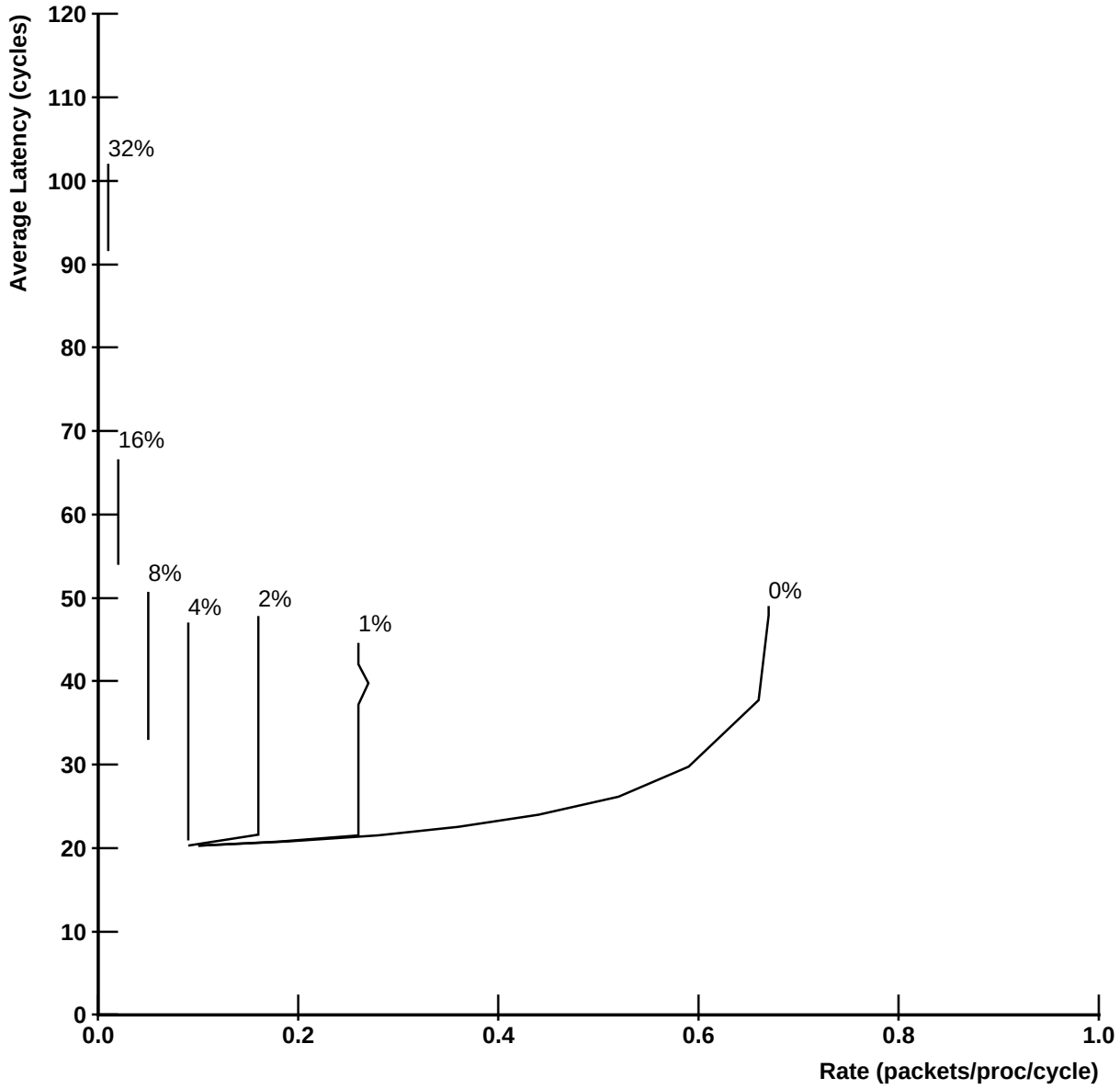

Fig. 10(a): All Requests

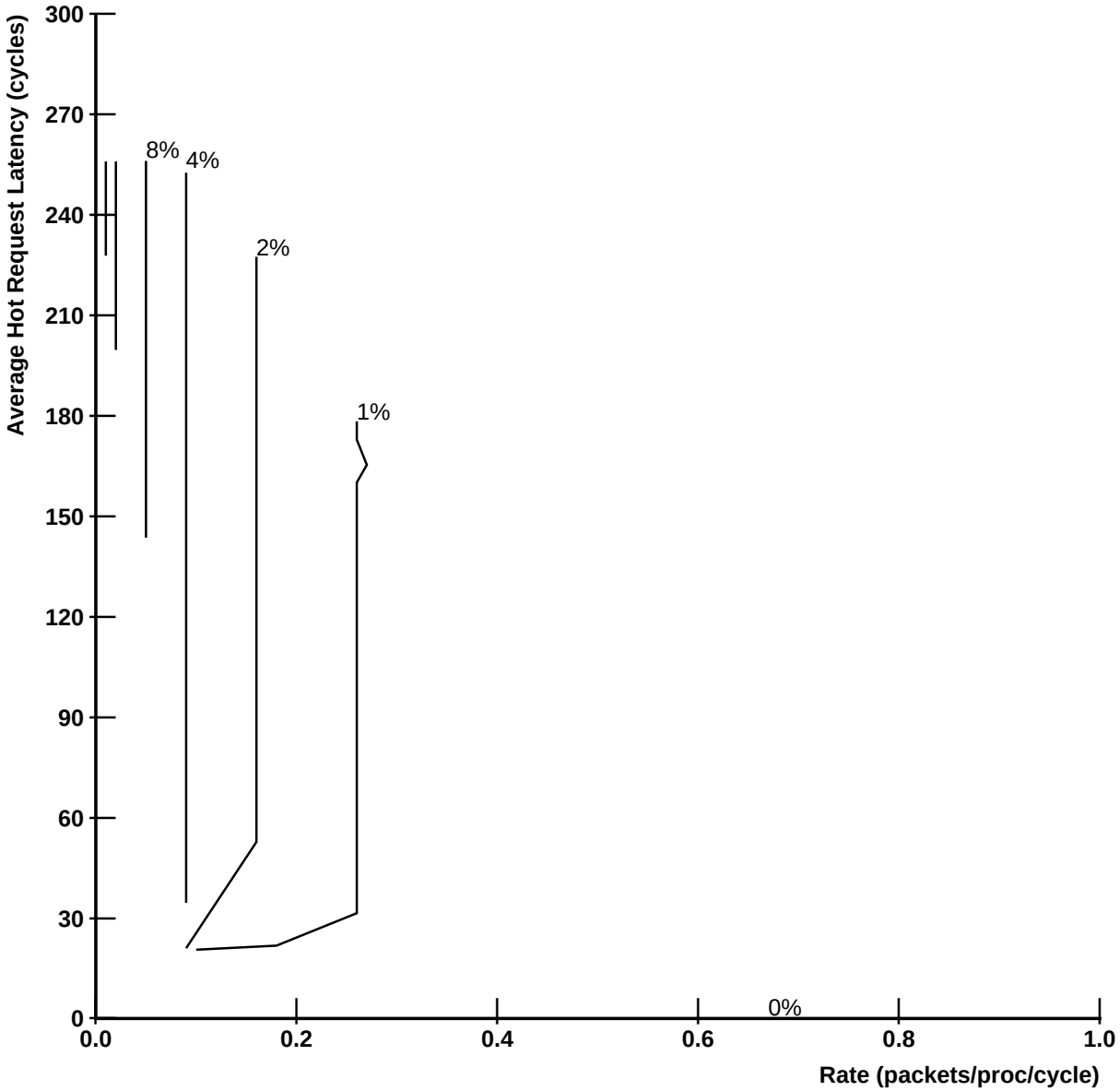

Fig. 10(b): Hot Requests

Fig. 10: Average Latency vs Maximum Bandwidth: No Combining

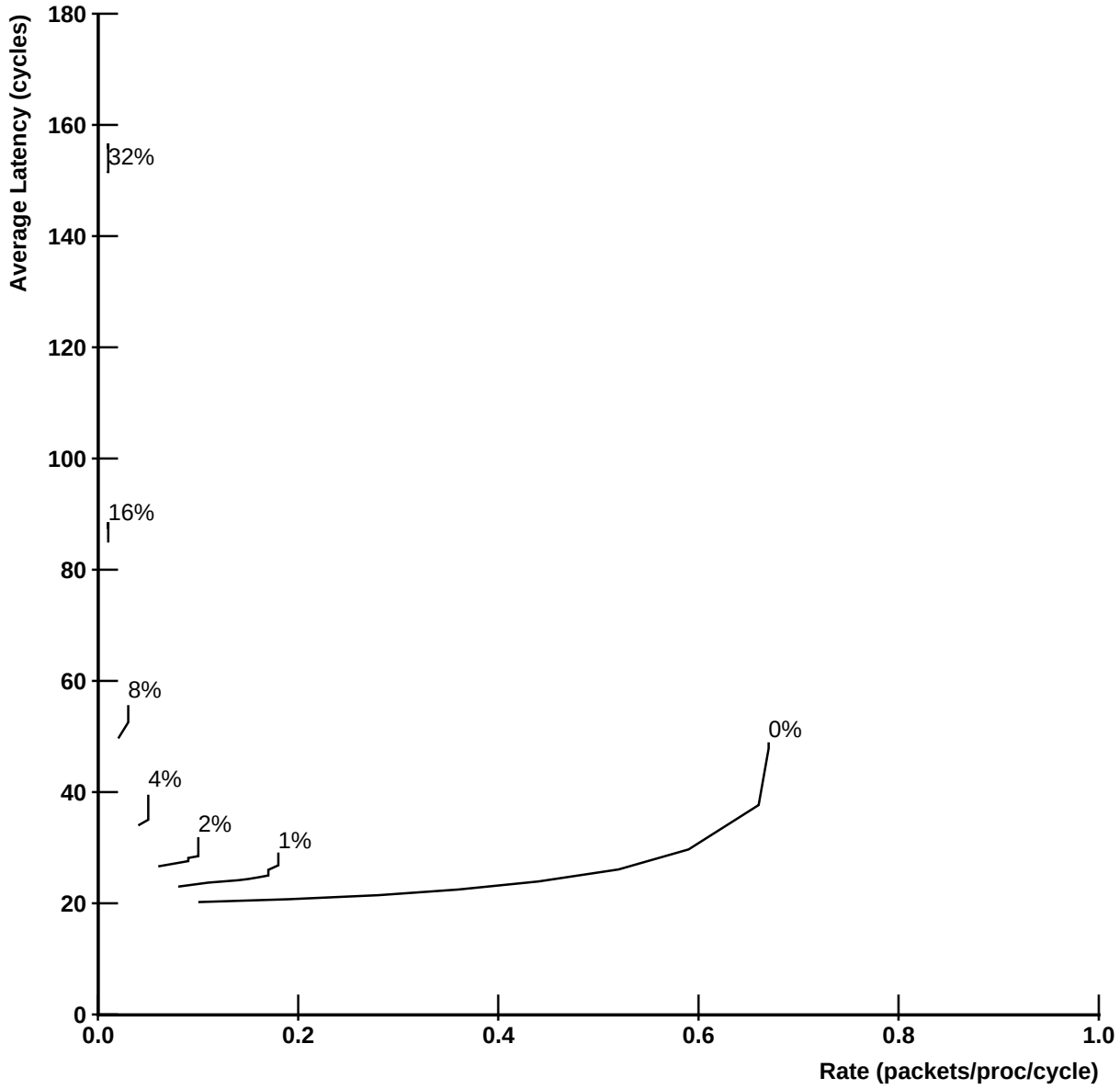

Fig 11(a): All Requests

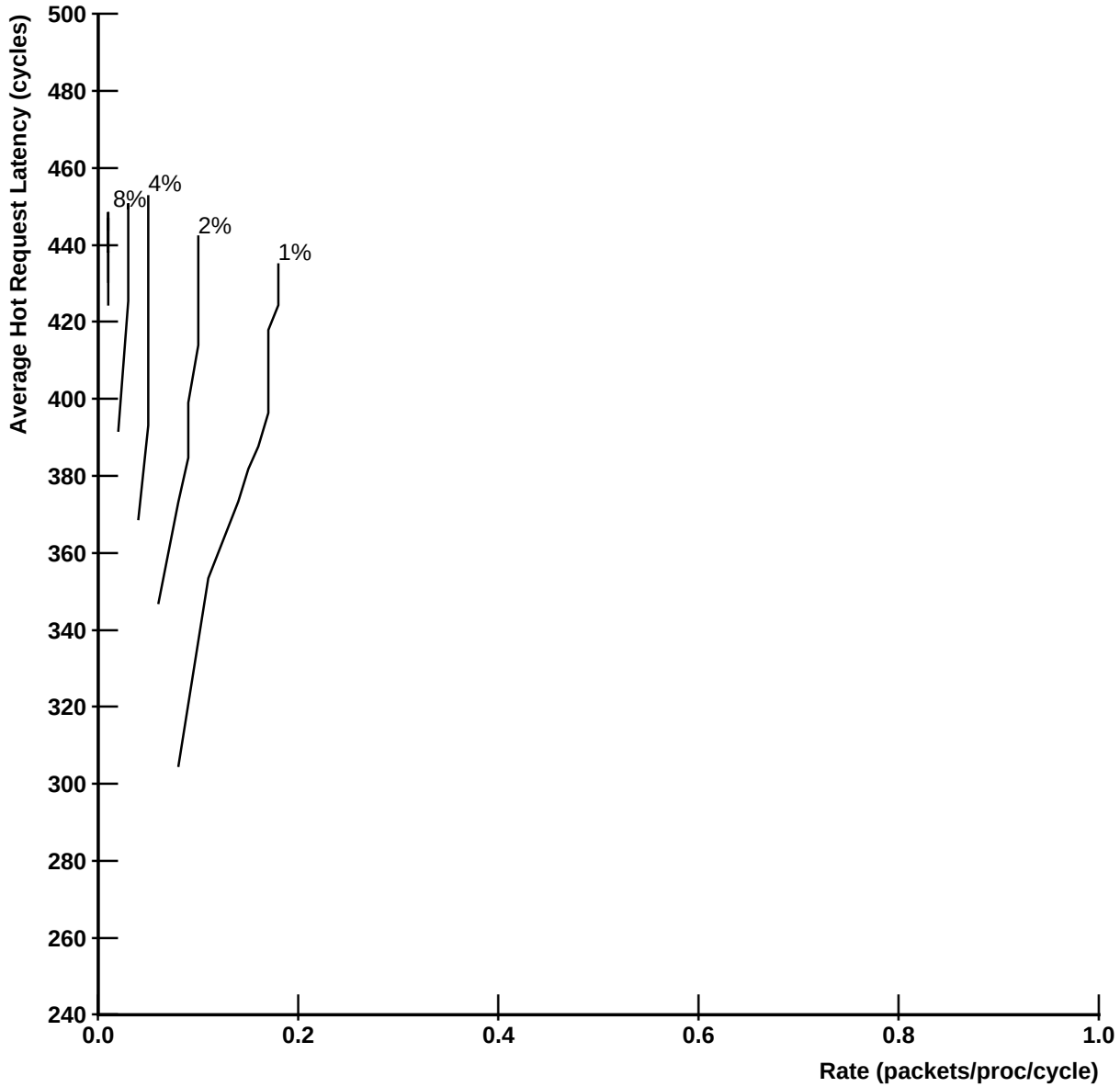

Fig 11(b): Hot Requests

Fig 11: Average Latency vs Maximum Bandwidth: Lock Memory

Average latency and maximum bandwidth when combining is enabled and the memory location is locked while the results are distributed.

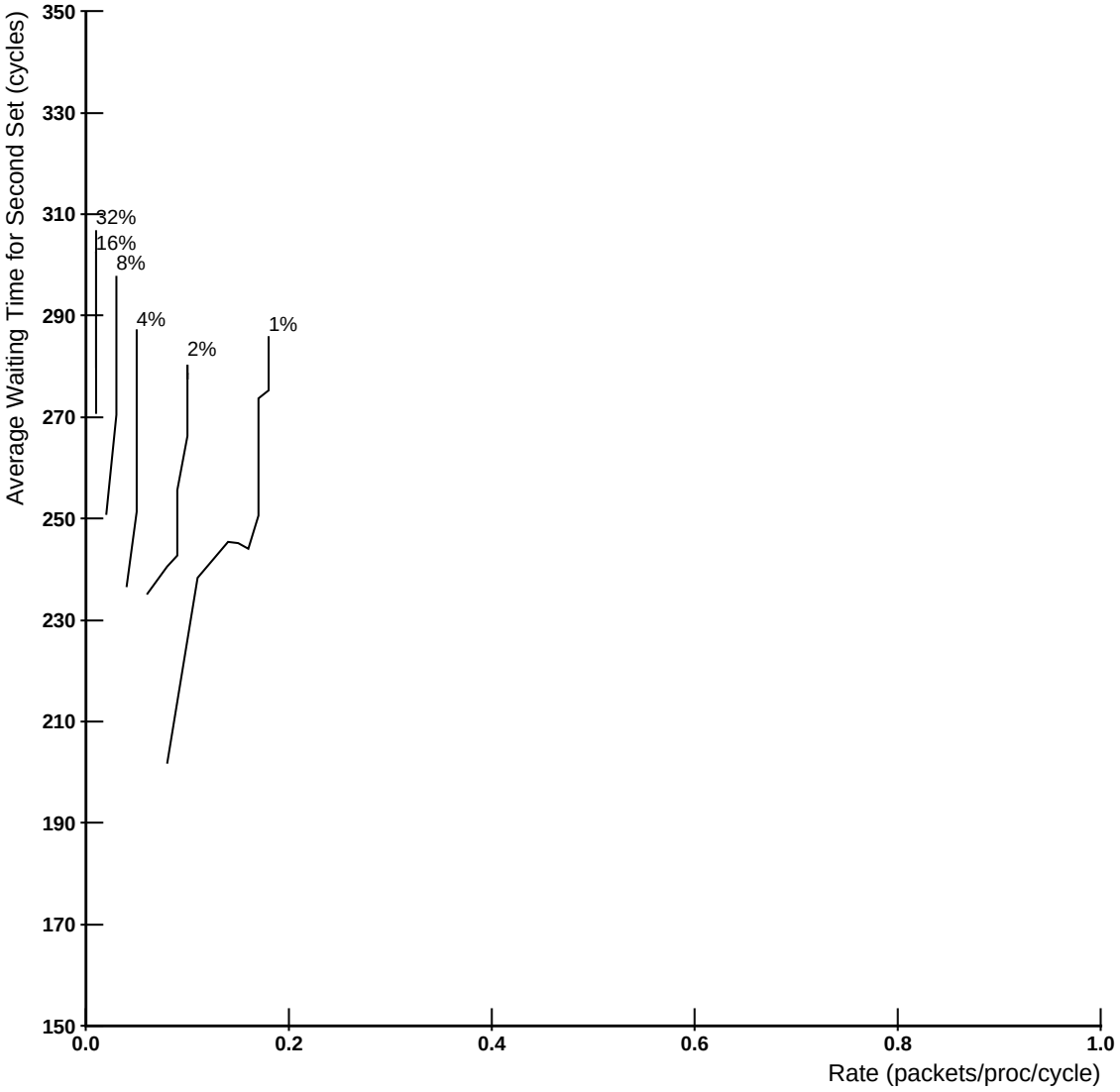

Fig. 12: Waiting Time at Memory

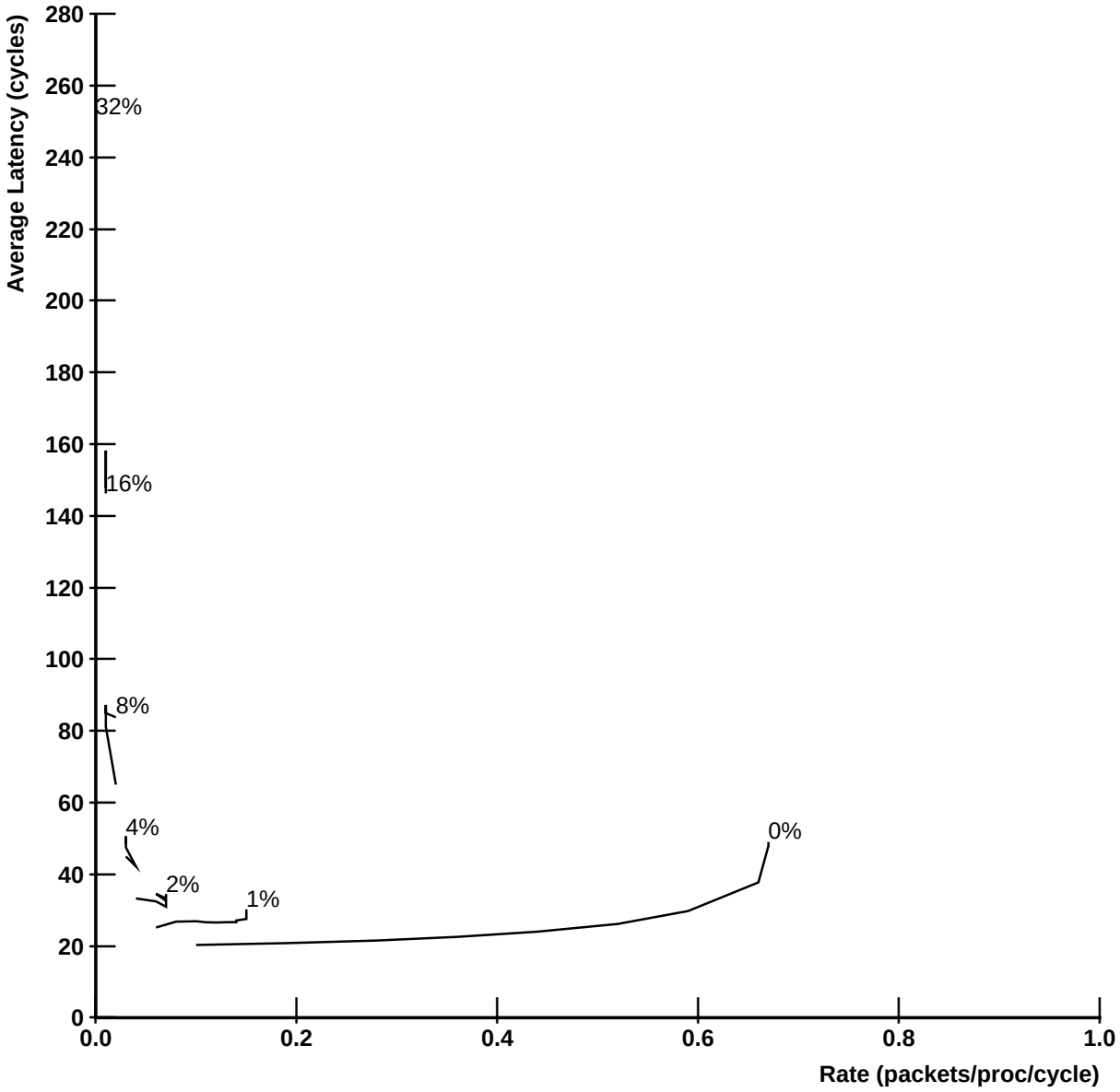

Fig 13(a): All Requests

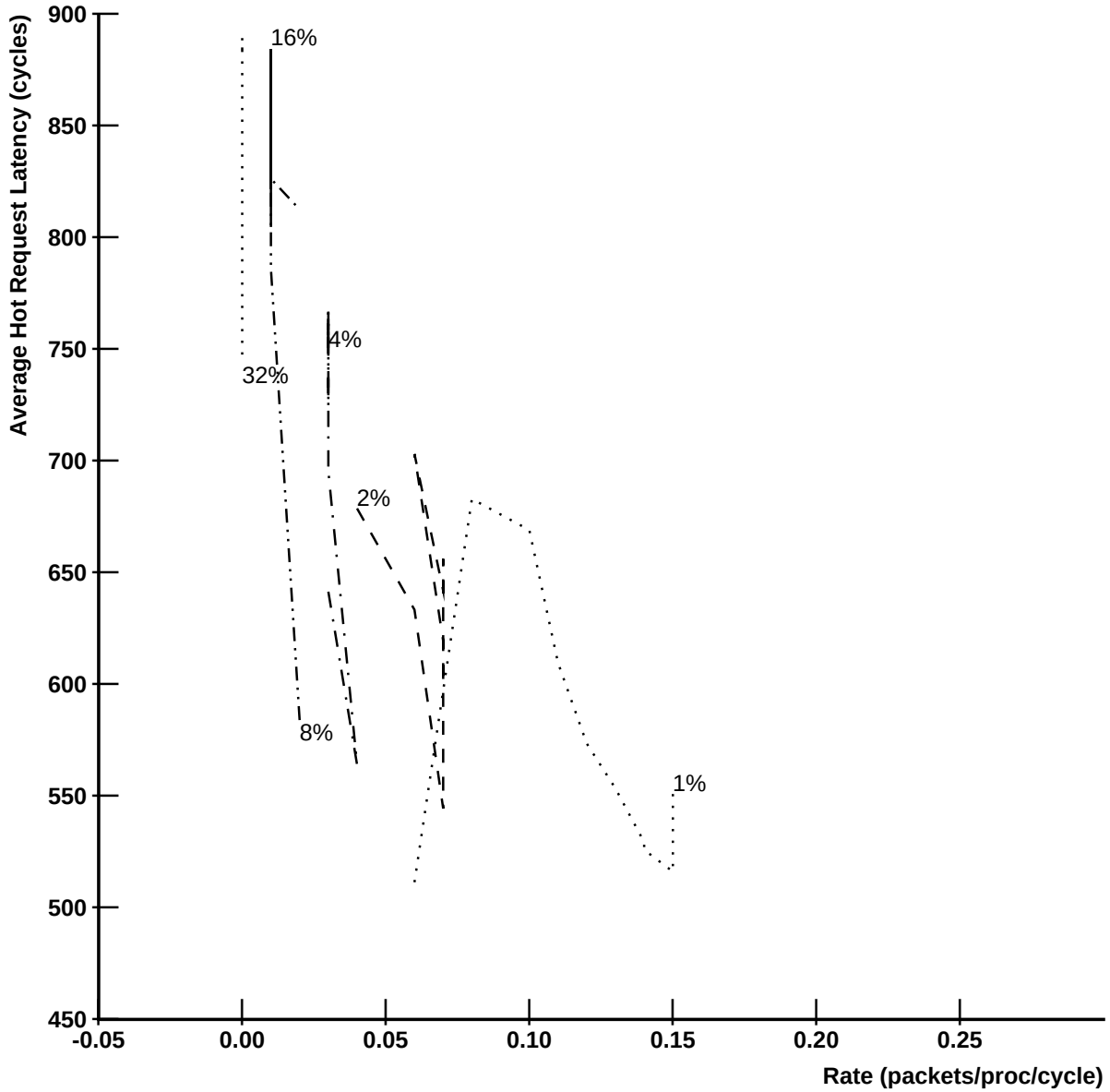

Fig 13(b): Hot Requests

Fig 13: Average Latency when Combining Sets are Appended

Figure 13(a) shows the average latency vs maximum bandwidth when the subsequent combining sets are appended to the existing set. As shown in figure 13(b), the average delay of hot requests is quite large, hence the average latency of all requests increases.

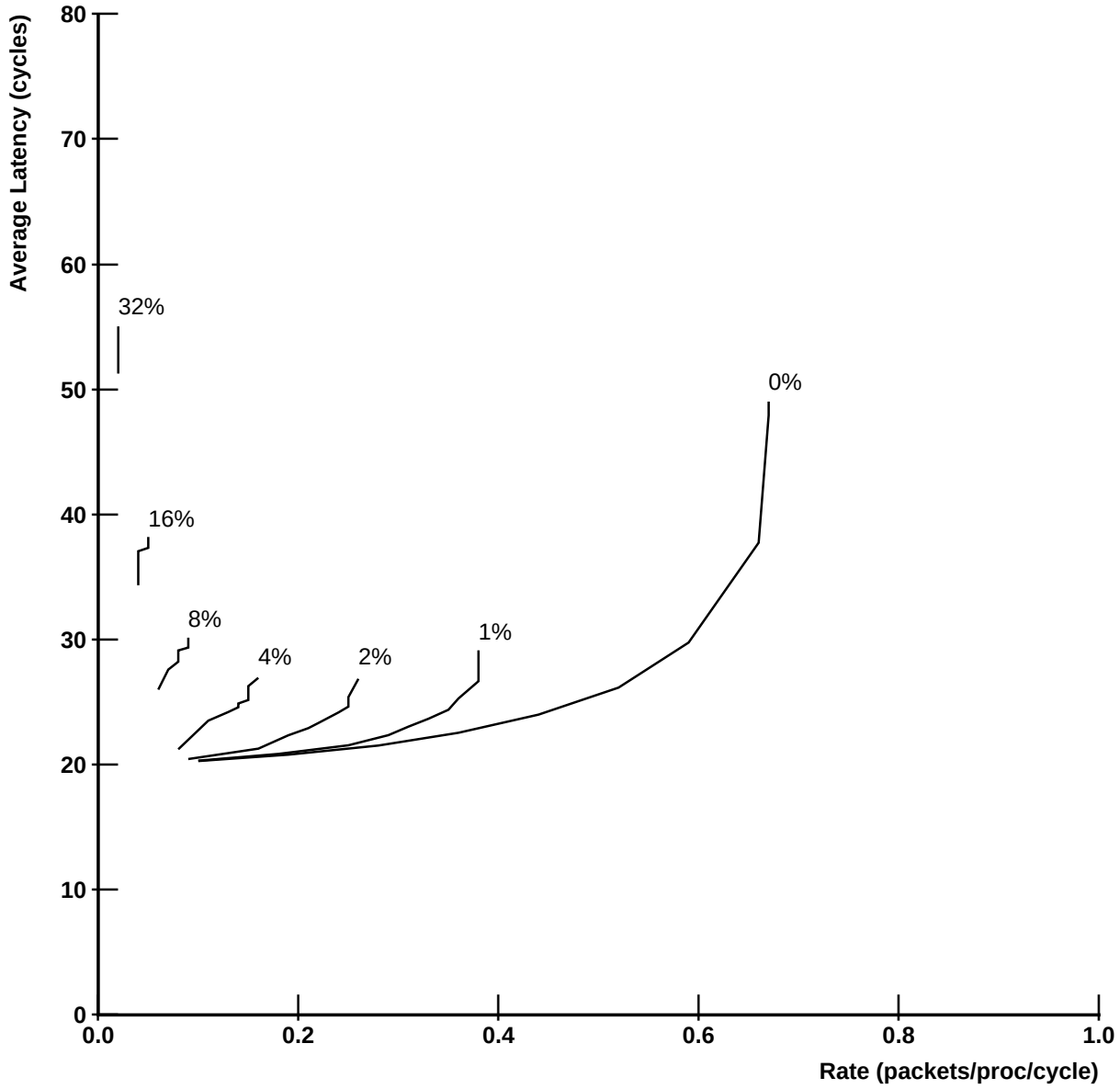

Fig. 14(a): All Requests

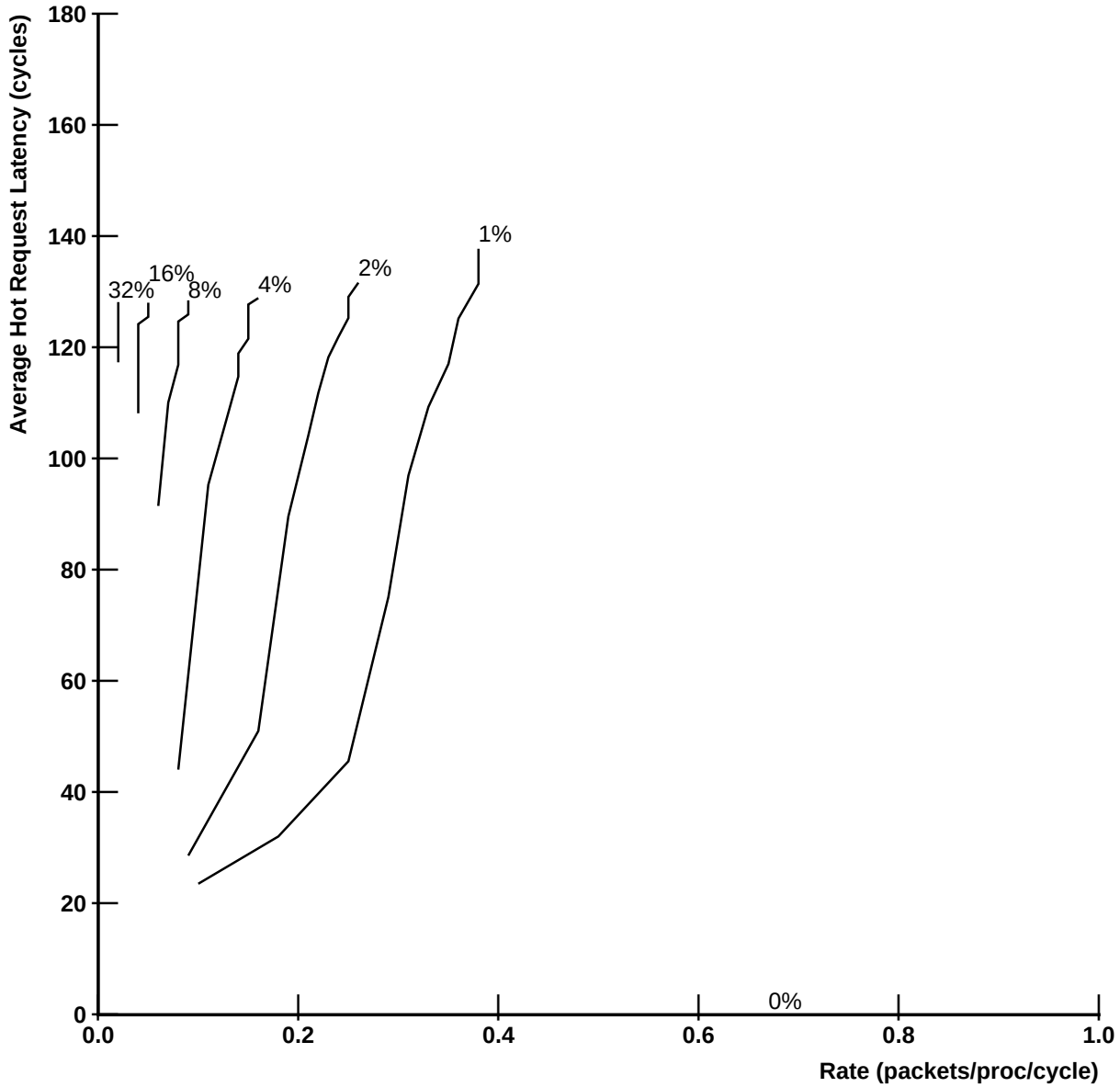

Fig. 14(b): Hot Requests

Fig. 14: Average Latency with ALU's in Interconnect Nodes

Figure 14(a) shows the average latency vs maximum bandwidth when the nodes of the network have ALU's. The ALU's eliminate the need to lock memory for result distribution, greatly reducing the average latency of hot requests, as shown in figure 14(b).