Efficient Communication Using Message Prediction for Cluster of Multiprocessors

Ahmad Afsahi and Nikitas J. Dimopoulos

Technical Report ECE-99-5

Department of Electrical and Computer Engineering, University of Victoria
P.O. Box 3055, Victoria, B.C., Canada, V8W 3P6
{aafsahi, nikitas}@ece.uvic.ca

December, 1999

Abstract. *With the increasing uniprocessor and SMP computation power available today, interprocessor communication has become an important factor that limits the performance of cluster of workstations. Many factors including communication hardware overhead, communication software overhead, and the user environment overhead (multithreading, multiuser) affect the performance of the communication subsystems in such systems.*
A significant portion of the software communication overhead belongs to a number of message copying. Ideally, it is desirable to have a true zero-copy protocol where the message is moved directly from the send buffer in its user space to the receive buffer in the destination without any intermediate buffering. However, due to the fact that message-passing applications at the send side do not know the final receive buffer addresses, early arrival messages have to be buffered at a temporary area.
In this paper, we show that there is a message reception communication locality in message-passing applications. We have utilized this communication locality and devised different message predictors at the receiver sides of communications. In essence, these message predictors can be efficiently used to drain the network and cache the incoming messages even if the corresponding receive calls have not been posted yet. The performance of these predictors, in terms of hit ratio, on some parallel applications are quite promising and suggest that prediction has the potential to eliminate most of the remaining message copies.

1.0 Introduction

With the increasing uniprocessor and SMP computation power available today, interprocessor communication has become an important factor that limits the performance of workstations clusters. Essentially, communication overhead is one of the most important factors affecting the performance of parallel computers. Many factors affect the performance of communication subsystems in parallel systems. Specifically, communication hardware and its services, communication software, and the user environment (multiprogramming, multiuser) are the major sources of the communication overhead.

Communication software overhead currently dominates communication time in cluster of workstations. In the current generation of parallel computer systems, the software overheads are tens of microseconds [14]. This is worse in cluster of workstations. Even with high performance networks [8, 18] available today, there is still a gap between what the network can offer and what the user applica-

tion can see. The communication software overhead cost comes mainly from three different sources; crossing protection boundaries several times between the user space and the kernel space, passing several protocol layers, and involving a number of memory copying.

Several researchers are working to minimize the cost of crossing protection boundaries, and using simpler protocol layers by utilizing *user-level messaging* techniques such as active messages (AM) [36], fast messages (FM) [28], VMMC-2 [16], U-Net [37], LAPI [32], BIP [29], VIA [17], and PM [35]. A significant portion of the software communication overhead belongs to a number of message copying. Ideally, message protocols should transfer messages in a single copy (this is usually called a true zero-copy). In other words, the protocol should copy the message directly from the send buffer in its user space to the receive buffer in the destination without any intermediate buffering. However, the application at the send side does not know the final receive buffer addresses and, hence, the communication subsystems at the receiving end still copy messages unnecessarily from the network interface to a system buffer, and then from the system buffer to the user buffer when the receiving application posts the receive call.

Some researchers have tried to avoid memory copying [16, 24, 30, 5, 34, 33]. While they have been able to remove the memory copying between the application buffer space and the network interface at the sender side by using user-level messaging techniques, they haven't been able to remove the memory copying at the receiver sides completely. They may achieve a zero-copy messaging at the receiver sides only if the receive call is already posted, a rendez-vous type communication is used for large messages, or the destination buffer address is already known by a pre-communication. Note, however, that MPI-2 [26] supports a remote memory access operation but this is mostly suitable for receiver-initiated communications arising from the shared-memory paradigm.

We are interested in bypassing the memory copying at the destination in the general case, synchronous or asynchronous, eager or rendez-vous and for sender-initiated communications as in MPI [25, 26]. In this paper, we argue that it is possible to address the message copying problem at the receiving side by speculation. We support our claim by showing that messages display a form of locality at the receiving ends of communications.

This paper, for the first time as far as the authors know, introduces the notion of message prediction for the receiving side of message-passing systems. By predicting the next receive communication call, and hence the next destination buffer address, before the receiving call is posted we will be able to copy the message directly into the CPU cache speculatively before it is needed so that an effect of a zero-copy can be achieved.

We are interested in utilizing similar predictors as in [1, 2], but this time at the receiver sides to predict the next consumable message and drain the network as soon as the message arrives. Upon a message arrival, a user-level thread is invoked. If the receive call has not been issued yet, the message will be cached, but efficient cache mapping mechanisms need to be devised to facilitate binding at the moment the receive call is issued. If the receive call has already been issued, then the message can be written to its final destination.

The first contribution of this paper is that we show evidence that there exists message communication locality at the receiver sides of message-passing parallel applications. The second contribution of this work is the introduction and evaluation of different message predicting techniques for the receiving side of message-passing systems.

This paper concentrates on message predictions at the destinations in message-passing systems using MPI in isolation. This is analogous to branch prediction, and coherence activity prediction [27] in isolation. Our tools are not ready for measuring the effectiveness of our predictors on the application run-time yet. Our preliminary evaluation measures the accuracy of the predictors in terms of hit ratio. The results are quite promising and suggest that prediction has the potential to eliminate most of the remaining message copies.

In Section 2.0 of this paper, we explain the motivation behind this work and mention related works. We elaborate on how prediction would help eliminate the message copies at the receiving side of communications, in Section 3.0. Our experimental methodologies to gather communication traces of our parallel applications are explained in Section 4.0. In Section 5.0, we show communication frequency and unique message identifier distributions in the applications, and present evidence of message locality at the receiver sides. In Section 6.0, we propose our message predictors and present their performance on the applications. Finally, we conclude our paper in Section 7.0.

2.0 Motivation and Related Work

High performance computing is increasingly concerned with efficient communication across the interconnect due to the availability of high-speed highly-advanced processors. Modern switched networks, called *System Area Networks* (SAN), such as Myrinet [8] and ServerNet [18], provide high communication bandwidth and low communication latency. However, because of high processing overhead due to communication software including network interface control, flow control, buffer management, memory copying, polling and interrupt handling, users cannot see much difference compared to traditional local area networks.

Fortunately, several user-level messaging techniques have been developed to remove the operating system kernel and protocol stack from the critical path of communications [36, 28, 16, 37, 17, 29, 32, 35]. This way, applications can send and receive messages without operating system intervention which often greatly reduces the communication latency.

Data transfer mechanisms and message copying, control transfer mechanisms, address translation mechanisms, protection mechanisms, and reliability issues are the key factors for the performance of a user-level communication system. In this paper, we are particularly interested to avoid message copying at the receiver sides of communications.

A significant portion of the software communication overhead belongs to a number of message copying. With the traditional software messaging layers, there are usually four message copying operations from the send buffer to the receive buffer, as shown in Figure 1. These copies are namely from the send buffer to the system buffer (1), from the system buffer to the *network interface* (NI) (2), and at the other end of communication from the network interface to the system buffer (3), and from the system buffer to the receive buffer (4) when the receive call is posted. Note that, we haven't considered data transfer from the network interfac*e* (NI) at the sending process to the network interface at the receiving process as a separate copy. Also, the network interface's place can be either on the I/O bus or on the memory bus.

At the send side, some user-level messaging layers use programmed I/O to avoid system buffer copying. FM uses programmed I/O while AM-II and BIP do so only for small messages. Some other user-messaging layers use DMA. VMMC-2, U-Net, and PM use DMA to bypass the system buffer copy while AM-II and BIP do so only for large messages. In systems that use DMA, applications or a

library dynamically pins and unpins pages in the user space that contain the send and the receive buffers. Address translation can be done using a kernel module as in BIP, or by caching a limited number of address translations for the pinned pages as in VMMC-2, U-Net/MM [6], and PM. Some network interfaces also permit bypassing message copying at the network interface by directly writing into the network.

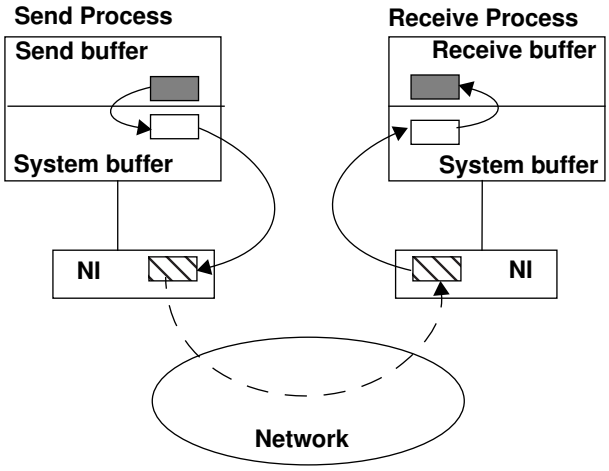


Figure 1. Data transfers in a traditional messaging layer

On the contrary to the send side, bypassing the system buffer copying at the receiving side may or may not be achievable. Processes at the sending sides do not know the destination buffer addresses. Therefore, when a message arrives at the receiving side it has to be buffered if the receive call has not been posted yet. VMMC [7] for the SHRIMP multicomputer is a communication model that provides direct data transfer between the sender's and receiver's virtual address space. However, it can achieve zero-copy transfer only if the sender knows the destination buffer address. Therefore, the receiver exports its buffer address by scouting a message to the sender before the actual transmission can take place.

VMMC-2 [16], uses a *transfer redirection* mechanism instead. It uses a default, redirectable receive buffer for a sender who does not know the address of the receive buffer. When a message arrives at the receiving network interface, the redirection mechanism checks to see if the receiver has already posted its buffer address. If the receive buffer has been posted earlier than the message arrival, the message will be directly transferred to the user buffer. Thus it achieves a zero-copy transfer. If the buffer address is not posted, the message must be buffered in the default buffer. It will then be transferred when the receive buffer is posted. Thus, it achieves a one-copy transfer. However, if the receiver posts its buffer address when the message arrives, part of the message is buffered at the default buffer and the rest is transferred to the user buffer.

Fast sockets [30] has been built using active messages. It uses a mechanism at the receiver side called *receive posting* to avoid the message copy in the fast socket buffer. If the message handler knows that the data's final memory destination is already known upon message arrival the message is directly moved to the application user space. Otherwise, it has to be copied into the fast socket buffer.

FM 2.x [24] uses a similar approach as fast sockets, namely *layer interleaving*. FM collaborates with the handler to direct the incoming messages into the destination buffer if the receive call has already been posted.

MPI-LAPI [5] is an implementation of MPI on top of LAPI [32] for the IBM SP machines. In the implementation of the eager protocol, the header handler of the LAPI returns a buffer pointer to LAPI which tells LAPI where the packets of the message must be reassembled. If a receive call has been posted, the address of the user buffer is returned to LAPI. If the header handler doesn't find a matching receive, it will return the address of an *early arrival buffer* and hence a one-copy transfer is accomplished. Meanwhile, message sizes of larger than eager size is transferred using 2-phase rendez-vous protocol.

MPICH-PM/CLUMP [33] is an MPI library implemented on a cluster of SMPs. It uses a message-passing only model where each process runs on a processor of an SMP node. For internode communications, it uses *eager* and *rendez-vous* protocols internally. For short messages, it achieves one-copy using eager protocol as the message is copied into a temporary buffer if the MPI receive primitive has not been issued. For large message, it uses rendez-vous protocol to achieve zero-copy by using a remote write operation but it needs an extra communication. For intranode communications, it achieves a one-copy using a kernel primitive that allows to copy messages from the sender to the receiver without involving the communication buffer.

TOMPI [12] is a threaded implementation of MPI on a single SMP node. It copies a message only once by utilizing multiple threads on an SMP node. Unfortunately, it is not scalable to cluster of SMP machines.

Another technique to bypass extra copying is the *re-mapping* technique. A zero-copy TCP stack is implemented in Solaris by using copy-on-write pages and re-mapping to improve communication performance [10]. It achieves a relatively high throughput for large messages. However, it does not have a good performance for small messages. This work is also solely dedicated to the SUN Solaris virtual memory system.

fbufs [15] is also using the re-mapping technique to avoid the penalty of copying large messages across different layers of protocol stack. However, fbufs allows re-mapping only for a limited range of user virtual memory.

It is quite clear that the user-level messaging techniques may not achieve a zero-copy communication all the time at the receiver side of communications. Meanwhile, the major problem with all page re-mapping techniques is their poor performance for short messages which is extremely important for parallel computing.

Prediction techniques have been proposed in the past to predict the future accesses of sharing patterns and coherence activities in distributed shared memory (DSM) by looking at their observed behavior [27, 23, 20, 39, 11, 31]. These techniques assume that memory accesses and coherence activities in the near future will follow past patterns. Sakr and his colleagues have used time series and neural networks for the prediction of the next memory sharing requests [31]. Dahlgren and his colleagues devised hardware regular stride techniques to prefetch several blocks ahead of the current data block [11]. More elaborate hardware-based irregular stride prefetching approaches have been proposed by Zhang and Torrellas [39]. Kaxiras and Goodman have recently proposed an instruction-based approach which maintains the history of load and store instructions in relation to cache misses and predicting their future behavior [20]. Mukherjee and Hill proposed a general pattern-based predictor to learn and predict the coherence activity for a memory block in a DSM [27]. In a recent paper, Lai and Falsafi proposed a new class of pattern-based predictors, *memory sharing predictors*, to eliminate the coherence overhead on a remote access latency by just predicting the memory request messages [23].

As stated above, many prediction techniques have been proposed to reduce or hide the latency of a remote memory access in shared memory systems. Recently, Afsahi and Dimopoulos proposed some heuristics to predict the destination target of subsequent communication requests at the send side of communications in message-passing systems [1, 2]. However, to the best of our knowledge, no prediction technique has been proposed for the receive side of communications in message-passing systems to reduce the latency of a message transfer.

This paper, reports on an innovative approach for removing message copying at the receiving ends of communications for message-passing systems. We argue that it is possible to address the message copying problem at the receiving sides by speculation. We introduce message prediction techniques such that messages can be directly transferred to the cache even if the receive calls have not been posted yet.

3.0 Using Message Predictions

In this section, we analyze the problem with the early arrival of messages at the destinations in message-passing systems. In such systems, a number of messages arrive in arbitrary order at the destinations. The consuming process or thread will consume one message at a time. If we know which message is going to be consumed next, we can move the message upon its arrival to near the place that it is to be consumed (e.g. a staging cache).

For this, we have to consider three different issues. First, deciding which message is going to be consumed next. This can be done by devising receive call predictors, history-based predictors that predict subsequent receive calls by a given node in a message-passing program, Second, deciding where and how this message is to be moved in the cache. Third, efficient cache re-mapping and late binding mechanisms need to be devised for when the receive call is posted.

In this work, we are addressing the first problem. That is, devising message predictors and evaluating their performance. We are working on several methods to address the remaining issues. We shall report on these issues in the future.

4.0 Experimental Methodology

In exploring the effect that different heuristics have in predicting the next receive call, we utilized a number of parallel benchmarks, and extracted their communication traces on which we applied our predictors.

We have used some well-known parallel benchmarks form the *NAS parallel benchmarks* (NPB) suite [4], and the *Parallel Spectral Transform Shallow Water Model* (PSTSWM) application [38]. We used the MPI [25] implementation of the NPB suite (version 2.3), and version 6.2 of the PSTSWM application.

We are only interested in the patterns of the point-to-point communications between pair-wise nodes in our applications. For this, we executed these applications on an IBM SP2 machine. We wrote our own profiling code using the wrapper facility of the MPI to gather the communication traces. We did this by inserting monitor operations in the profiling MPI library for the communication related activities. These operations include arithmetic operations for the calculation of the desired characteristics. Collecting communication traces does not affect the communication patterns of the applications.

We considered different system sizes and problem sizes for our applications to evaluate the performance of our prediction heuristics. Specifically, we experimented with the workstation class "W", and the large class "A" of the NPB suite, and the default problem size for the PSTSWM application. The NPB results are almost the same for "W" and "A" classes. Hence, we report only for the "A" class here. Note that we also removed the initialization part from the communication traces of the PSTSWM application. Although the derived results are for the above mentioned parallel applications, however, we believe that these applications are representative of the existing scientific and engineering parallel applications.

5.0 Receiver-side Locality Estimation

Our applications use synchronous and asynchronous MPI receive primitives, namely *MPI_Recv* and *MPI_Irecv* [25]. *MPI_Recv (buf, count, datatype, source, tag, comm, status)* is a standard blocking receive call. When it returns, data is available at the destination buffer. The PSTSWM application uses this type of receive call. *MPI_Irecv (buf, count, datatype, source, tag, comm, request)* is a standard nonblocking receive call. It immediately posts the call and returns. Hence, data is not available at the time of return. It needs another call to complete the call. All applications in our study use this type of receive call.

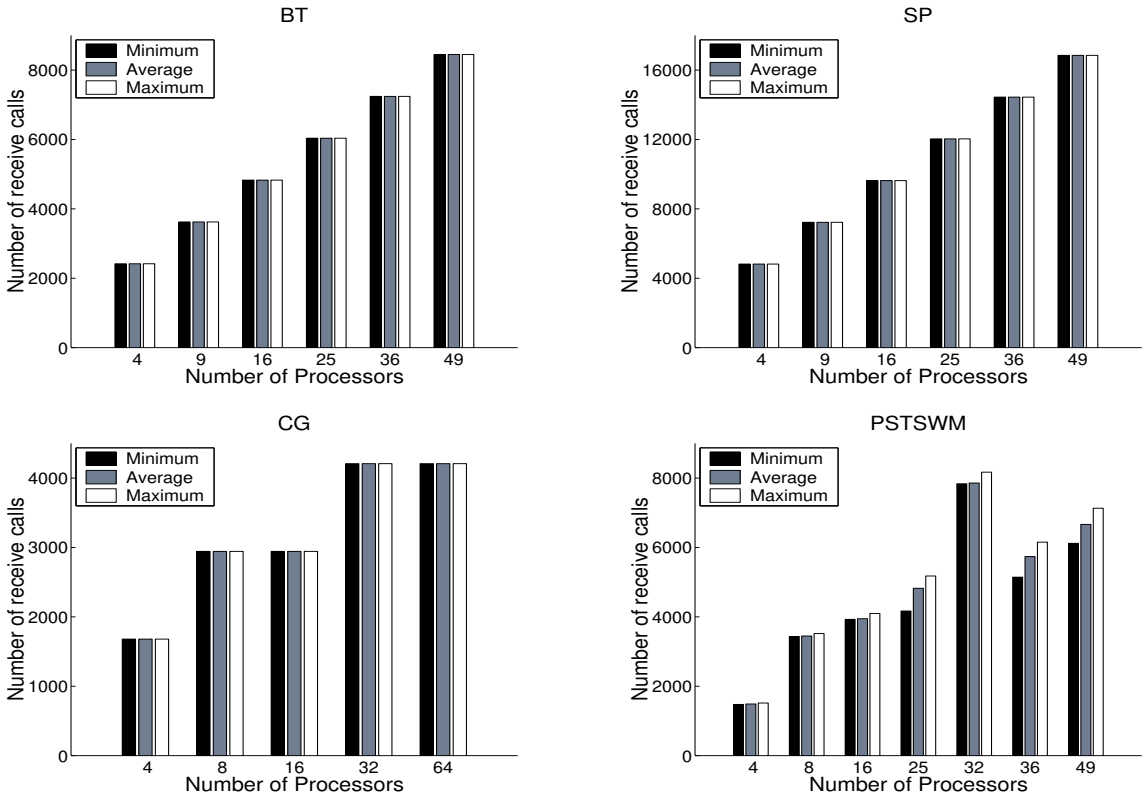


Figure 2. Number of receive calls in the applications under different system sizes

One of the communication characteristics of any parallel application is the frequency of communications. Figure 2, illustrates the minimum, average, and maximum number of receive communication calls in the applications under different system sizes. We ran our applications once for each

different system size and counted the number of receive calls for each node of the applications. Hence, in Figure 2, by average, minimum, and maximum, we mean the average, minimum, and maximum number of receive calls taken over all nodes of each application. It is clear that all nodes in the BT, SP, and CG applications have the same number of receive communication calls. While nodes in the PSTSWM application have different number of receive communication calls.

As stated earlier, MPI_Recv and MPI_Irecv calls have a 7-tuple set consisting of *source, tag, count, datatype*, *buf, comm,* and *status* or *request.* In order to choose precisely one of the received messages at the network interface and transfer it to the cache, our predictors need to consider all the details of a message envelop. That is, *source, tag, count, datatype*, *buf,* and *comm* (we don't consider *status* and *request* as they are just a handle when the calls return). We cannot rely only on the buffer address, *buf*, of a receive call as many nodes may send their messages to the same buffer address of a particular destination node. We cannot also rely only on the sender, *source*, of a message, or on the length, *count*, of a message. We can only rely on the combination of all six fields. Therefore, we assign a different identifier for each unique 6-tuple found in the communication traces of the applications. Figure 3, shows the number of *unique message identifiers* in our applications under different system sizes. By average, minimum, and maximum, we mean the average, minimum, and maximum number of unique identifiers taken over all nodes of each application. It is evident that all nodes in the BT, and CG applications have the same number of unique message identifiers while nodes in the SP, and PSTSWM applications have different number of unique message identifiers (except when the number of processors is four for the SP benchmark).

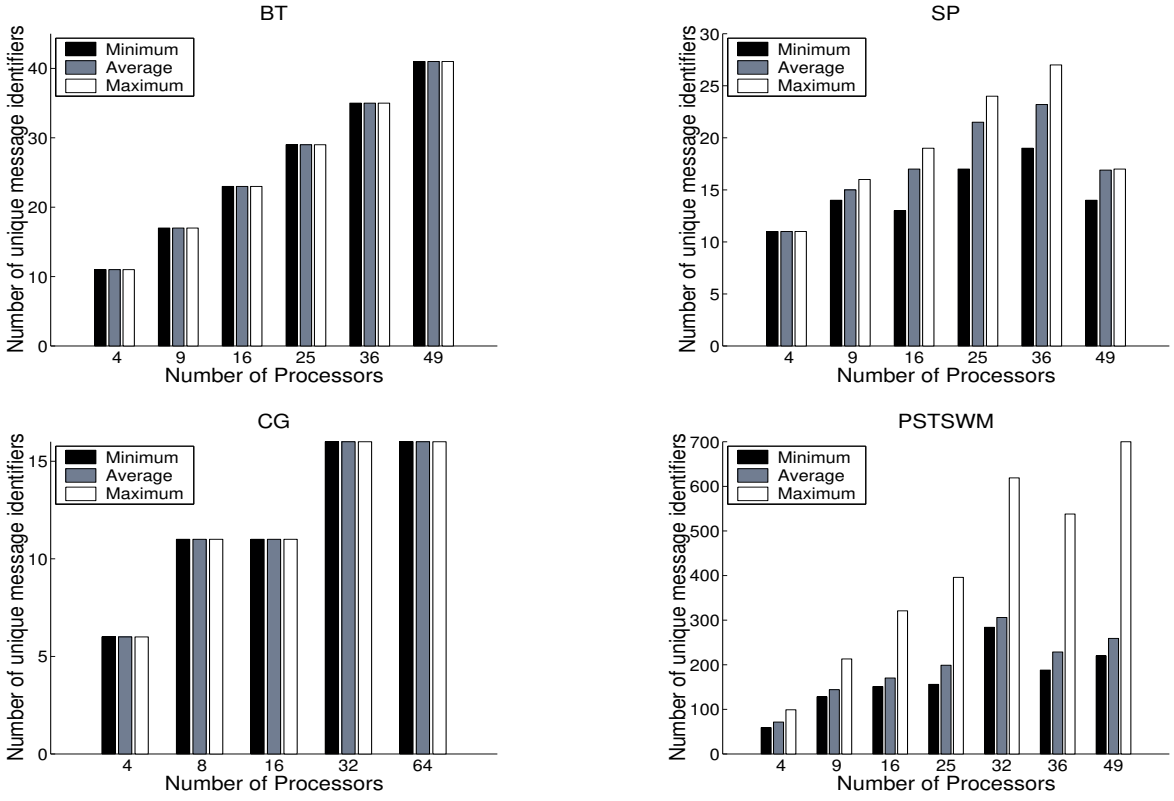


Figure 3. Number of unique message identifiers in the applications under different system sizes

Figure 4, shows the distribution of each unique message identifier for node zero of the applications when the number of processors is 64 for CG and 49 for the other applications. We chose node zero because this node has the largest number of unique message identifiers among all nodes and is also responsible for distributing data and verifying the results of the computation. As it is shown in Figure 4, the message identifiers are evenly distributed in BT. However, the distribution of the message identifiers in CG and PSTSWM are almost bimodal with two separated peaks. The SP benchmark shows four different peaks for the message identifiers.

5.1 Communication Locality

In the context of message passing programming, many parallel algorithms are built from loops consisting of computation and communication phases. Therefore, communication patterns may be repetitive. This has motivated researchers to find or use the *communications locality* properties of parallel applications [1, 2, 21, 19, 22, 13, 9]. Kim and Lilja [21] have shown that there is a locality in message destination, message sizes, and consecutive runs of send/receive primitives in parallel algorithms. They have proposed and expanded the concept of memory access locality based on the *Least Recently Used*, LRU, stack model to determine these localities. In [1, 2], Afsahi and Dimopoulos have shown the communication locality of message-passing application in terms of message destination locality. Karlsson and Brorsson [19] have compared the communication properties of parallel applications in message-passing systems using MPI, and shared memory systems using TreadMarks [3].

We define the terms *message reception locality* in conjunction with this work. By message reception locality we mean that if a certain message reception call has been used it will be re-used with high probability by a portion of code that is "near" the place that was used earlier, and that it will be re-used in the near future.

In the following subsection, we present the performance of the classical LRU, LFU, and FIFO heuristics on the applications to see the existence of locality or repetitive receive calls. We use the *hit ratio* to establish and compare the performance of these heuristics. As a hit ratio, we define the percentage of times that the predicted receive call was correct out of all receive communication requests.

5.2 The LRU, FIFO and LFU Heuristics

The *Least Recently Used* (LRU)*, First-In-First-Out* (FIFO), and *Least Frequently Used* (LFU) heuristics, all maintain a set of k (k is the window size) unique message identifiers. If the next message identifier is already in the set, then a hit is recorded. Otherwise, a miss is recorded and the new message identifier replaces one of the identifiers in the set according to which of the LRU, FIFO or LFU strategies is adopted.

Figure 5, shows the results of the LRU, FIFO, and LFU heuristics on the application benchmarks when the number of processors is 64 for CG and 49 for all other applications. It is clear that the hit-ratios in all benchmarks approach 1 as the window size increases. The performance of the FIFO algorithm is the same as the LRU for BT, and PSTSWM benchmarks, and almost the same for the SP and CG benchmarks. The LFU algorithm consistently has a better performance than the LRU and FIFO heuristics on the BT, CG, and PSTSWM applications. It also has a better performance than the LRU and FIFO heuristics on the SP benchmark for window sizes of greater than five.

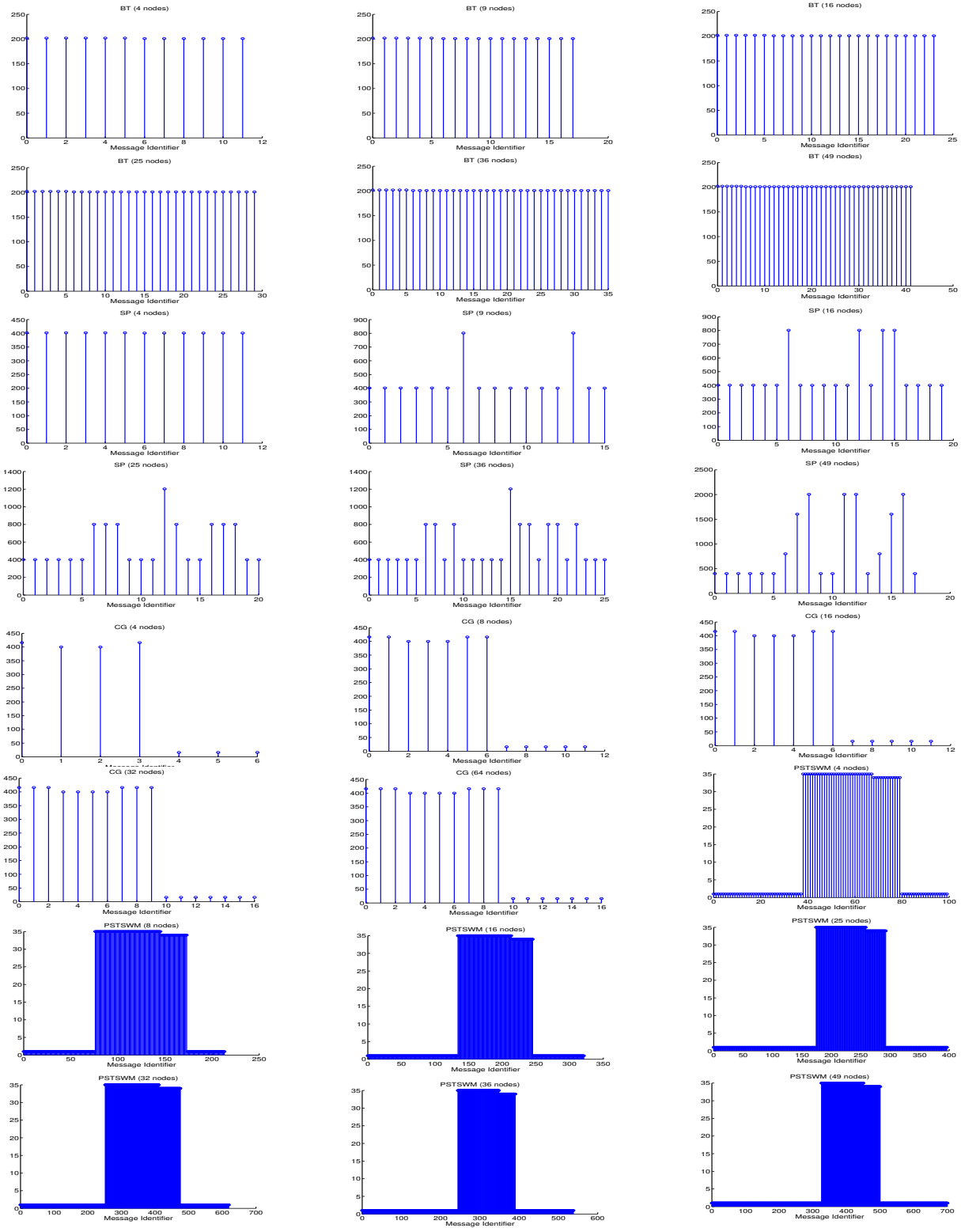


Figure 4. Distribution of the unique message identifiers in the applications under different system sizes

Essentially, the LRU, FIFO and LFU heuristics do not predict exactly the next receive call but shows the probability that the next receive call might be in the set. For instance, the SP benchmark shows nearly 60% hit ratio for a window size of five under the LRU heuristic. This means that 60% of the time one of the five most recently issued call will be issued next. These heuristics perform better when the window size k is sufficiently large. However, this large window adds to the hardware/software implementation complexity as one need to move all messages in the set to the cache in the likelihood that one of them is going to be used next. This is prohibitive for large window sizes.

We are interested to devise predictors that can predict the next receive call with a high probability. In Section 6.0, we introduce our novel message predictors employing different heuristics and evaluate their performance.

6.0 Message Predictors

The set of predictors introduced in this section predict the subsequent receive calls based on the past history of communication patterns on a per node basis. These heuristics were originally proposed in [1, 2] to predict the destination target of subsequent communication requests at the sender sides of communications to reconfigure the interconnect concurrent to the computation. These predictors can be used dynamically at the communication assist with or without the help of a programmer or a compiler. In the following figures, by average, minimum, and maximum, we mean the average, minimum, and maximum hit ratio taken over all nodes of each application.

6.1 The Tag Predictor

The Tag predictor assumes a static communication environment in the sense that a particular communication receive call in a section of code, will be the same one with a large probability. We attach a different *tag* (this is different than the tag in an MPI communication call; It may be a unique identifier or the program counter at the address of the communication call) to each of the receive calls found in the applications. This can be implemented with the help of the compiler or by the programmer through a *pre-receive (tag)* operation which will be passed to the communication subsystem to predict the next receive call before the actual receive call is issued.

To this tag and at the communication assist, we assign this receive call. A hit is recorded if in subsequent encounters of the tag, the requested communication is the same as the receive call already associated with the tag. Otherwise, a miss is recorded and the tag is assigned the newly requested receive call. The performance of the Tag predictor is shown in Figure 6. It is evident that this predictor doesn't have a good performance on the applications. It cannot predict the communication patterns of PSTSWM at all, and has a degrading performance for all other applications when the number of processors increases.

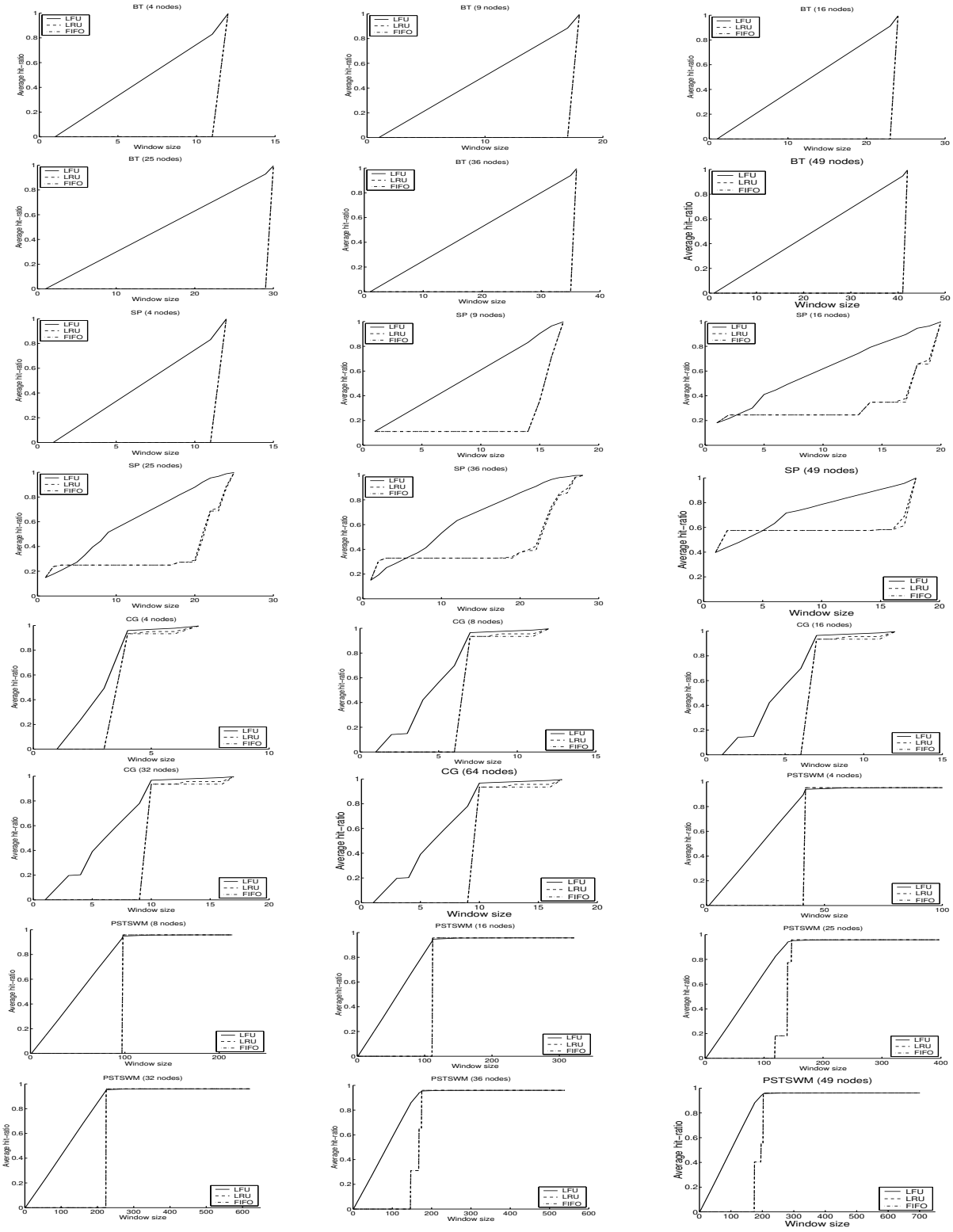


Figure 5. Effects of the LRU, FIFO, and LFU heuristics on the applications under different system sizes

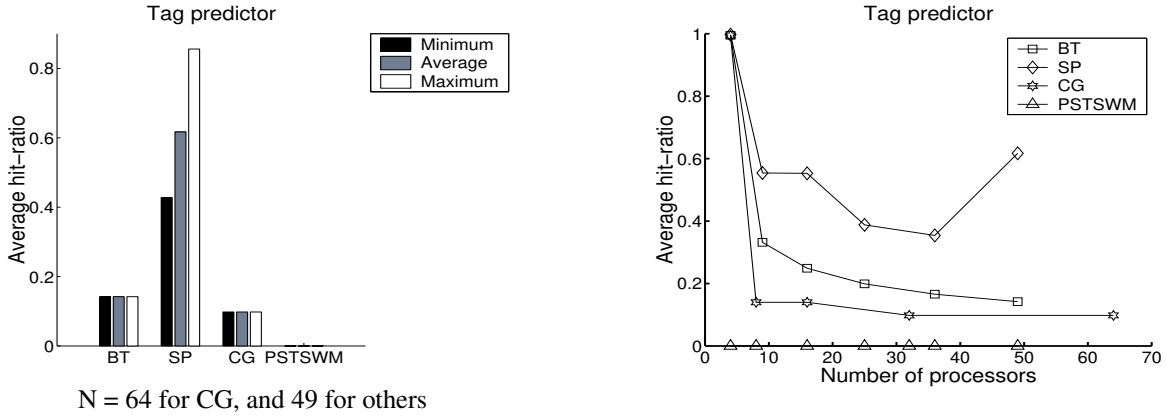
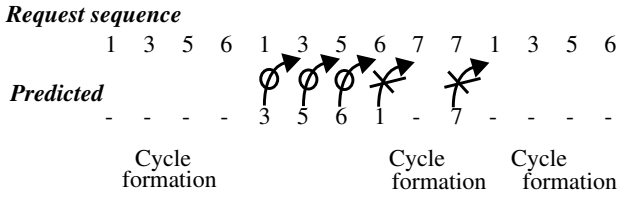


Figure 6. Effects of the Tag predictor on the applications

6.2 The Single-cycle Predictor

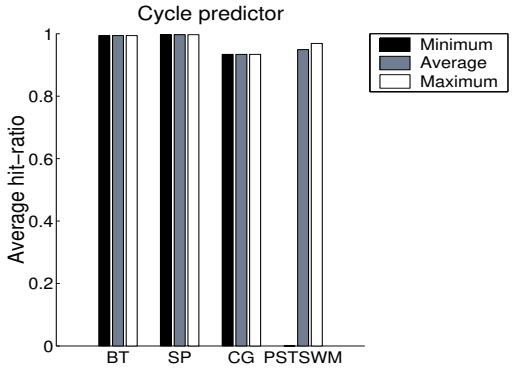
The *Single-cycle* predictor is based on the fact that if a group of receive calls are issued repeatedly in a cyclical fashion, then we can predict the next request one step ahead. The following example illustrates the single-cycle predictor. The top trace represents the sequence of requested receive calls, while the bottom trace represents the predicted sequence. The arrows with the cross represent misses, while the ones with the circle represent hits. The "dash" in place of a predicted request indicates that a cycle is being formed, and therefore no prediction is offered (note that this is also added to the misses).



This predictor implements a simple cycle discovery algorithm. Starting with a *cycle-head* receive call (this is the first receive call that is requested at start-up, or the receive call that causes a miss), we log the sequence of requests until the cycle-head receive call is requested again. This stored sequence constitutes a cycle, and can be used to predict the subsequent requests. If the predicted receive call coincides with the subsequent requested one, then we record a hit. If the requested receive call does not coincide with the predicted one, then we record a miss and the cycle formation stage commences with the cycle-head being the receive call that caused the miss. The performance of the Single-cycle predictor is shown in Figure 7. It is evident that its performance is consistently very high (hit ratios of more than 0.9).

6.3 The Tag-cycle Predictor

The Tag predictor didn't have a good performance on the applications while the Single-cycle predictor had a very good performance. We would like to see the impact of the cycle algorithm on the Tag predictor. Therefore, we combine the Tag algorithm with the Single-cycle algorithm and call it the *Tag-cycle* predictor.



N = 64 for CG, and 49 for others

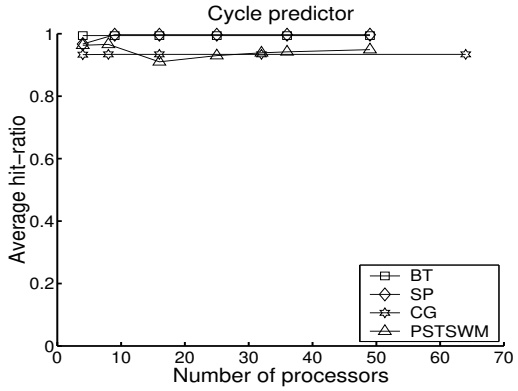
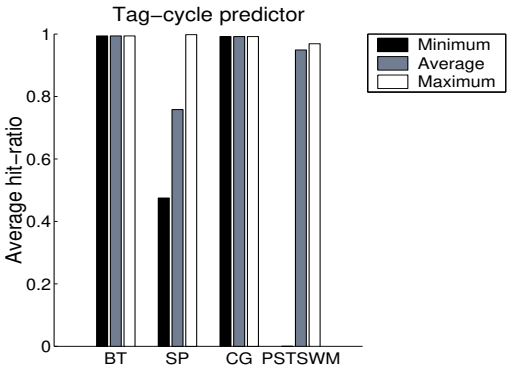


Figure 7. Effects of the Single-cycle predictor on the applications

In the Tag-cycle predictor, we attach a different tag to each of the communication requests found in the benchmarks and do a Single-cycle discovery algorithm on each tag. To this tag and at the communication assist, we assign the requested receive call, to be called *tagcycle-head* node (this is the first receive call that is requested at this tag, or the node that causes a miss). We log the sequence of the requests at this tag until the tagcycle-head node is requested again. This stored sequence constitutes a cycle at each tag, and can be used to predict the subsequent requests. The performance of the Tag-cycle predictor is shown in Figure 8. The Tag-cycle predictor performs well on all benchmarks. Its performance is the same as the Single-cycle predictor on BT and PSTSWM. However, it has a better performance on CG and a lower performance on SP.



N = 64 for CG, and 49 for others

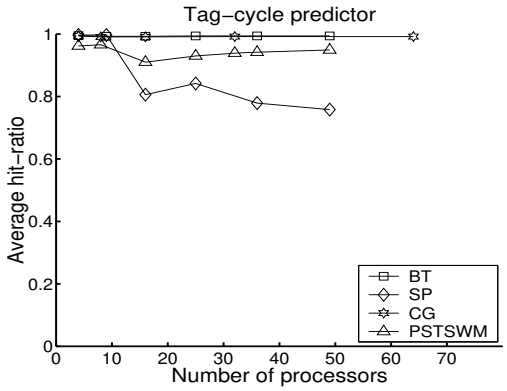
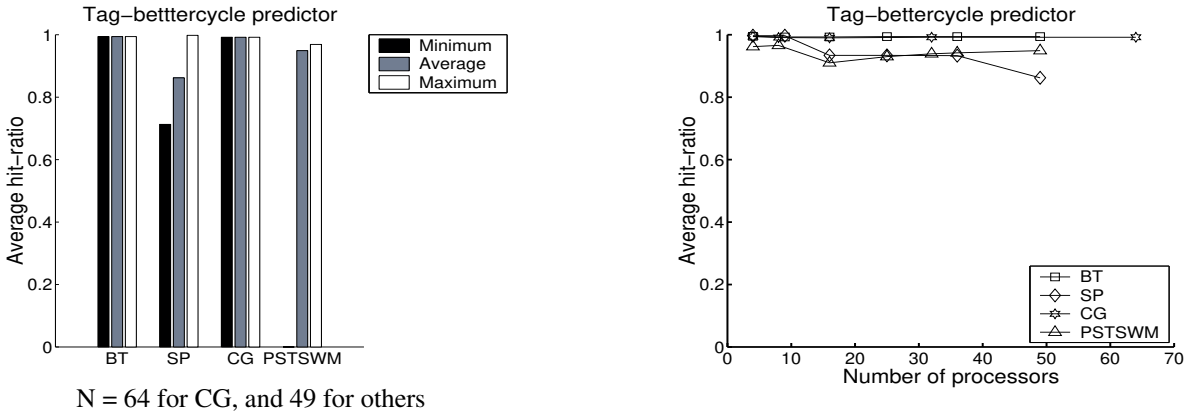


Figure 8. Effects of the Tag-cycle predictor on the applications

6.4 The Tag-bettercycle Predictor

In the Single-cycle and Tag-cycle predictors, as soon as a receive call breaks a cycle we remove the cycle and form a new cycle. In the *Tag-bettercycle* predictor, we keep the last cycle associated with each tagcycle-head encountered in the communication patterns of each node. This means that when a cycle breaks we maintain this cycle in memory for later references. If we haven't already seen the new tagcycle-head then we form a new cycle for it, otherwise we predict the next communication call based on the member of the cycle associated with this new tagcycle-head that we have from the past in memory. The performance of the Tag-bettercycle predictor is shown in Figure 9.

The Tag-bettercycle predictor performs well on all benchmarks. Its performance is the same as the Single-cycle and Tag-cycle predictors on BT and PSTSWM. However, it has a better performance on CG and a lower performance on SP relative to the Single-cycle predictor. The Tag-bettercycle predictor has a better performance on SP compared to the Tag-cycle predictor.

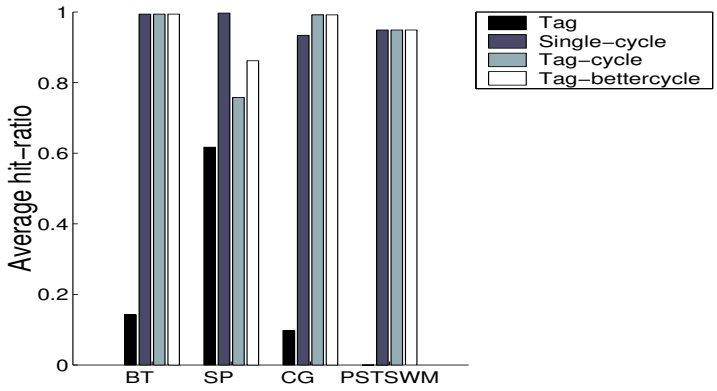


N = 64 for CG, and 49 for others

Figure 9. Effects of the Tag-bettercycle predictor on the applications

6.5 Message Predictors' Comparison

Figure 10, presents a comparison of the performance of the predictors presented in this paper when the number of processors is 64 for CG and 49 for the other benchmarks. As we have seen so far, Single-cycle, Tag-cycle and Tag-bettercycle all perform well on the benchmarks. However, the performance of the Single-cycle is better on the SP benchmark while Tag-cycle and Tag-bettercycle have better performance for the CG benchmark. Similar results have been found for the other system sizes.



N = 64 for CG, and 49 for others

Figure 10. Comparison of the performance of the predictors on the applications

7.0 Conclusion

Communication latency adversely affects the performance of networks of workstations. A significant portion of the software communication overhead belongs to a number of message copying operations. Ideally, it is very desirable to have a true zero-copy protocol where the message is moved directly from the send buffer in its user space to the receive buffer in the destination without any inter-

mediate buffering. However, this is not always possible as a message may arrive at the destination where the corresponding receive call has not been issued yet. Hence, the message has to be buffered in a temporary buffer.

In this paper, we have shown that there is a message reception communication locality in message-passing applications. We have utilized this communication locality and devised different message predictors for the receiver sides of communications. By predicting receive calls early, a node can perform the necessary data placement upon message reception and move the message directly into the cache. We presented the performance of these predictors on some parallel applications. The performance results are quite promising and justify more work in this area.

We envision these predictors to be used to drain the network and place the incoming messages in the cache in such a way so as to increase the probability that the messages will still be in cache when the consuming thread needs to access them.

Further issues we are presently investigating include mechanisms for in-the-cache late binding and thread scheduling to guarantee that the consuming thread finds the message in the cache of the processor it executes on. We shall report on these issues in the future.

Acknowledgments

This work was supported by grants from NSERC and the University of Victoria. We would like to thank Dr. Murray Campbell at the IBM T J Watson Research Center for his kind help in accessing the IBM Deep Blue machine. We also would like to thank the anonymous referees for their valuable comments and suggestions.

References

[1] A. Afsahi and N. J. Dimopoulos, "Hiding Communication Latency in Reconfigurable Message-Passing Environments", *Proceedings of the of IPPS/SPDP 1999, 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, April 1999, pp. 55-60.

[2] A. Afsahi and N. J. Dimopoulos, "Communication Latency Hiding in Reconfigurable Message-Passing Environments: Quantitative Studies", *13th Annual International Symposium on High Performance Computing Systems and Application*s, *HPCS'99*, June, 1999, pp. 111-126.

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstation", IEEE Computer, Volume 29, no. 2, February 1996, pp. 18-28.

[4] D. H. Bailey, T. Harsis, W. Saphir, R. V. der Wijngaart, A. Woo and M. Yarrow, "The NAS Parallel Benchmarks 2.0: Report NAS-95-020", Nasa Ames Research Center, December 1995.

[5] M. Banikazemi, R. K. Govindaraju, R. Blackmore and D. K. Panda, "Implementing Efficient MPI on LAPI for IBM RS/6000 SP Systems: Experiences and Performance Evaluation, "*Proceedings of the of IPPS/SPDP 1999, 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, April 1999, pp. 183-190.

[6] A. Basu, M. Welsh, T. V. Eicken, "Incorporating Memory Management into User-Level Network Interfaces", *Hot Interconnects V*, August 1997.

[7] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg, "A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer", *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, pp. 142-153.

[8] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic and W-K.

Su, "Myrinet: A Gigabit-per-Second Local Area Network", *IEEE Micro*, February 1995.

[9] S. Chodnekar, V. Srinivasan, A. Vaidya, A. Sivasubramaniam and C. Das, "Towards a Communication Characterization Methodology for Parallel Applications", Proceedings of the Third International Symposium on High Performance Computer Architecture, 1997.

[10] H. Chu, "Zero-copy TCP in Solaris, "*Proceedings of the USENIX Annual Technical Conference*, 1996, pp. 253-263.

[11] F. Dahlgren, M. Dubois and P. Stenström, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, 6(7), 1995.

[12] E. D. Demaine, "A Threads-Only MPI Implementation for the Development of Parallel Programs", *Proceedings of the 11th International Symposium on High Performance Computing Systems, HPCS'97*, 1997, pp. 153-163.

[13] B. V. Dao, Sudhakar Yalamanchili, and Jose Duato, "Architectural Support for Reducing Communication Overhead in Multiprocessor Interconnection Networks", *Proceedings of the Third International Symposium on High Performance Computer Architecture*, 1997, pp. 343-352.

[14] J. J. Dongarra and T. Dunigan, "Message-Passing Performance of Various Computers", *Concurrency: Practice and Experience*, Volume 9, Issue 10, 1997, pp. 915-926.

[15] P. Druschel and L. L. Peterson, "Fbufs: A High-bandwidth Cross-domain Transfer Facility", *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles,* 1993, pp. 189-202.

[16] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis and K. Li, "VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication", *Proceedings of the Hot Interconnect'97*, 1997.

[17] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke and C. Dodd, "The Virtual Interface Architecture", *IEEE Micro*, March-April, 1998, pp. 66-76.

[18] R. W. Horst and D. Garcia, "ServerNet SAN I/O Architecture", *Proceedings of the Hot Interconnects V*, 1997.

[19] S. Karlson and M. Brorsson, "A Comparative Characterization of Communication Patterns in Applications Using MPI and Shared Memory on an IBM SP2", *Proceedings of the Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, International Symposium on High Performance Computer Architecture*, February 1998.

[20] S. Kaxiras and J. R. Goodman, "Improving CC-NUMA Performance Using Instruction-Based Prediction", *International Symposium on High Performance Computer Architecture,* 1999.

[21] J. Kim and D. J. Lilja, "Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs", *Proceedings of the Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, International Symposium on High Performance Computer Architecture*, February 1998, pp. 202-216.

[22] D. G. de Lahaut and C. Germain, "Static Communications in Parallel Scientific Programs", *Proceedings of PARLE'94, Parallel Architecture and Languages*, July 1994.

[23] A.-C. Lai and B. Falsafi, "Memory Sharing Predictor: The Key to a Speculative Coherent DSM", *Proceedings of the 26th Annual International Symposium on Computer Architectures*, 1999, pp. 172-183.

[24] M. Lauria, S. Pakin and A. A. Chien, "Efficient Layering for High Speed Communication: Fast Messages 2.x", *Proceedings of the 7th High Performance Distributed Computing, HPDC7, Conference*, 1998.

[25] *Message Passing Interface Forum: MPI: A Message-Passing Interface Standard*. Version 1.1

(June 1995).

[26] *Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface*, (July 1997).

[27] S. S. Mukherjee and M. D. Hill, "Using Prediction to Accelerate Coherence Protocols", *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[28] S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstation: Illinois Fast Messages (FM) for Myrinet," *Proceedings of the Supercomputing'95*, Nov., 1995.

[29] L. Prylli and B. Tourancheau, "BIP: A New Protocol Designed for High Performance Networking on Myrinet", *Proceedings of the PC-NOW98: International Workshop on Personal Computer based Networks Of Workstations, in conjunction with IPPS/SPDP'98*, 1998.

[30] S. H. Rodrigues, T. E. Anderson and D. E. Culler, "High-Performance Local Area Communication with Fast Sockets", *USENIX 1997 Annual Technical Conference*, January 1997.

[31] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles, "Predicting Multiprocessor Memory Access Patterns with Learning Models", *Proceedings of the Fourteenth International Conference on Machine Learning*," 1997, pp. 305-312.

[32] G. Shah, J. Nieplocha, J. Mirza and C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, "Performance and Experience with LAPI -- a New High-Performance Communication Library for the IBM RS/6000 SP", *First Merged Symposium IPPS/ SPDP 1998 12th International Parallel Processing symposium & 9th Symposium on Parallel and Distributed Processing,* 1998.

[33] T. Takahashi, F. O'Carrol, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, P.H. Beckman, "Implementation and Evaluation of MPI on an SMP Cluster", *Proceedings of the PC-NOW99: International Workshop on Personal Computer based Networks Of Workstations, in conjunction with PPS/SPDP'99*, 1999.

[34] Y. Tanaka, M. Matsuda, M. Ando, K. Kubota and M. Sato, "COMPaS: A Pentium Pro PC-based SMP Cluster and its Experience", *Proceedings of the PC-NOW98: International Workshop on Personal Computer based Networks Of Workstations, in conjunction with IPPS/ SPDP'98*, 1998.

[35] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication", *First Merged Symposium IPPS/SPDP 1998 12th International Parallel Processing symposium & 9th Symposium on Parallel and Distributed Processing,* 1998.

[36] T. V. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation", *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 256-265.

[37] T. V. Eicken, A. Basu, V. Buch and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", Proceedings of the 15th ACM Symposium on Operating Systems Principles, December, 1995.

[38] P. H. Worley and I. T. Foster, "Parallel Spectral Transform Shallow Water Model: A Runtime-tunable parallel benchmark code", *Proceedings of the Scalable High Performance Computing Conference*, 1994, pp. 207-214.

[39] Z. Zhang and J. Torrellas, "Speeding Up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching", *Proceedings of the 22nd Annual International Symposium on Computer Architectures*, 1995, pp. 188-199.