NetCache: A Network/Cache Hybrid for Multiprocessors *

Enrique V. Carrera and Ricardo Bianchini

COPPE Systems Engineering
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil 21945-970

{vinicio,ricardo }@cos.ufrj.b r

Technical Report ES-455/97, November 1997, COPPE/UFRJ

Abstract

In this paper we propose the use of an optical network not only as the communication medium, but also as a system-wide cache for the shared data in a multiprocessor. More specifically, the basic idea of our novel network/cache hybrid (and associated coherence protocol), called NetCache, is to use an optical ring network on which some amount of recently-accessed shared data is continually sent around. These data are organized as a cache shared by all processors. We use detailed execution-driven simulations of a dozen applications to evaluate a multiprocessor based on our NetCache architecture. We compare a 16-node multiprocessor with a third-level NetCache against three highly-efficient systems based on the DMON and LambdaNet optical interconnects. Our results demonstrate that the NetCache multiprocessor outperforms the DMON systems consistently for our applications; running time differences can be as significant as 105%. The NetCache system also compares favorably against the LambdaNet multiprocessor. For nine of our applications, the running time advantage of the NetCache machine ranges from 7% for applications with little data reuse in the shared cache to 79% for applications with significant data reuse. For the other applications, the two systems perform similarly. Based on these results and on our parameter space study, our main conclusion is that the NetCache is highly efficient under most architectural assumptions and for most applications.

1 Introduction

The gap between processor and memory speeds continues to widen; memory accesses usually take hundreds of processor cycles to complete, especially in scalable shared-memory multiprocessors. Caches are generally considered the best technique for tolerating the high latency of memory accesses. Microprocessor chips include caches so as to avoid processor stalls due to the unavailability of data or instructions. Modern computer systems use off-chip caches to reduce the average cost of memory accesses. Unfortunately, caches are often somewhat smaller than their associated processors' working sets, meaning that a potentially large percentage of memory references might still end up reaching the memory modules. This scenario is particularly damaging to performance when memory references are directed to remote memory modules through the scalable interconnection network. In this case, the performance of the network is critical.

The vast majority of multiprocessors use electronic interconnection networks. However, relatively recent advances in optical technology have prompted several studies on the use of optical networks in multiprocessors. Optical fibers exhibit extremely high bandwidth and can be multiplexed to provide a large number of

*This research was supported by Brazilian CNPq.

independent communication channels. These characteristics can be exploited to improve performance (significantly) by simply replacing a traditional multiprocessor network with an optical equivalent. However, we believe that optical networks can be exploited much more effectively. The extremely high bandwidth of optical media provides data storage in the network itself and, thus, these networks can be transformed into fast-access data storage devices.

Based on this observation and on the need to reduce the average latency of memory accesses in parallel computers, in this paper we propose the use of an optical network not only as the communication medium, but also as a system-wide cache for the shared data in a multiprocessor. More specifically, the basic idea of our novel network/cache hybrid (and associated coherence protocol), called NetCache, is to use an optical ring network on which some amount of recently-accessed shared data is continually sent around. These data are organized as a cache shared by all processors. Most aspects of traditional caches, such as hit/miss rates, capacity, storage units, and associativity, also apply to the NetCache.

The functionality provided by the NetCache suggests that a multiprocessor based on it should deliver better performance than previously-proposed optical network-based multiprocessors. To verify this hypothesis, we use detailed execution-driven simulations of a dozen applications running on a 16-node multiprocessor with a third-level shared cache, and compare it against three highly-efficient systems based on the DMON [15] and LambdaNet [13] optical interconnects. Our results demonstrate that the NetCache multiprocessor outperforms the DMON systems consistently for our applications; the performance differences can be as significant as 105%. The NetCache system also compares favorably against the LambdaNet multiprocessor. For nine of our applications, the running time advantage of the NetCache machine ranges from 7% for applications with little data reuse in the shared cache to 79% for applications with significant data reuse. For the other applications, the two systems perform similarly.

Our parameter space study evaluates the impact of each of our architectural assumptions, including the transmission rate and the memory block read latency. The results of this study show that the NetCache provides consistent performance improvements under most architectural assumptions. In fact, the widening gap between processor and memory speeds indicates that the potential benefit of the NetCache architecture will continually rise in the future.

Based on our large set of performance results, our main conclusion is that the NetCache architecture is highly efficient in most scenarios. Given that optical technology will likely provide a better cost/performance ratio than electronics in the future, we believe that designers should definitely consider the NetCache architecture as a practical approach to both low-latency communication and memory latency tolerance.

The remainder of this paper is organized as follows. The next section presents some background material on optical networks and Wavelength Division Multiplexing, and describes the DMON and LambdaNet interconnects. Section 3 describes the architecture of our network/cache hybrid in detail. Section 4 presents our experimental methodology and application workload. Section 5 presents the results of our base experiments, studies the effectiveness of the shared cache in the NetCache architecture, evaluates different shared cache organizations and policies, and presents our parameter space study. Section 6 discusses the related work. Finally, section 7 summarizes our findings and concludes the paper.

2 Background

In this section we discuss the background behind our work. We focus on issues related to optical communication systems and on the two systems we compare performance against, DMON and LambdaNet-based multiprocessors. Note that the DMON and LambdaNet networks were chosen for comparison for different reasons: DMON is one of only a few networks proposed specifically for multiprocessors and, when coupled with the I-SPEED coherence protocol, has been shown to outperform multiprocessors based solely on snooping and solely on directories; LambdaNet trades off complexity for performance and thus can be turned

into a performance upper bound for multiprocessors that do not cache data on the network, if combined with efficient coherence protocols.

2.1 Optical Communication Systems

Two main topics in optical communication systems bear a direct relationship to the ideas proposed in this paper: Wavelength Division Multiplexing (WDM) networks and delay line memories. In the following we discuss each of these topics in turn.

WDM networks. Through careful fabrication of optical fibers, transmitters, and receivers it is nowadays possible to build dispersion-free optical communication systems with low attenuation and high bandwidth. The maximum bandwidth achievable over an optic fiber is on the order of Tbits/s [14]. However, due to the fact that the hardware associated with the end points of an optical communication system is usually of an electronic nature, transmission rates are currently limited to the Gbits/s level. In order to approach the full potential of optical communication systems, multiplexing techniques must be utilized.

WDM is one such multiplexing technique. With WDM, several independent communication channels can be implemented on the same fiber. Optical networks that use this multiplexing technique are called WDM networks. The simplest way of implementing a WDM network is through a passive star coupler and a set of receivers and transmitters [3]. The star coupler broadcasts every WDM channel to the processing nodes connected to the network. Nodes usually do not "listen" to all channels however, as the number of optical devices ultimately determines the cost of the network. In most cases a channel has a transmitter node and a receiver node, but often "broadcast" channels are implemented, on which any node can transmit to the other nodes.

The advent of tunable transmitters and receivers has contributed to a significant reduction of the number of optical devices in WDM networks, but has introduced a new set of issues involved in channel access control and communication. Different networks handle these issues in different ways, e.g. [7, 16].

Due to the rapid development of the technology used in its implementation, WDM has become one of the most popular multiplexing techniques. WDM multiplexors and demultiplexors can now be found commercially with hundreds of channels.

Delay line memories. Given that light travels at a constant and finite propagation speed on the optical fiber (approximately 2.1E+8 meters/s), a fixed amount of time elapses between when a datum enters and leaves an optical fiber. In effect, the fiber acts as a delay line. Connecting the ends of the fiber to each other (and regenerating the signal periodically), the fiber becomes a delay line memory [25], since the data sent to the fiber will remain there until overwritten. An optical delay line memory exhibits characteristics that are hard to achieve by other types of delay lines. For instance, due to the high bandwidth of optical fibers, it is possible to store a reasonable amount of memory in just a few meters of fiber (e.g. at 10 Gbits/s, about 5 Kbits can be stored on one 100 meters-long WDM channel).

2.2 DMON

The Decoupled Multichannel Optical Network (DMON) is an interesting WDM network that has been proposed by Ha and Pinkston in [15]. The network divides its $p + 2$ (where p is the number of nodes in the system) channels into two groups: one group is used for broadcasting, while the other is used for point-to-point communication between nodes. The first group is formed by two channels shared by all nodes in the system, the control channel and the broadcast channel. The other p channels, called home channels, belong in the second group of channels.

The control channel is used for distributed arbitration of all other channels through a reservation scheme [7]. A node that wants to transmit on one of the channels must first wait for its turn to access the control channel and then broadcast this desire on it. This broadcast makes other nodes aware of the communication

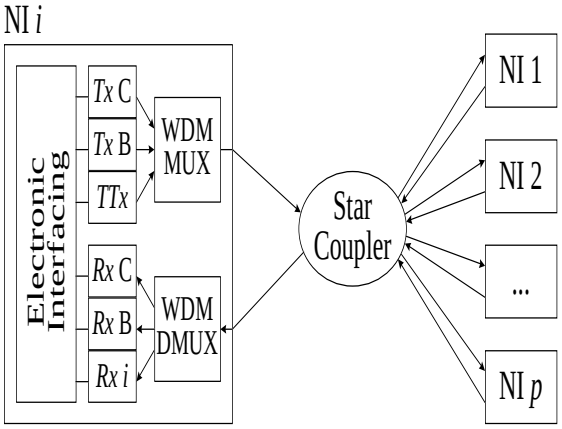

Figure 1: Overview of DMON.

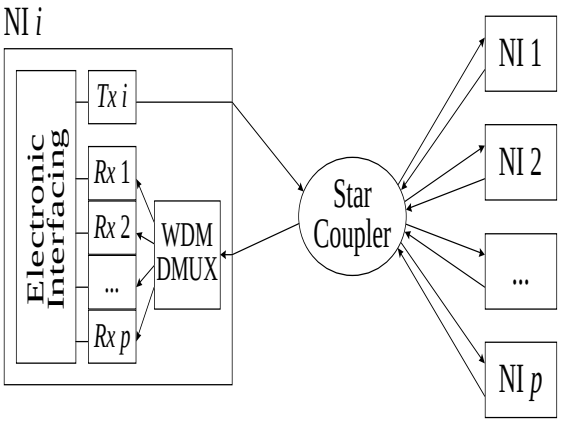

Figure 2: Overview of LambdaNet.

about to take place, thereby avoiding any conflicts. The control channel itself is multiplexed using the TDMA (Time Division Multiple Access) protocol [7].

The broadcast channel is used for broadcasting global events, such as coherence and synchronization operations, while home channels are used only for memory block request and reply operations. Each node can transmit on any home channel, but can only receive from a single home channel. Each node acts as "home" (the node responsible for providing up-to-date copies of the blocks) for $1/p$ of the cache blocks. A node receives requests for its home blocks from its home channel. Block replies are sent on the requester's home channel.

Note that the write and read transactions follow different network paths in DMON. This decoupling of channel resources based on reference type is one of the distinctive features of DMON. Even though decoupling has the potential benefit of reducing the average memory access latency, it can cause a critical race when a coherence operation and memory read corresponding to the same block overlap in time.

Figure 1 overviews a network interface ("NI") in the DMON architecture, with its transmitters (labeled "*Tx*"), receivers ("*Rx*"), and tunable transmitters ("*TTx*"). As seen in the figure, in this architecture each node requires two fixed transmitters[1] (one for each broadcast channel), a tunable transmitter (for the home channels), and three fixed receivers (two for the broadcast channels and one for the node's home channel). The overall hardware cost of the DMON architecture in terms of optical components is then $6 \times p$.

The Snoopy Protocol Enhanced and Extended with Directory (SPEED) is a high-performance cache coherence protocol created to exploit the communication features of DMON. In its invalidate version (I-SPEED), the only version described in [15], the protocol defines four cache and memory block states: clean, exclusive, shared, and invalid. The protocol allows only one copy of the block to be in exclusive or shared state. A node that caches a block in one of these states is the owner of the block. A cache-forwarded copy of an exclusive or shared block is received as clean by the requester. The home node of each memory block includes a directory entry that stores the current owner of the block. All misses to a memory block are sent to its home node and, if necessary, forwarded to the owner node.

I-SPEED also defines states that handle critical races. A critical race is detected when a coherence operation is seen for a block that has a pending read. I-SPEED treats the race by forcing the invalidation of the would-be-incoherent copy of the block right after the pending read is completed. Further details about I-SPEED can be found in [15].

We also compare performance against an update-based protocol we previously proposed for DMON

[1] These fixed transmitters are not part of the original DMON proposal. We add them here to avoid incurring the overhead of re-tuning the tunable transmitter all the time.

[4]. The protocol is very simple since all writes to shared data are sent to their corresponding home nodes, through coalescing write buffers. Thus, a cache miss can be satisfied immediately by the home node, obviating the need for any directory information. Our update protocol also includes support for handling critical races. Like in I-SPEED, a critical race is detected when a coherence operation is seen for a block that has a pending read. Our protocol treats the race by buffering the updates received during the pending read operation and applies them to the block right after the read is completed.

Given that a single broadcast channel would not be able to deal gracefully with the heavy update traffic involved in a large set of applications, we extended DMON with an extra broadcast channel for transferring updates. A node can transmit on only one of the coherence channels, which is determined as a function of the node's identification, but can receive from both of these channels. Besides this extra channel (and associated receivers), the hardware of the modified DMON network is the same as presented in figure 1. Thus, the overall hardware cost of this modified DMON architecture in terms of optical components is then $7 \times p$.

2.3 LambdaNet

The LambdaNet architecture has been proposed by Goodman *et al.* in [13]. The network allocates a WDM channel for each node; the node transmits on this channel and all other nodes have fixed receivers on it. In this organization each node uses one fixed transmitter and p fixed receivers, as shown in figure 2. The overall hardware cost of the LambdaNet is then $p^2 + p$.

No arbitration is necessary for accessing transmission channels. Each node simultaneously receives all the traffic of the entire network, with a subsequent selection, by electronic circuits, of the traffic destined for the node. This scheme thus allows channels to be used for either point-to-point or broadcast communication.

Differently from DMON, the LambdaNet was not proposed with an associated coherence protocol. The LambdaNet-based multiprocessor we study in this paper uses a write-update cache coherence protocol, where write and synchronization transactions are broadcast to nodes, while the read traffic uses point-to-point communication between requesters and home nodes. Just as the update-based protocol we proposed for DMON, the memory modules are kept up-to-date at all time, so that home nodes can reply to block requests resulting from cache misses immediately. Again, in order to reduce the write traffic to home nodes, we assume coalescing write buffers.

Note that the LambdaNet architecture is impractical due to its hardware cost. Our only reason for including this scheme in our study is to use it as a basis for comparison against the other schemes. The combination of the LambdaNet and the coherence protocol we suggest for it represents a performance upper bound for multiprocessors that do not cache data on the network, since the update-based protocol avoids coherence-related misses, the LambdaNet channels do not require any medium access protocol, and the LambdaNet hardware does not require the tunning of transmitters or receivers.

3 The Network/Cache Hybrid

In this section we describe the NetCache architecture. We start by overviewing the basic architecture of our network and then move on to describing its coherence protocol and memory access operations in detail. To end the section, we discuss an alternative network implementation that achieves the same objectives as the NetCache.

3.1 Overview

As seen in figure 3, each node in a NetCache-based multiprocessor is extremely simple. In fact, all of the node's hardware components are pretty conventional, except for the network (NetCache) interface. More

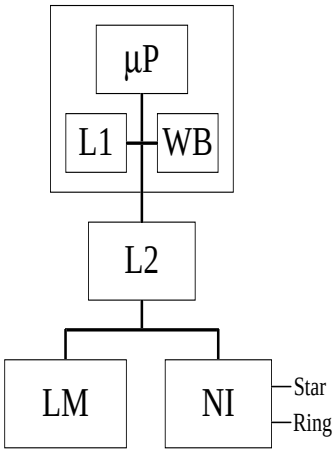

Figure 3: Overview of Node Architecture.

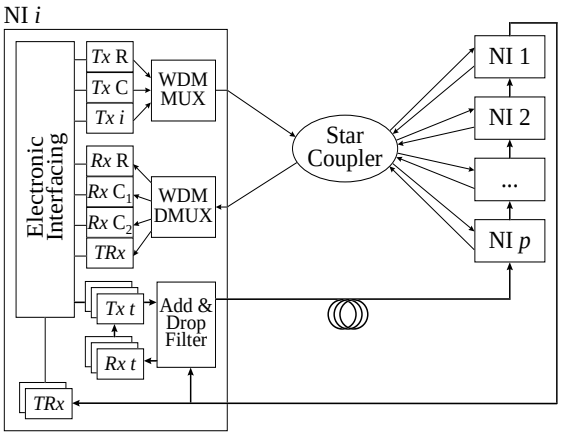

Figure 4: Overview of NetCache Architecture.

specifically, the node includes one processor (labeled "μP"), a coalescing write buffer ("WB"), first ("L1") and second-level ("L2") caches, local memory ("LM"), and the NetCache interface ("NI") that connects the node to two subnetworks: a star coupler WDM subnetwork and a WDM ring subnetwork.

Figure 4 overviews the NetCache architecture, including its interfaces and subnetworks. The star coupler subnetwork can be seen on the top half of the figure, while the ring network is mostly concentrated on the bottom half. The star coupler subnetwork is conceptually similar to the DMON network, while the ring subnetwork is unlike any other previously-proposed optical network. The following sections describe each of these components in detail.

3.2 Star Coupler (Sub)Network

Just as in DMON, the star coupler WDM subnetwork in the NetCache divides the channels into two groups: one group for broadcast-type traffic and another one for direct point-to-point communication. Three channels, a request channel and two coherence channels, are assigned to the first group, while the other p channels, called home channels, are assigned to the second group.

The request channel is used for requesting memory blocks. The response to such a request is sent by the block's home node (the node responsible for providing up-to-date copies of the block) on its corresponding home channel. The coherence channels are used for broadcasting coherence and synchronization transactions. Just like the control channel in DMON, the request channel uses TDMA for medium access control. The access to the coherence channels, on the other hand, is controlled with TDMA with variable time slots [7]. Differently from DMON, home channels do not require arbitration, since only the home node can transmit on the node's home channel.

In summary, each node can transmit on the request channel, one of the coherence channels (determined as a function of node identification), and its home channel, but can receive from any of the broadcast or home channels. Hence, as seen in figure 4, each node in the star coupler subnetwork requires three fixed transmitters (one for the request channel, one for the home channel, and the last for one of the coherence channels), three fixed receivers (for the broadcast channels), and one tunnable receiver labeled "*TR*" (for the home channels). The hardware cost for this subnetwork is then $7 \times p$ optical components.

Note that our novel star coupler (sub)network can be used by itself, as the only network of the multiprocessor, if caching of shared data is not required/desired. A full-blown comparison of such a scheme and similar networks such as DMON can be found in [4].

3.3 Ring (Sub)Network

The ring subnetwork is the most interesting aspect of the NetCache architecture. The ring is used to store some amount of recently-accessed shared data. These data are organized as a cache shared by all nodes of the machine. In essence, the ring becomes an extra level of caching that can store data from any node. However, the ring does not respect the inclusion property of cache hierarchies, i.e. the data stored by the ring is not necessarily a superset of the data stored in the upper cache levels. In other words, the storage capacity of the ring is completely independent of the individual or combined second-level cache sizes. The storage capacity of the ring is simply proportional to the number of available channels, and the channels' bandwidth and length. More specifically, the storage capacity is given by: $capacity\_in\_bits = (num\_channels \times fiber\_length \times transmission\_rate) \div speed\_of\_light\_on\_fiber$, where $speed\_of\_light\_on\_fiber = 2 \times 10^8 m/s$.

The ring works as follows. Data are continually sent around the ring's WDM channels, referred to as cache channels, in one direction. Each cache channel stores the cached blocks of a particular home node. Assuming a total of c cache channels, each home node has $t = c/p$ cache channels for its blocks. Since cache channels and blocks are associated with home nodes in interleaved, round-robin fashion, the cache channel assigned to a certain block can be determined by the following expression: <block_address> $modulo\ c$. A particular block can be placed anywhere within a cache channel.

Just as in a regular cache, our shared cache has an address tag on each block frame (shared cache line). The tags contain part of the shared cache blocks' addresses and consume a (usually extremely small) fraction of the storage capacity of the ring. The access to each block frame by different processors is strictly sequential as determined by the flow of data on the cache channel. The network interface accesses a block via a shift register that gets replicas of the bits flowing around the ring. Each shift register is as wide as a whole block frame. When a shift register is completely filled with a block, the hardware moves the block to another register, the access register, which can then be manipulated. In our base simulations, it takes 10 cycles to fill a shift register. This is the amount of time the network interface has to check the block's tag and possibly to move the block's data before the access register is overwritten.

The NetCache interface at each node can read any of the cache channels, but can only write to the t cache channels associated to the node. In addition, the NetCache interface regenerates, reshapes, and reclocks these t cache channels. To accomplish this functionality, the interface requires two tunable receivers, and t fixed transmitter and receiver sets, as shown in figure 4. One of the tunable receivers is for the cache channel used last and the other is for pre-tuning to the next channel[2]. The t fixed transmitters are used to insert new data on any of the cache channels associated with the node. In conjunction with these transmitters, the t fixed receivers are used to re-circulate the data on the cache channels associated with the node. Thus, the hardware cost for this subnetwork is then $2 \times p + 2 \times c$ optical components.

Taking both subnetworks into account, the overall hardware cost of the NetCache architecture is $9 \times p + 2 \times c$. In our experiments we set $c = 8 \times p$, leading to a total of $25 \times p$ optical components. This hardware complexity is a factor of 4 greater than that of DMON, but is linear on p, as opposed to quadratic on p as the LambdaNet's.

3.4 Coherence Protocol

The coherence protocol is an integral part of the NetCache architecture. The NetCache protocol is based on update coherence, supported by both broadcasting and point-to-point communication. The update traffic flows through the coherence channels, while the data blocks are sent across the home and cache channels. The request channel carries all the memory read requests and update acknowledgements. The description

[2]The goal here is to hide the overhead of tuning by predicting the channel that will be required next and tuning one of the receivers to it in advance.

that follows describes the coherence protocol in greater detail in terms of the actions taken on read and write accesses.

Reads. On a read access, the memory hierarchy is traversed from top to bottom, so that the required word can be found as quickly as possible. Thus, the contents of the first and second-level caches are checked, just like in any other computer system with multiple caches. A miss in the second-level cache blocks the processor and is handled differently depending on the type of data read. In case the requested block is private or maps to the local memory, the read access is treated by the local memory, which returns the block directly to the processor.

If the block is shared and maps to another home node, the miss causes an access to the NetCache. The request is sent to the corresponding node through the request channel, and the tunable receiver in the star coupler subnetwork is tuned to the correct home channel. In parallel with this, the requesting node tunes one of its tunable receivers in the ring subnetwork to the corresponding cache channel. The requesting node then waits for the block to be received from either the home channel or the cache channel, reads the block, and returns it to the second-level cache.

When a request arrives at the home node, the home checks if the block is already in any of its cache channels. In order to manage this information, the NetCache interface keeps a hash table storing the address of its blocks that are currently cached on the ring. If the block is already cached, the home node will simply disregard the request; the block will eventually be received by the requesting node. If the block is not currently cached, the home node will read it from memory and place it on the correct cache channel, replacing one of the blocks already there if necessary. (Replacements are random and do not require write backs, since the memory and the cache are always kept up-to-date.) In addition to returning the requested block through a cache channel, the NetCache interface also sends the block through the home node's home channel.

It is important to note that our protocol starts read transactions on both the star coupler and ring subnetworks so that a read miss in the shared cache takes no longer than a direct access to remote memory. If reads were only started on the ring subnetwork, shared cache misses would take half a roundtrip longer (on average) to satisfy than a direct remote memory access.

Writes. Our multiprocessor architecture implements the release consistency memory model [10]. Consecutive writes to the same cache block are coalesced in the write buffer. Coalesced writes to a private block are sent directly to the local memory through the first and second-level caches. Coalesced writes to a shared block are always sent to one of the coherence channels in the form of an update, again through the local caches. An update only carries the words that were actually modified in each block.

Each update must be acknowledged by the corresponding block's home node before another update by the same node can be issued, just so the memory modules do not require excessively long input queues (i.e. update acks are used simply as a flow control measure). The other nodes that cache the block simply update their local caches accordingly upon receiving the update. When the home node sees the update, the home inserts it into the memory's FIFO queue, and sends an ack through the request channel. The ack might not be sent immediately however, if the memory queue is filled beyond a hysteresis point. In that case, the home node delays the transfer of the ack until it can safely allow the updating node to issue another update. A node can only acquire a lock or pass a barrier point after having emptied its memory FIFO queue. Note that update acks usually do not overload the request channel, since an ack is a short message (much shorter than multi-word updates) that fits into a single request channel slot.

After having inserted the update in its memory queue, the home node checks whether the updated block is present in a cache channel. If so, besides updating its memory and local caches, the home node updates the cache channel. If the block is not present in a cache channel, the home node will not include it.

There are two types of critical races in our coherence protocol that must be taken care of. The first occurs when a coherence operation is seen for a block that has a pending read. Similarly to the update protocol we proposed for the DMON network, our coherence protocol treats this type of race by buffering updates

and later combining them with the block received from memory. The second type of race occurs because there is a window of time between when an update is broadcast and when the copy of the block in the shared cache is actually updated. During this timing window, a node might miss on the block, read it from the shared cache and, thus, never see the update. To avoid this problem, each network interface includes a small FIFO queue that buffers the block addresses of some previous updates. Any shared cache access to a block represented in the queue, must be delayed until the address of the block leaves the queue, which guarantees that when the read is issued, the copy of the block in the shared cache will be up-to-date. The management of the queue is pretty simple. The entry in front of the queue can be dropped when it has resided in the queue for the same time as two roundtrips around the ring subnetwork, i.e. the maximum amount of time it could take a home node to update the copy of the block in the shared cache. Hence, the maximum size of the queue is the maximum number of updates that can be issued during this period; in our simulations, this number is 54.

3.5 An Alternative Implementation

The main goal of the NetCache architecture, namely to keep data closer to processors than remote memory, could have been achieved with a different network implementation. More specifically, instead of storing data on the optical fiber itself by using the ring subnetwork, we could store data on electronic memory and continually broadcast them through additional channels on the star coupler subnetwork. In fact, we could exchange the ring's c cache channels for c extra channels on the star coupler subnetwork, transferring the same data as before on each channel. This modified architecture would perform exactly the same as the NetCache, as it would retain the NetCache's storage capacity, logical organization, coherence protocol, and 2nd-level read miss latencies. In addition, the new architecture would save on optical hardware, since c fixed receivers could be eliminated, but would require enough additional electronic memory to store all the data to be broadcast.

Given the similarities between these two architectural alternatives, we consider this paper to be evaluating them both at the same time. The choice of actual implementation is simply one of hardware (electronic vs. optical) cost at the time the multiprocessor is to be designed. In this paper we opted for our NetCache implementation for two reasons: a) the decreasing cost of optical components suggests that in the future optics might provide a better *cost/performance* ratio than electronics; and, more importantly for the moment, b) we intend to extrapolate the use of the NetCache to disk block caching, which would require an excessive amount of electronic memory in the alternative implementation. Our NetCache architecture can be applied to disk caching with only a *marginal* cost increase: the cost of a longer optical fiber.

4 Methodology and Workload

We are interested in evaluating the performance of our proposed NetCache multiprocessor and comparing it against previously-reported proposals for other optical network-based multiprocessors. Hence, we use simulation of real applications for our studies.

4.1 Multiprocessor Simulation

We simulate multiprocessors based on the NetCache, DMON and LambdaNet interconnects. We use a detailed execution-driven simulator (based on the MINT front-end [27]) of 16-node multiprocessors. Each node of the simulated machines contains a single 200-MHz processor, a 16-entry write buffer, a 4-KByte direct-mapped first-level data cache with 32-byte cache blocks, a 16-KByte direct-mapped second-level data cache with 64-byte cache blocks, local memory, and a network interface. (Note that the cache sizes

Operation	Latency
Shared cache read hit	
1. 1st-level tag check	1
2. 2nd-level tag check	4
3. Avg. shared cache delay	25
4. NI to 2nd-level cache	16
Total shared cache hit	46
Shared cache read miss	
1. 1st-level tag check	1
2. 2nd-level tag check	4
3. Avg. TDMA delay	8
4. Memory request	1^*
5. Flight	1
6. Memory read	76^+
7. Block transfer	11
8. Flight	1
9. NI to 2nd-level cache	16
Total shared cache miss	119

Table 1: Read Latency Breakdowns for NetCache System.

Operation	Latency	
	LambdaNet	DMON
2nd-level read miss		
1. 1st-level tag check	1	1
2. 2nd-level tag check	4	4
3. Avg. TDMA delay	–	8
4. Reservation	–	1^*
5. Tuning delay	–	4
6. Memory request	1^*	2
7. Flight	1	1
8. Memory read	76^+	76^+
9. Avg. TDMA delay	–	8
10. Reservation	–	1^*
11. Block transfer	11^*	12
12. Flight	1	1
13. NI to 2nd-level	16	16
Total 2nd-level miss	111	135

Table 2: Read Latency Breakdowns for DMON and LambdaNet Systems.

we simulate were purposedly kept small, because simulation time limitations prevent us from using real life input sizes.)

Shared data are interleaved across the memories at the block level. All instructions and first-level cache read hits are assumed to take 1 processor cycle (pcycle). First-level read misses stall the processor until the read request is satisfied. A second-level read hit takes 12 pcycles to complete. Writes go into the write buffer and take 1 pcycle, unless the write buffer is full, in which case the processor stalls until an entry becomes free. Reads are allowed to bypass writes that are queued in the write buffers. A memory module can provide the first two words requested 12 pcycles after the request is issued. Other words are delivered at a rate of 2 words per 8 pcycles. Memory and network contention are fully modeled.

In the update-based coherence protocols we simulate only the secondary cache is updated when an update arrives at a node; the copy of the block in the first-level cache is invalidated. In addition, in order to reduce the write traffic, our multiprocessors use coalescing write buffers for all protocol implementations. A coalesced update only carries the words that were actually modified in each block. Our protocol implementations assume a release-consistent memory model [10].

The optical transmission rate we simulate is 10 Gbits/s. In the NetCache simulations we assume 128 cache channels. The length of the fiber is approximately 45 meters. These parameters lead to a ring roundtrip latency of 40 cycles and a storage capacity of 32 KBytes of data. The shared cache line size is 64 bytes.

The coherence protocol implemented on top of the shared cache has been described in the previous section. The latency of shared cache reads in the NetCache architecture is broken down into its base components in table 1. The latency of 2nd-level read misses in the LambdaNet and DMON-based systems is broken down in table 2. The coherence transaction latencies of the NetCache, LambdaNet, DMON with update-based coherence (DMON-U), and DMON with I-SPEED (DMON-I) systems are listed in table 3 and assume 8 words written in the cache block[3]. All numbers in the tables are in pcycles and assume chan-

[3] Note that most or all of the coherence transaction overhead is usually hidden from the processor by the write buffer.

Operation	Latency (in processor cycles)			
	OPTNET	LambdaNet	DMON-U	DMON-I
1. 2nd-level tag check	4	4	4	4
2. Write to NI	10	10	10	2
3. Avg. TDMA delay	8*	–	8	8
4. Reservation	–	–	1*	1*
5. Update/Invalidate	8	7	8	2
6. Flight	1	1	1	1
7. Avg. TDMA delay	8	–	8	8
8. Reservation	–	–	1*	1*
9. Ack	1*	1*	1	1
10. Flight	1	1	1	1
11. Write	–	–	–	8
Total coherence transaction	41	24	43	37

Table 3: Latency (in 5-ns pcycles) of Coherence Transactions for NetCache, LambdaNet, DMON-U, and DMON-I. Assuming 8 words written in block.

Program	Description	Input Size
CG	Conjugate Gradient kernel	1400 $\times$ 1400 doubles, 78148 non-zeros
Em3d	Electromagnetic wave propagation	8 K nodes, 5% remote, 10 iterations
FFT	1D Fast Fourier Transform	16 K points
Gauss	Unblocked Gaussian Elimination	256 $\times$ 256 floats
LU	Blocked LU factorization	512 $\times$ 512 floats
Mg	3D Poisson solver using multigrid techniques	24 $\times$ 24 $\times$ 64 floats, 6 iterations
Ocean	Large-scale ocean movement simulation	66 $\times$ 66 grid
Radix	Integer Radix sort	512 K keys, radix 1024
Raytrace	Parallel ray tracer	teapot
SOR	Successive Over-Relaxation	256 $\times$ 256 floats, 100 iterations
Water	Simulation of water molecules, spatial alloc.	512 molecules, 4 timesteps
WF	Warshall-Floyd shortest paths algorithm	384 vertices, i,j connected w/ 50% chance

Table 4: Application Description and Main Input Parameters.

nel and memory contention-free scenarios. The values marked with "*" and "+" are the ones that may be increased by network and memory contention/serialization, respectively.

Note that in our base simulations the minimum TDMA slot duration is 1 pcycle for both DMON and NetCache networks. Thus, each control channel slot in DMON and request channel slot in NetCache are 1 pcycle long. Each coherence channel slot in the NetCache is at least 2 pcycles long; the actual duration of each slot depends on the number of words updated.

The simulation parameters we assume represent our perception of what's "reasonable" for current and near future multiprocessors. The parameter space study presented in section 5 allows us to investigate the impact of our most important architectural assumptions.

4.2 Workload

Our application workload consists of twelve parallel programs: CG, Em3d, FFT, Gauss, LU, Mg, Ocean, Radix, Raytrace, SOR, Water, and WF. Table 4 lists the applications and their input parameters.

FFT, LU, Ocean, Radix, Raytrace, and Water are from the SPLASH-2 suite and have been described in detail elsewhere [28]. CG and Mg are parallel implementations of the conjugate gradient and multigrid benchmarks of the NAS suite, which is described in detail in [1]. Em3d is from UC Berkeley [5] and simulates electromagnetic wave propagation through 3D objects. Gauss, SOR, and WF have been developed locally. Gauss performs unblocked Gaussian Elimination without pivoting or back-substitution. SOR performs successive over-relaxation on a grid of elements. WF uses a parallelization of the Warshall-Floyd algorithm to compute the shortest paths between all pairs of nodes in a graph represented by an adjacency matrix.

5 Experimental Results

In this section we evaluate the performance of a NetCache-based multiprocessor, while comparing it to multiprocessors based on the LambdaNet and DMON networks. We start with speedup and execution time results and then move on to a detailed analysis of the effectiveness of data caching in the NetCache architecture. After that, we evaluate different cache organizations and replacement policies for the NetCache. Finally, we study the effect of several of our simulation assumptions.

5.1 Overall Performance

Figure 5 shows the speedup of our applications running on a 16-node NetCache-based multiprocessor with a 32-KByte shared cache. The figure demonstrates that, except for CG, LU, and WF, our applications exhibit good speedup levels on 16 nodes. The two extremes in speedup performance, Em3d and WF, deserve further explanation. Em3d achieves superlinear speedup as a result of its terrible single-node 1st and 2nd-level cache behaviors; caches are simply not effective for this application on a single node. WF achieves poor performance on 16 nodes as a result of large barrier overheads due mostly to significant load imbalance.

Figure 6 shows the running times of our applications again on a 16-node multiprocessor with a 32-KByte shared cache. For each application we show, from left to right, the NetCache, LambdaNet, DMON-U, and DMON-I performances, normalized to the NetCache results. This figure demonstrates that the DMON-U system performs as well or better than the DMON-I multiprocessor. The two systems achieve roughly the same performance for FFT, LU, Raytrace, Water, and WF, but for the other applications the advantage of DMON-U ranges from 5% (CG) to 42% (Radix), averaging 21%. This result can be explained in part by the fact that the update-based systems exhibit lower 2nd-level cache read miss rates than DMON-I. The differences in miss rates are not terribly significant however, since our applications are dominated by replacement misses. The most important factor in this comparison is that the DMON-I multiprocessor suffers

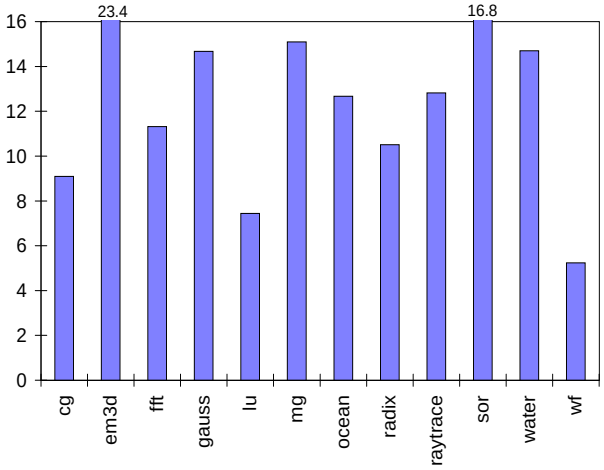

Figure 5: Speedups of 16-Node NetCache Multiprocessor.

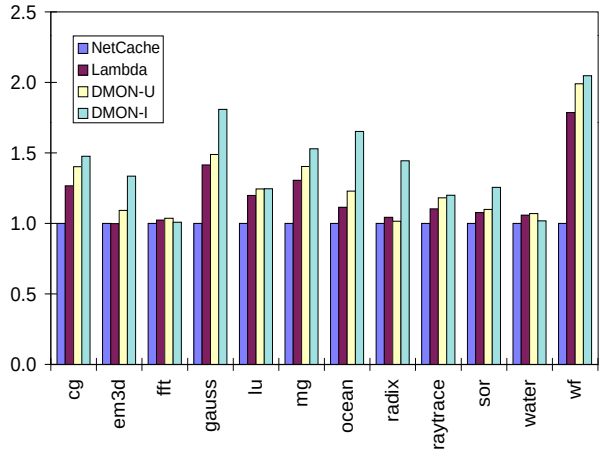

Figure 6: Run Times of (from left to right) NetCache, LambdaNet, DMON-U, and DMON-I for 16 nodes.

more significantly from memory and network contention than the other systems. Network and memory contention are more pronounced in the DMON-I system due to writebacks of dirty cache blocks that are evicted from caches, the directory lookups required in all memory requests, and the extra messages involved in forwarding requests to the current owners of blocks.

A comparison between the LambdaNet and DMON-U systems shows that the former multiprocessor performs at least as well as the latter for all applications, but performance differences are always small. The two systems achieve roughly the same performance for FFT, LU, Radix, SOR, and Water. For the other 7 applications, the performance advantage of the LambdaNet ranges from 5% (Gauss) to 11% (CG and WF), averaging 9%. This relatively small difference might seem surprising at first, given that the read latency is by far the largest overhead in all our applications (except WF) and the contention-free 2nd-level read miss latency in the DMON-U system is 22% higher than in the LambdaNet system. However, the LambdaNet multiprocessor is usually more prone to contention effects than the DMON-U and NetCache systems, due to two characteristics of the former system: a) its read and write transactions are not decoupled; and b) its absence of serialization points for updates from different nodes leads to an enormous update throughput. Contention effects are then the reason why performance differences in favor of the LambdaNet system are not more significant.

A comparison between the NetCache and DMON-I systems is clearly favorable to our system in all cases, except FFT and Water for which the two systems perform similarly. For the other 10 applications, the performance advantage of the NetCache multiprocessor ranges from 20% for Raytrace to 105% for WF, averaging 50%. The main reasons for the immense performance disparity between NetCache and DMON-I are twofold: a) a significant fraction of the 2nd-level read misses in the NetCache architecture hit in the shared cache, which reduces the latency of these operations tremendously; and b) even when the shared cache is relatively ineffective, the average 2nd-level read miss latencies in the DMON-I system are higher than in the NetCache multiprocessor, especially when the amount of network traffic generated by the application is significant.

The NetCache system also compares favorably against the DMON-U multiprocessor. Except for FFT and Radix, where the performance difference between these two systems is negligible, the advantage of the NetCache ranges from 7% for Water to 99% for WF, averaging 32%. This significant performance difference can be credited to a large extent to the ability of the shared cache to reduce the average read latency, since their contention-free 2nd-level read miss latencies only differ by 13% and contention affects the two systems in similar ways.

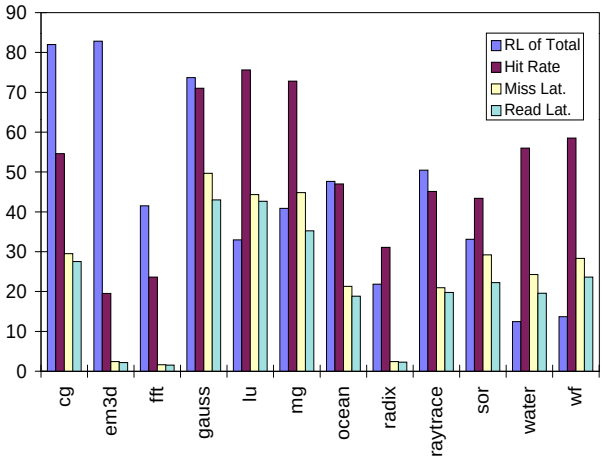

Figure 7: (From left to right) Read Latency as % of Run Time (without shared cache), 32-KByte Shared Cache Hit Rates, NetCache 2nd-Level Read Miss Latency Reductions, and NetCache Read Latency Reductions for 16 nodes.

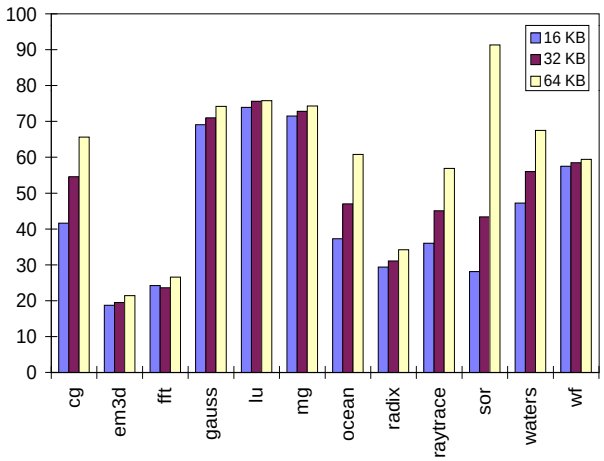

Figure 8: Hit Rates for (from left to right) 16, 32, and 64-KByte Shared Caches for 16 nodes.

Figure 6 demonstrates that a comparison between the NetCache and LambdaNet multiprocessors also clearly favors our system. The performance of the two systems is equivalent for 3 applications: Em3d, FFT, and Radix. For the other 9 applications, the performance advantage of the NetCache multiprocessor averages around 26%, ranging from about 7% for SOR and Water to 41 and 79% for Gauss and WF, respectively. Disregarding the 5 applications for which the performance difference between the two systems is smaller than 10%, the average NetCache advantage over the LambdaNet goes up to 31%. Again, the main reason for these favorable results is the ability of the NetCache architecture to reduce the average read latency, when a non-negligible percentage of the 2nd-level read misses hit in the shared cache. Our experiments with a NetCache multiprocessor without a shared cache, i.e. without the ring subnetwork of the NetCache architecture, confirm this claim. In general, the performance of this multiprocessor is a little worse (1% on average) than that of the LambdaNet system, a little better (4% on average) than that of the DMON-U system, and still far superior (17% on average) to the DMON-I system. Similar experiments assuming slightly different architectural parameters are described in [4]. In order to evaluate the shared cache in the NetCache architecture in more detail, the next subsection assesses the effectiveness of data caching in the architecture. Continuing this assessment, subsection 5.3 quantitatively evaluates alternative shared cache organizations and policies.

5.2 Effectiveness of Data Caching

Figure 7 concentrates our data on the effectiveness of data caching in the NetCache architecture, again assuming 16 processor nodes. For each application we show a group of 4 bars. The leftmost bar represents the read latency as a percentage of the total execution time for a NetCache multiprocessor without a shared cache. The other bars present results from the NetCache multiprocessor with a 32-KByte shared cache: from left to right, the shared cache hit rate, the percentage reduction of the average 2nd-level read miss latency, and the percentage reduction in the total read latency.

The figure demonstrates that for only 3 applications in our suite (Radix, Water, and WF) the read latency is a small fraction of the overall running time of the machine without a shared cache; the read latency fraction is significant for the other 9 applications. It is exactly for the applications with large read latency

fractions that a shared cache is most likely to improve performance. However, as seen in the figure, not all applications exhibit high hit rates when a 32-KByte shared cache is assumed.

In essence, the hit rate results divide our applications into 3 groups as follows. The first group (from now on referred to as *Low-reuse*) includes the applications with insignificant data reuse in the shared cache. In this group are Em3d, FFT, and Radix, for which less than 32% of the 2nd-level read misses hit in the shared cache. As a result, the reductions in 2nd-level read miss latency and total read latency are almost negligible for these applications. The second group of applications (*High-reuse*) exhibit significant data reuse in the shared cache. Three applications belong in this group: Gauss, LU, and Mg. Their hit rates are very high, around 70%, leading to reductions in 2nd-level read miss latency and total read latency of at least 35%. The other 6 applications (CG, Ocean, Raytrace, SOR, Water, and WF) form the third group (*Moderate-reuse*), with intermediate hit rates and latency reductions in the 20 to 30% range.

Overall, our caching results suggest that the NetCache multiprocessor could not significantly outperform the LambdaNet system for Em3d, FFT, Radix, and Water; these applications simply do not benefit significantly from our shared cache, either because their hit rates are low or because the latency of reads is just not a performance issue. On the other hand, the results suggest that WF should not benefit from the NetCache architecture either, which is not confirmed by our simulations. The reason for this (apparently) surprising result is that the read latency reduction promoted by the NetCache has an important side-effect: it improves the synchronization behavior of 7 applications in our suite (CG, LU, Mg, Ocean, Radix, Water, and WF). For WF in particular, a 32-KByte shared cache improves the synchronization overhead by 56% by alleviating the load imbalance exposed during this application's barrier synchronizations.

5.3 Evaluating Different Shared Cache Organizations and Replacement Policies

In this subsection we evaluate different shared cache organizations and replacement policies. In particular, we study the shared cache size, the shared cache block size, the shared cache associativity, and the shared cache replacement policy.

5.3.1 Shared Cache Size

Figure 8 presents the NetCache hit rates for 16, 32, and 64-KByte shared caches for our 16-node multiprocessor. We vary the size of the shared cache by adjusting the number of cache channels accordingly. More specifically, a 16-KByte shared cache is implemented with 64 cache channels, while a 64-KByte shared cache is implemented with 256 cache channels.

The figure shows that the hit rates of the applications in the *Low-reuse* and *High-reuse* groups are not affected by the size of the cache. In case of the *Low-reuse* applications, the reason for this result is that the shared cache is simply way too small to hold the working sets of all nodes. The insensitivity of the *High-reuse* applications to the shared cache size has a different reason: even a 16-KByte shared cache is already enough to hold most of the joint working sets, so increases in size have a small impact on hit rates. The applications in the *Moderate-reuse* group display the most interesting results. The hit rates of these applications improve quite significantly with the shared cache size increases, except in the case of WF. The reason for this exception is that the size of the shared cache is insignificant compared to the size of the WF joint working set. This application does not behave exactly like the ones in the *Low-reuse* group simply due to its excellent spatial locality properties.

Figures 9 and 10 show the read latencies and overall running times, respectively, for each shared cache size. Note that the figures also include a bar (the leftmost bar) corresponding to a NetCache multiprocessor without a shared cache. The data in the figures are normalized to the results of the multiprocessor without a shared cache. The most important observation to be made out of figure 9 is that the use of a shared cache reduces the read latency significantly for the vast majority of our applications, all applications in groups

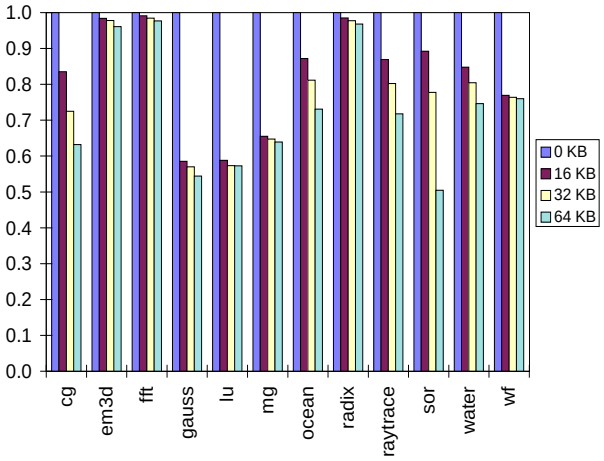

Figure 9: Read Latencies for (from left to right) No, 16, 32, and 64-KByte Shared Caches for 16 nodes.

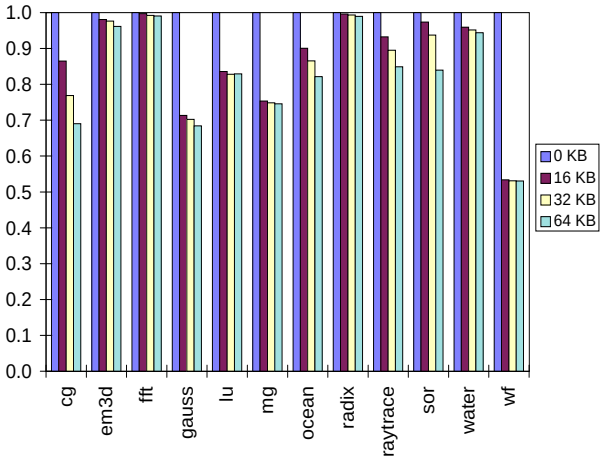

Figure 10: Run Times for (from left to right) No, 16, 32, and 64-KByte Shared Caches for 16 nodes.

Moderate-reuse and *High-reuse*. Latency reductions for these applications can be as large as 50%, as in the case of SOR with a 64-KByte shared cache, and average 28% for 32-KByte shared caches. In addition, as one would expect, the latency reductions entailed by the different shared cache sizes follow the same trends as in figure 8.

The effect of these improvements on overall performance is also noteworthy. Figure 10 demonstrates that indeed the use of a shared cache in the NetCache architecture improves running times quite significantly, except for Em3d, FFT, Radix, and Water, as aforementioned. The performance improvement achieved by WF is particularly striking, 47%. A comparison of the results for the different sizes justifies our choice of a 32-KByte shared cache for our base architecture; this size strikes a better cost/benefit ratio than its competitors.

5.3.2 Shared Cache Block Size

We also evaluated our choice of block size for the shared cache. Recall that the base shared cache block size is 64 bytes, the smallest possible given that 2nd-level cache block sizes are also 64 bytes in our simulated multiprocessors. Our experiments show that increasing the block size to take advantage of program locality turns out to be a bad decision. For the applications with poor spatial locality such an increase simply increases the amount of pollution in the shared cache; the performance penalty associated with a 128-byte block reaches 33% for Em3d and 12% for CG, for instance. For several applications the performance penalties are not as pronounced, but still there is no advantage to using a block size larger than 64 bytes.

These shared cache block size results follow the same trends observed for traditional caches, where block size increases often degrade performance when the cache is relatively small or when the miss rate improvement achieved by longer blocks does not compensate for their higher fetch latency.

5.3.3 Shared Cache Associativity

The NetCache architecture currently determines that a certain block can only be found on one cache channel, i.e. blocks are directly mapped to channels. In addition, the architecture determines that the block may be anywhere on the cache channel, i.e. cache channels are fully-associative. This organization greatly simplifies the ring subnetwork hardware for two reasons. The first is that a node that misses on a block does not require more than one tunable receiver, since it can simply tune the single receiver to the correct channel and wait for the block to show up. The second reason is that the home node does not have to worry about specific

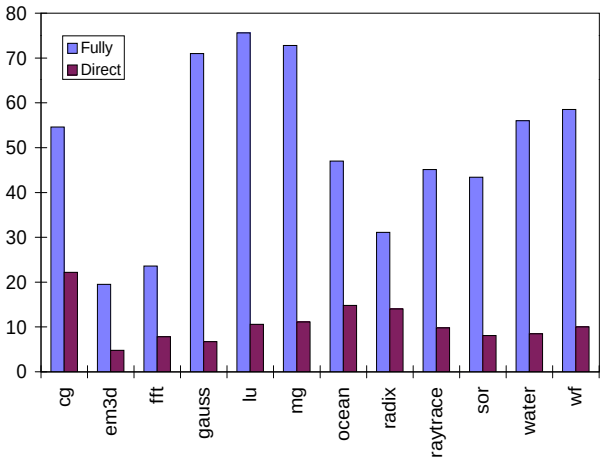

Figure 11: 32-KByte Shared Cache Hit Rates for Fully-Associative (left) and Direct-Mapped (right) Channels.

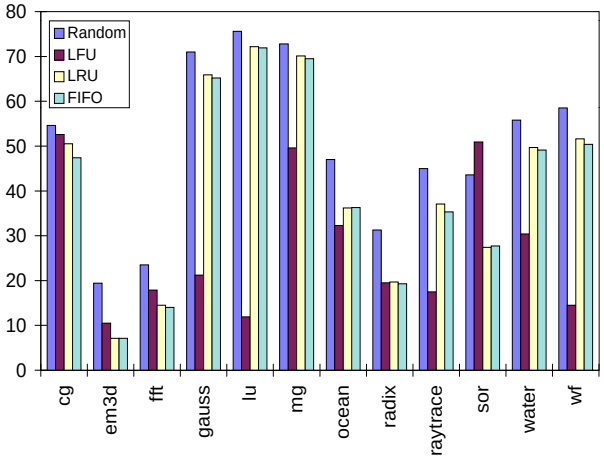

Figure 12: 32-KByte Shared Cache Hit Rates for (from left to right) Random, LFU, LRU, and FIFO Replacement Policies.

shared cache line addresses within the channel; the home node can simply insert the block requested in the first available shared cache line.

In order to evaluate whether it would be possible to improve the performance of NetCache-based multiprocessors by changing the shared cache associativity, we performed experiments where cache channels are directly-mapped, i.e. a certain block must be placed at a specific shared cache line within its cache channel. Changing the assignment of blocks to channels is not an option, since this would involve much greater hardware complexity.

Figure 11 presents the hit rates of a 32-KByte shared cache as a function of the associativity of each cache channel. The bar on the left of each group of bars represents the standard NetCache organization ("Fully"), while the bar on the right represents the direct-mapped cache channels option ("Direct"). The figure shows that direct-mapped channels achieve very low shared cache hit rates in all cases; hit rates are never higher than 25%. Thus, these results clearly justify our choice of shared cache associativity for the NetCache architecture.

These shared cache associativity results also follow the same trends observed for traditional caches, where fully-associative caches lead to higher hit rates than direct-mapped caches.

5.3.4 Shared Cache Replacement Policy

Another important aspect of the NetCache architecture is the shared cache block replacement policy. The architecture determines that a random block (the block contained in the next shared cache line to pass through the node) on the correct cache channel should be replaced, when the channel space has been exhausted and a new block must be inserted in the shared cache. Again, this policy greatly simplifies the hardware of the ring subnetwork, since the subnetwork interface need not keep track of reference or any other type of information. In addition, our standard policy optimizes the replacement operation, since the new block will be inserted in the shared cache as soon as possible.

In order to evaluate whether it would be possible to improve the performance of NetCache-based multiprocessors by using a better or more informed replacement policy, we performed experiments assuming LFU (Least Frequently Used), LRU (Least Recently Used), and FIFO (First In, First Out) policies.

Figure 12 presents the hit rates of a 32-KByte shared cache as a function of the replacement policy. From left to right, the bars in each group represent the random policy currently applied by the NetCache

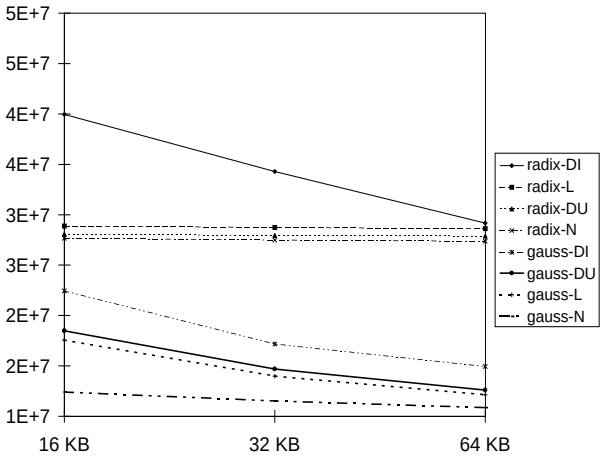

Figure 13: Run Time on 16 Nodes as a Function of 2nd-Level Cache Sizes.

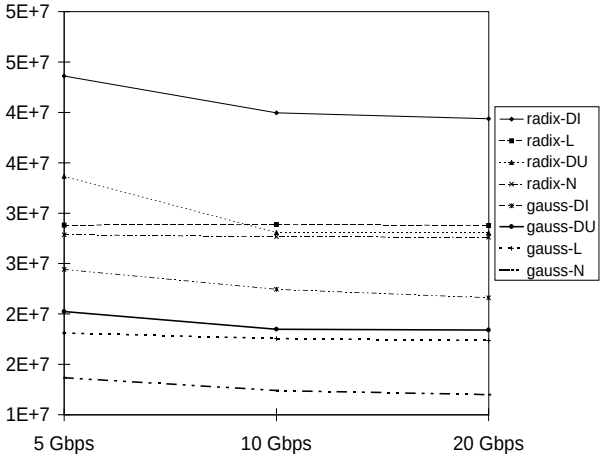

Figure 14: Run Time on 16 Nodes as a Function of Optical Frequencies.

architecture and the LFU, LRU, and FIFO policies. The figure shows that Random achieves the highest hit rates in all cases, except SOR. For certain applications, like Em3d, Ocean, and Radix, the superiority of the Random policy over all the other policies is substantial. The LRU and FIFO policies achieve comparable performance that is sometimes significantly worse than that of Random. Even though LFU exhibits the best result for SOR, it performs very poorly in comparison to the other policies in several cases (Gauss, LU, Mg, Raytrace, Water, and WF). Again, these results clearly justify our choice of shared cache replacement policy for the NetCache architecture.

These shared cache replacement policy results are somewhat surprising in that the more sophisticated strategies exhibit lower hit rates than the random strategy. The LRU replacement policy in particular usually achieves good performance for traditional caches. Our shared cache behaves differently from a traditional cache for two main reasons: a) each of our cache channels can only store 4 blocks and so the random strategy has a reasonable chance of choosing the best block to replace; and, more importantly, b) all processors in the system insert blocks in the shared cache, not just the local processor connected to a traditional cache. Given this characteristic and the different timings of the memory accesses issued by the processors in the system, policies such as LRU start to make much less sense.

5.4 Impact of Architectural Parameters

In this subsection we evaluate the impact of several of our simulation assumptions in order to understand the behavior of the NetCache architecture more fully. We start with a study of the effect of different 2nd-level cache sizes, moving on to an evaluation of the impact of the transmission rate, and finally addressing the effect of the memory block read latency. To simplify our analysis, we concentrate on one representative application from each of the *High-reuse* and *Low-reuse* groups, Gauss and Radix, respectively.

5.4.1 2nd-Level Cache Sizes

Figure 13 presents the running time performance of 16-node NetCache ("N"), LambdaNet ("L"), DMON-U ("DU"), and DMON-I ("DI") systems for Gauss and Radix, as a function of the size of the 2nd-level cache. Recall that our base experiments assumed 16-KByte 2nd-level caches. All the NetCache results assume a 32-KByte shared cache.

As one would expect, increasing the size of the 2nd-level cache reduces the potential benefit achievable by our shared cache, as long as the increase leads to read miss rate reductions. This is exactly what happens

with Gauss, an application with very good locality of reference. Nevertheless, the advantage of the NetCache system remains considerable in all cases, even for a 2nd-level cache 4 times as large as that in our base experiments.

In contrast, increases in 2nd-level cache size do not significantly affect the Radix miss rates, since this application exhibits terrible cache locality. As a result, the only system for which these increases improve the Radix running time is DMON-I. The improvement is a consequence of the reduced contention experienced by the system, when even a relatively small number of misses and block writebacks are avoided.

Finally, note the important result that for both Gauss and Radix, a 4-fold increase in the amount of 2nd-level cache space in the LambdaNet and DMON-based multiprocessors is not enough for these systems to outperform a NetCache system with 16-KByte 2nd-level caches and a 32-KByte shared cache. This result illustrates the fact that systems usually require a large amount of additional electronic cache memory to approach the benefit of a 32-KByte shared cache.

5.4.2 Transmission Rates

Figure 14 presents the running time performance of 16-node NetCache, LambdaNet, DMON-U, and DMON-I systems as a function of the optical transmission rate (bandwidth) of each channel. Recall that our base experiments assumed a 10 Gbits/s transmission rate. All the NetCache results assume a 32-KByte shared cache, i.e. we adjusted the length of the ring subnetwork with the inverse of variations in transmission rate. For instance, doubling the transmission rate was accompanied by halving the length of the ring.

The figure shows that applications suffer significant performance losses on the DMON-based multiprocessors with a transmission rate of 5 Gbits/s, particularly when applications exhibit poor locality of reference. The losses experienced by the NetCache and LambdaNet systems are not as significant. However, decreases in transmission rate do have a negative impact on the potential benefit of our shared cache, especially for the applications in the *High-reuse* group, since the percentage difference between shared cache hits and misses is reduced. Take the 5 Gbits/s rate, for instance. At this rate a shared read hit takes 68 pcycles, while a shared read miss takes 140 pcycles, a factor of 2 difference. At 10 Gbits/s, a shared read miss takes 2.6 times longer than a shared read hit.

The technology trend is one of increasing transmission rates however, rather than decreasing rates. Increases in transmission rate have a greater impact on the running time of the NetCache multiprocessor than on its competitors beyond a certain point, especially for the applications with significant data reuse in the shared cache.

5.4.3 Memory Block Read Latency

Figure 15 shows the running time performance of 16-node NetCache, LambdaNet, and DMON-based multiprocessors as a function of the memory block read latency (in pcycles). Recall that the memory block read latency we assumed in our base experiments was 76 processor cycles. The NetCache results again assume a 32-KByte shared cache.

The figure shows a few interesting results, the most important of which is that increases in latency increase the running time of the NetCache multiprocessor much less than the other systems. This behavior is, in fact, one of the great advantages of the NetCache system, since increases in latency increase the potential benefit achievable by our shared cache.

Given that the performance gap between processors and the memory system continues to widen at a frenetic pace, our results indicate that the performance benefits of our architecture will continue to increase for quite some time.

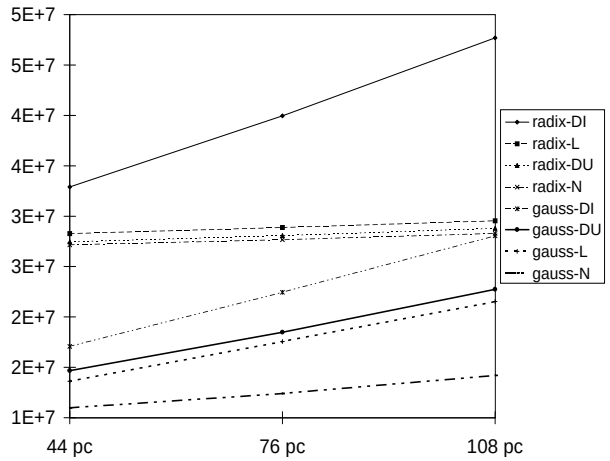

Figure 15: Run Time on 16 Nodes as a Function of Memory Block Read Latencies.

5.5 Comparison to Other Systems

To end this section we find it important to compare the NetCache architecture to yet another set of systems, at least in qualitative terms:

- We find a direct comparison between the NetCache and a traditional, electronic network-based scalable multiprocessor unnecessary, due to the significant performance disparity between the latter type of system and the scalable multiprocessors based on optical networks. This disparity comes from several factors, some of which are the network bandwidth, the possibility of having independent paths between different sets of nodes, and the ability to use snooping-based coherence.

- One might claim that it should be more profitable to use a simpler optical network (such as DMON or our star coupler subnetwork) and increase the size of 2nd-level caches than to use a full-blown NetCache. Even though this arrangement does not enjoy the *shared* cache properties of the NetCache architecture, it is certainly true that larger 2nd-level caches could improve the performance of certain applications enough to outperform a NetCache system with small 2nd-level caches. However, as shown in subsection 5.4.1, the amount of additional cache space necessary for this to occur can be quite significant. Furthermore, even on system with large 2nd-level caches, most applications should still profit from a shared cache at a lower level, especially as optical technology costs decrease and problem sizes, memory latencies (in terms of pcycles) and optical transmission rates increase.

- In the same vein, it is possible to conceive an arrangement where the additional electronic cache memory is put on the memory side of a simpler optical network. These memory caches would be shared by all nodes of the machine, but would add a new level to the hierarchy, a level which would effectively slow down the memory accesses that could not be satisfied by the memory caches. The NetCache also adds a level to the memory hierarchy, but this level does not slow down the accesses that miss in it. Thus, even in this scenario, the system could profit from a NetCache, especially as optical costs decrease and latencies and transmission rates increase.

6 Related Work

As far as we are aware, the NetCache architecture is the only one of its kind; a network/cache hybrid has never before been proposed and/or evaluated for multiprocessors. A few research areas are related to our

proposal however, such as the use of optical networks in parallel machines, the use of optics as delay line memory, and the use of shared caches in multiprocessors.

Delay line memories have been implemented in optical communication systems [19] and in all-optical computers [17]. These temporal memories exhibit an access time and storage capacity proportional to the channel length and data transmission rate [25]. Optical delay line memories have not been used as caches and have not been applied to multiprocessors before, even though optical networks have been a part of several parallel computer designs, e.g. [11, 15]. However, the only aspects of optical communication that these designs exploit are its high bandwidth and the ability to broadcast to a large number of nodes. Besides these aspects, the NetCache architecture also exploits the data storage potential of optics to its benefit.

A common approach to using optical communication in computer networks is through WDM networks [7]. The use of this type of networks has become widespread as a result of recent advances in tunable transmitters and receivers and integrated optics technology [12, 18]. A WDM network is ideal for small to medium-scale parallel computing as it can provide point-to-point channels between each pair of nodes on a single optical medium with broadcasting capability. Larger systems can be constructed by replacing this single-hop scheme with multi-hop or multidimensional WDM approaches [21, 6].

Optical networks with OTDM (Optical Time Division Multiplexing) have been proposed as an alternative to WDM networks, e.g. [26, 23]. OTDM networks do have some advantageous characteristics in comparison to WDM networks, but the OTDM technology is not yet mature. The NetCache architecture focuses on WDM technology due to its immediate availability, but there is nothing in our proposal that is strictly dependent on this specific type of multiplexing.

Optical networks with WDM have been a part of other scalable parallel computer designs. Ghose *et al.* [11] proposed a WDM-based optical bus called Optimul to explore the benefits of the concurrent operation of multiple channels for both shared-memory and message-passing parallel computers. The architecture of Optimul is similar to that of the LambdaNet. However, in order to reduce the number of receivers, several nodes share (i.e. transmit on) the same wavelength (channel). The medium access strategy used in Optimul is implemented through an electronic network that interconnects all the nodes and provides an arbitration mechanism. The LambdaNet is a performance upper bound for Optimul; contention for shared channels degrades Optimul's performance proportionally to the number of nodes sharing each channel. Optimul for a multiprocessor employs a snoopy write-update scheme to take advantage of the high bandwidth of optical links. Our comparison of NetCache and LambdaNet-based systems indicates that our network should also compare favorably against Optimul, since: a) the performance differences between NetCache and Optimul systems should be even larger than between the NetCache and LambdaNet systems; and b) the optical hardware cost of Optimul is only a constant factor better than that of the LambdaNet.

Ha and Pinkston [15] have proposed the DMON network and the DMON-I system studied in this paper. Our performance analysis has shown that the NetCache multiprocessor outperforms the DMON-based systems in most cases. However, DMON-based systems have an advantage over the NetCache architecture: they allow certain latency tolerance techniques, such as prefetching or multithreading, that the NetCache architecture, as presented here, does not. This limitation results from the star coupler subnetwork having a single tunable receiver that must be tuned to a single home channel on a read access. Latency tolerance techniques could be implemented on top of our architecture, if it were extended with a larger number of tunable receivers. However, given that the shared cache itself is a latency tolerance mechanism, this type of extension might not be cost-effective.

Several studies have proposed the use of shared caches for single and multi-chip multiprocessors, e.g. [22, 9, 8, 20, 2, 24]. In terms of single-chip multiprocessors, Nayfeh and Olukotun [22] have proposed the construction of clusters of processors on one chip or chip set. In their system processors share a large cache, but do not have individual processor caches. In [9] the design of a single-chip cluster with local caches and a shared second-level cache is discussed. [8] reports results for a shared processor cache that is infinite and fully associative, and compares this system to one with an individual cache for each processor.

A few studies have considered network caches for multi-chip multiprocessors. Each node in the ASURA system [20] is a processor cluster with a network interface that contains part of the global address space and caches remote data. In the same vein, Bennett *et al.* [2] evaluate the benefit of adding a shared cache to the network interface of a processor cluster as a means of improving the performance of networked workstations configured as a distributed shared-memory multiprocessor. Finally, Stache [24] implements a fully-associative, software-controlled alternative to a network cache.

Even though NetCaches avoid accesses to main memory just like shared on-chip caches and network caches do, our work differs from these approaches in several ways: a) the NetCache stores data from all nodes in a potentially large parallel system, not just the from the ones that can fit on a single chip; b) the NetCache stores data without repetition, while isolated network caches might waste space storing multiples copies of data; c) in addition to storing data, the NetCache serves as a system-wide network; and d) the NetCache is based on optics, not electronics.

7 Conclusions

In this paper we proposed a novel architectural concept: an optical network/cache hybrid (and associated coherence protocol) called NetCache. Through a large set of detailed simulations we showed that NetCache-based multiprocessors easily outperform other optical network-based systems, especially for the applications with a large amount of data reuse in the NetCache. In addition, we justified our NetCache design decisions through an extensive evaluation of the shared cache organization and policies. Finally, our parameter space study shows that the benefit of the NetCache architecture will increase in the future. Based on these results and on the continually-decreasing cost of optical components, our main conclusion is that the NetCache architecture is highly efficient and should definitely be considered by multiprocessor designers.

Acknowledgements

The authors would like to thank Luiz Monnerat, Cristiana Seidel, Rodrigo dos Santos, and Lauro Whately for comments that helped improve the presentation of the paper. We would also like to thank Timothy Pinkston and Joon-Ho Ha for their careful evaluation of our work and for discussions that helped improve this paper significantly.

References

[1] D. Bailey *et al.* The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[2] J. K. Bennett, K. E. Fletcher, and W. E. Speight. The Performance Value of Shared Network Caches in Clustered Multiprocessor Workstations. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.

[3] C. A. Brackett. Dense Wavelength Division Networks: Principles and Applications. *IEEE Journal on Selected Areas of Communication*, 8, Aug 1990.

[4] E. V. Carrera and R. Bianchini. OPTNET: A Cost-Effective Optical Network for Multiprocessors. Technical Report Tech. Report ES-457/97, COPPE Systems Engineering, Federal University of Rio de Janeiro, December 1997.

[5] D. Culler *et al.* Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[6] K. R. Desai and K. Ghose. An Evaluation of Communication Protocols for Star-Coupled Multidimensional WDM Networks for Multiprocessors. In *Proceedings of the 2nd Intl. Conf. On Massive Parallel Processing Using Optical Interconnections*, Oct 1995.

[7] P. W. Dowd and J. Chu. Photonic Architectures for Distributed Shared Memory Multiprocessors. In *Proceedings of the 1st International Workshop on Massively Parallel Processing using Optical Interconnections*, pages 151–161, April 1994.

[8] A. Erlichson *et al.* The Benefi ts of Clustering in Shared Address Space Multiprocessors: An Applications-Driven Investigation. Technical report, Computer Systems Laboratory. Stanford University, October 1994.

[9] M. Farrens, G. Tyson, and A. R. Pleszkun. A Study of Single Chip Processor/Cache Organizations for Large Numbers of Transistors. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 338–347, April 1994.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[11] K. Ghose, R. K. Horsell, and N. Singhvi. Hybrid Multiprocessing in OPTIMUL: A Multiprocessor for Distributed and Shared Memory Multiprocessing with WDM Optical Fiber Interconnections. In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994.

[12] B. S. Glance, J. M. Wiesenfeld, U. Koren, and R. W. Wilson. New Advances on Optical Components Needed for FDM Optical Networks. *IEEE Photonics Technical Letters*, 5(10):1222–1224, October 1993.

[13] M. S. Goodman *et al.* The LAMBDANET Multiwavelength Network: Architecture, Applications, and Demonstrations. *IEEE Journal on Selected Areas in Communications*, 8(6):995–1004, August 1990.

[14] P. E. Green. Optical Networking Update. *IEEE Journal on Selected Areas in Communications*, 14(5):764–779, June 1996.

[15] J.-H. Ha and T. M. Pinkston. SPEED DMON: Cache Coherence on an Optical Multichannel Interconnect Architecture. *Journal of Parallel and Distributed Computing*, 41:78–91, 1997.

[16] E. Hall *et al.* The Rainbow-II Gigabit Optical Network. *IEEE Journal on Selected Areas in Communications*, 14(5):814–823, June 1996.

[17] H. F. Jordan, V. P. Heuring, and R. J. Feuerstein. Optoelectronic Time-of-Flight Design and the Demonstration of an All-Optical, Stored Program. *Proceedings of IEEE. Special issue on Optical Computing*, 82(11), Nov 1994.

[18] L. G. Kasovsky, T. K. Fong, and T. Hofmeister. Optical Local Area Network Technologies. *IEEE Communications Magazine*, pages 50–54, December 1994.

[19] R. Langenhorst *et al.* Fiber Loop Optical Buffer. *Journal of Lightwave Technology*, 14(3):324–335, March 1996.

[20] S. Mori *et al.* A Distributed Shared Memory Multiprocessor: ASURA - Memory and Cache Architectures. In *Proceedings of Supercomputing 93*, pages 740–749, 1993.

[21] B. Mukherjee. WDM-Based Local Lightware Networks - Part II: Multihop Systems. *IEEE Network*, pages 20–32, July 1992.

[22] B. A. Nayfeh and K. Olukotun. Exploring the Design Space for a Shared-Cache Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 166–175, April 1994.

[23] A. G. Nowatzyk and P. R. Prucnal. Are Crossbars Really Dead? The Case for Optical Multiprocessor Interconnect Systems. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 106–115, June 1995.

[24] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Thyphoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325–337, April 1994.

[25] D. B. Sarrazin, H. F. Jordan, and V. P. Heuring. Fiber Optic Delay Line Memory. *Applied Optics*, 29(5):627–637, Feb 1990.

[26] D. M. Spirit, A. D. Ellis, and P. E. Barnsley. Optical Time Division Multiplexing: Systems and Networks. *IEEE Communications Magazine*, pages 56–62, December 1994.

[27] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Effi cient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, 1994.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, May 1995.