CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2002; **14**:859–883 (DOI: 10.1002/cpe.697)

Efficient communication using message prediction for clusters of multiprocessors

Ahmad Afsahi[1,*,†] and Nikitas J. Dimopoulos[2]

[1]*Department of Electrical and Computer Engineering, Queen's University, Kingston, Canada K7L 3N6*
[2]*Department of Electrical and Computer Engineering, University of Victoria, P.O. Box 3055, Victoria, Canada V8W 3P6*

SUMMARY

With the increasing uniprocessor and symmetric multiprocessor computational power available today, interprocessor communication has become an important factor that limits the performance of clusters of workstations/multiprocessors. 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 operations. 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 yet been posted. 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. We also show that the proposed predictors do not have sensitivity to the starting message reception call, and that they perform better than (or at least equal to) our previously proposed predictors. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: message prediction; clusters; communication locality; message passing interface (MPI); zero-copy

1. INTRODUCTION

With the increasing uniprocessor and *symmetric multiprocessor* (SMP) computational power available today, interprocessor communication has become an important factor that limits the

*Correspondence to: Ahmad Afsahi, Department of Electrical and Computer Engineering, Queen's University, Kingston, Canada K7L 3N6.
†E-mail: ahmad@ee.queensu.ca

Received 14 September 2000
Revised 9 November 2001

performance of workstation 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 clusters of workstations. Even with high-performance networks [1,2] available today, there is still a gap between what the network can offer and what the user application 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 operations.

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) [3], Fast Messages (FM) [4], Virtual Memory-Mapped Communications (VMMC-2) [5], U-Net [6], Virtual Interface Architecture (VIA) [7], and PM [8]. A significant portion of the software communication overhead belongs to a number of message copying operations. 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, applications at the send side do 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 [4,5,9–11]. While they have been able to remove the memory copying between the application buffer space and the network interface at the send side by using user-level messaging techniques, they have not been able to remove the memory copying at the receiver sides completely. They may achieve zero-copy messaging at the receiver sides only if the receive call is already posted, a rendezvous-type communication is used for large messages, or the destination buffer address is already known by a pre-communication.

We are interested in bypassing the memory copying at the destination in the general case, synchronous or asynchronous, eager or rendezvous and for sender-initiated communications as in MPI [12]. 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 introduces the notion of message prediction for the receiving side of message-passing systems [13,14]. 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. As a matter of fact, communication operations at both the sender and the receiver sides may be issued at the earliest possible time, while the predictive algorithms will ensure that they will be scheduled and the data made available at the optimal place and time. We are interested in utilizing similar predictors as in [15], but this time at the receiver sides to predict the next consumable message and drain the network as soon as the message arrives.

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 [16] in isolation. Our tools are not yet ready to measure the effectiveness of our predictors on the application run-time. 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 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. Our experimental methodologies to gather communication traces of our parallel applications are explained in Section 4. In Section 5, 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, we propose our message predictors and present their performance on the applications. In Section 7, we briefly discuss the integration of the predictors with the network interface. Finally, we conclude our paper in Section 8.

2. MOTIVATION AND RELATED WORK

High-performance computing is increasingly concerned with efficient communication across the interconnect due to the availability of high-speed advanced processors. Modern networks such as Myrinet [1] and Gigabit Ethernet [2], 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 [3–8]. In 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 operations, control transfer mechanisms, address translation mechanisms, protection mechanisms, and reliability issues are the key factors for the performance of a user-level communication system. A significant portion of the software communication overhead belongs to a number of message copying operations. 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 (1) from the send buffer to the system buffer, (2) from the system buffer to the *network interface* (NI), (3) at the other end of the communication from the network interface to the system buffer, and (4) from the system buffer to the receive buffer when the receive call is posted. Note that we have not considered data transfer from the network interface at the sending process to the network interface at the receiving process as a separate copy. In this paper, we are particularly interested in avoiding message copying operations at the receiver sides of communications.

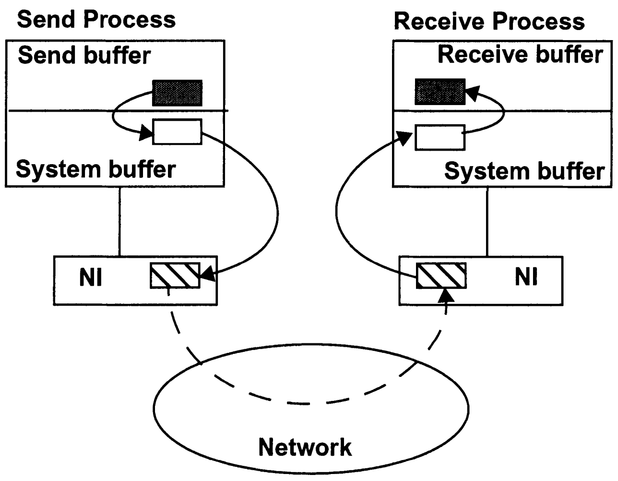

Figure 1. Data transfers in a traditional messaging layer.

2.1. Avoiding extra message copying operations

In the following sections, we mention a number of research works that have attempted to bypass the system buffer copying at the send and receive sides of communications.

2.1.1. Using programmed I/O and DMA at the send side

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 (Basic Interface for Parallelism) 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 pin and unpin pages in the user space that contain the send and the receive buffers.

2.1.2. Using redirection mechanisms at the receiver side

Contrary to the send side, bypassing the system buffer copying at the receiving side may not be achievable. Processes at the sending side do not know the receive buffer addresses, however they do know the receive processes. Therefore, when a message arrives at the receiving side it has to be buffered if the receive call has not been posted yet. VMMC [17] for the SHRIMP multicomputer used 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. This leads to a two-phase rendezvous protocol which adds to the network traffic and network latency, especially for short messages.

VMMC-2 [5] 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 [9] 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 [4] uses a similar approach to 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 [10] is an implementation of MPI on top of LAPI 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 does not find a matching receive, it will return the address of an *early arrival buffer* and hence a one-copy transfer is accomplished. Meanwhile, messages with larger sizes than the eager size are transferred using a two-phase rendezvous protocol.

Some research projects have proposed solutions to multi-protocol message-passing interfaces on *clusters of multiprocessors* (Clumps) using both shared-memory for intra-node communications and message-passing for inter-node communications [11,18]. MPICH-PM/CLUMP [11] 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 inter-node communications, it uses *eager* and *rendezvous* protocols. For short messages, it achieves one-copy using the eager protocol, as the message is copied into a temporary buffer if the MPI receive primitive has not been issued. For large messages, it uses the rendezvous protocol to achieve zero-copy by using a remote write operation but it needs an extra communication. For intra-node communications, it achieves one-copy using a kernel primitive that allows one to copy messages from the sender to the receiver without the involvement of a communication buffer.

BIP-SMP [18], for intra-node communications, uses shared memory for small messages with two memory copy, and direct copy for large messages with a kernel overhead. For inter-node communications, it works like MPI-BIP with one memory copy.

2.1.3. Re-mapping and copy-on-write techniques

Other techniques to bypass extra copying are the *re-mapping* and *copy-on-write* techniques [19,20]. Both techniques require one to switch to the supervisor mode, acquire necessary locks to the virtual memory data structure, change the virtual memory mapping at several levels for each page, then perform *Translation Lookaside Buffer* (TLB)/cache consistency actions, and finally return to the user mode. This limits the performance of the page re-mapping and copy-on-write techniques. A zero-copy TCP stack is implemented in Solaris by using copy-on-write pages and re-mapping to improve communication performance [19]. 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 [20] also uses the re-mapping technique to avoid the penalty of copying large messages across different layers of the protocol stack. However, fbufs allows re-mapping only for a limited range of user virtual memory.

As stated above, the user-level messaging techniques may not achieve 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.

2.1.4. Compiler techniques

There are numerous works at the compiler level, such as the works in [21,22], which are directly related to enhancing the communication performance. These works achieve this by moving non-blocking sends and receives as far up in the code as possible and move blocking waits as far down in the code as possible. This way, the chances that the corresponding receive has been invoked before the message arrives at the receiver are increased.

2.2. Prediction techniques

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 [16,23–26]. These techniques assume that memory accesses and coherence activities in the near future will follow past patterns. In [26], the authors proposed hardware regular stride techniques to prefetch several blocks ahead of the current data block. More elaborate hardware-based irregular stride prefetching approaches have been proposed in [25]. Kaxiras and Goodman [24] 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. This is in contrast to address-based techniques that keep data-access history for the predictions. In [16], the authors devised a general pattern-based predictor, *cosmos*, to learn and predict the coherence activity for a memory block in a DSM. Cosmos makes a prediction in two steps. First, it uses a cache block address to index into a message history table to obtain the <processor and message-type> tuples of the last few coherence messages received for that cache block. Then it uses these <processor, message-type> tuples to index a pattern history table to obtain a <processor, message-type> tuple prediction. In [23], the authors 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, those primary messages that invoke a sequence of protocol actions. It improves prediction accuracy over cosmos by eliminating the acknowledgements messages from the pattern tables. It also reduces memory overhead and perturbation in the tables due to message re-ordering.

In software-controlled prefetching, the programmer or compiler decides when and what to prefetch by analyzing the code and inserting *prefetch* instructions. In [27], the authors used software-controlled prefetching and multithreading to hide and reduce the latency in shared memory multiprocessors.

Recently, we proposed some heuristics to predict the destination target of subsequent communication requests at the send side of communications in message-passing systems [15]. 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 operations at the receiving ends of communications for message-passing systems [13,14]. 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 yet been posted.

3. USING MESSAGE PREDICTIONS

In this section, we analyze the problem of 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, then we can move the message upon its arrival to near the place where it is to be consumed (e.g. a staging cache), or we could schedule which thread to execute next, preferably at the same processor as the consuming thread, to enhance the chances that the data will be in the processor cache when it is accessed by the consumer.

To do this, we identify three different issues. First, we need to decide which message is going to be consumed next. This can be done by devising receive call predictors, which are history-based predictors that predict subsequent receive calls by a given process in a message-passing program. Second, we need to decide 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. EXPERIMENTAL METHODOLOGY

In exploring the effect that different heuristics have on 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 [28], and the *Parallel Spectral Transform Shallow Water Model* (PSTSWM) application [29]. We used the MPI [12] implementation of the NPB suite (version 2.3), and version 6.2 of the PSTSWM application. Specifically, we used the block tridiagonal (BT), scalar pentadiagonal (SP), and conjugate gradient (CG) benchmarks from the NPB suite [28]. We did not use the multigrid (MG) and lower-upper diagonal (LU) benchmarks from the NPB suite because these benchmarks use *MPI_ANY_SOURCE* in some of their receive calls (*MPI_Recv* and *MPI_Irecv*).

This means that the applications may receive a particular message from different sources depending on the order of arrival.

We are only interested in the patterns of the point-to-point communications between pairwise processes 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 into 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. Note that the applications use pure message passing for communications between different processes (no shared-memory programming or threading) and the processes can be run on the same or different processors of the IBM SP without any effect on the communication patterns. It is also worth mentioning that the applications have the same communication patterns in other environments such as networks of PCs and thus our work is not bound to any specific system.

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 the 'W' and 'A' classes. Hence, we report only those for the 'A' class here. Although the results presented in this paper are for the above parallel applications, these applications have been widely used as benchmarks representing the computations in scientific and engineering parallel applications.

5. RECEIVER-SIDE LOCALITY ESTIMATION

Our applications use blocking and non-blocking standard MPI receive primitives, namely *MPI_Recv* and *MPI_Irecv* [12]. *MPI_Irecv (buf, count, datatype, source, tag, comm, request)* is a standard non-blocking receive call. It immediately posts the call and returns. Hence, data are 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. *MPI_Recv (buf, count, datatype, source, tag, comm, status)* is a standard blocking receive call. When it returns, data are available at the destination buffer. The PSTSWM application uses this type of receive call.

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 process 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 processes of each application. It is clear that all processes in the BT, SP, and CG applications have the same number of receive communication calls, while processes in the PSTSWM application have different numbers 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 envelope, that is, *source, tag, count, datatype*, *buf*, and *comm* (we do not consider *status* and *request* as they are just a handle when the calls return). We did not rely only on the buffer address,

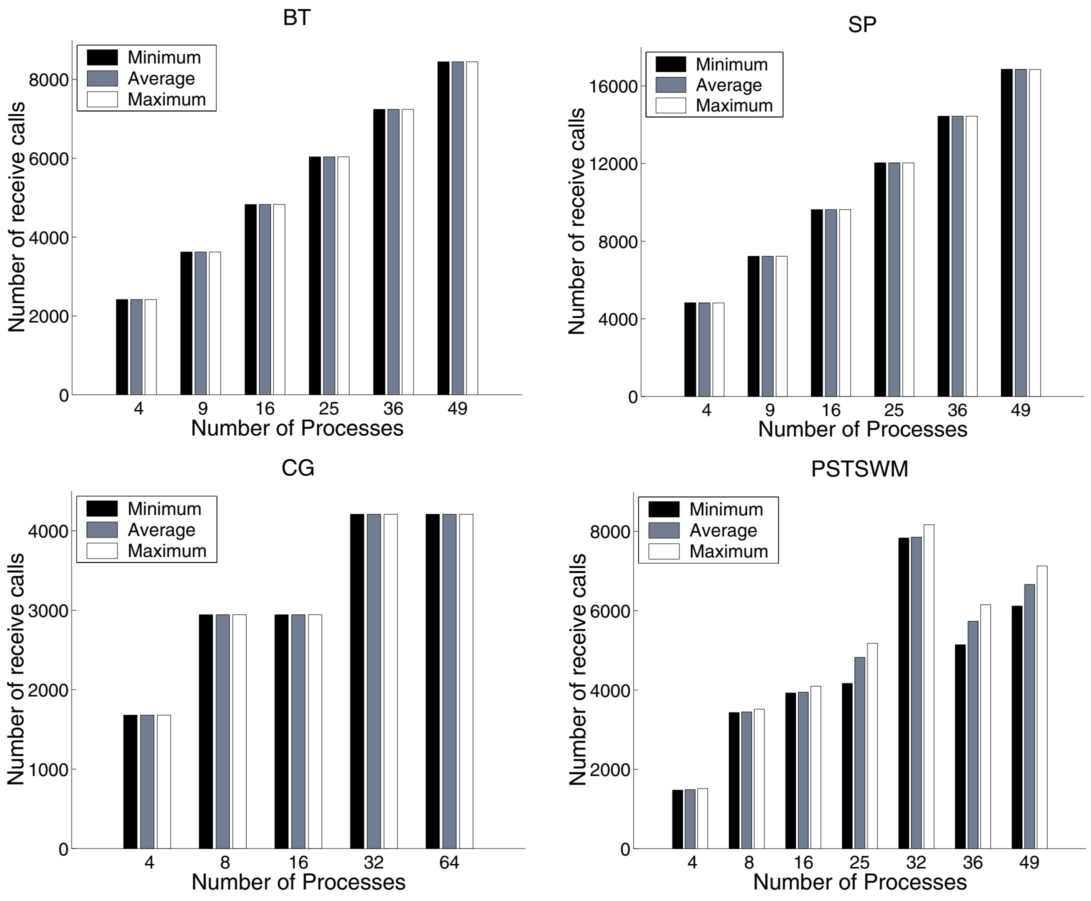

Figure 2. The number of receive calls per process in the applications under different system sizes.

buf, of a receive call as many processes may send their messages to the same buffer address of a particular destination process. Neither could we depend only on the sender, *source*, of a message, or on the length, *count*, of a message. Therefore, we assigned 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 processes of each application. It is evident that all processes in the BT and CG applications have the same number of unique message identifiers, while processes in the SP and PSTSWM applications have different numbers of unique message identifiers (except when the number of processes is four for the SP benchmark).

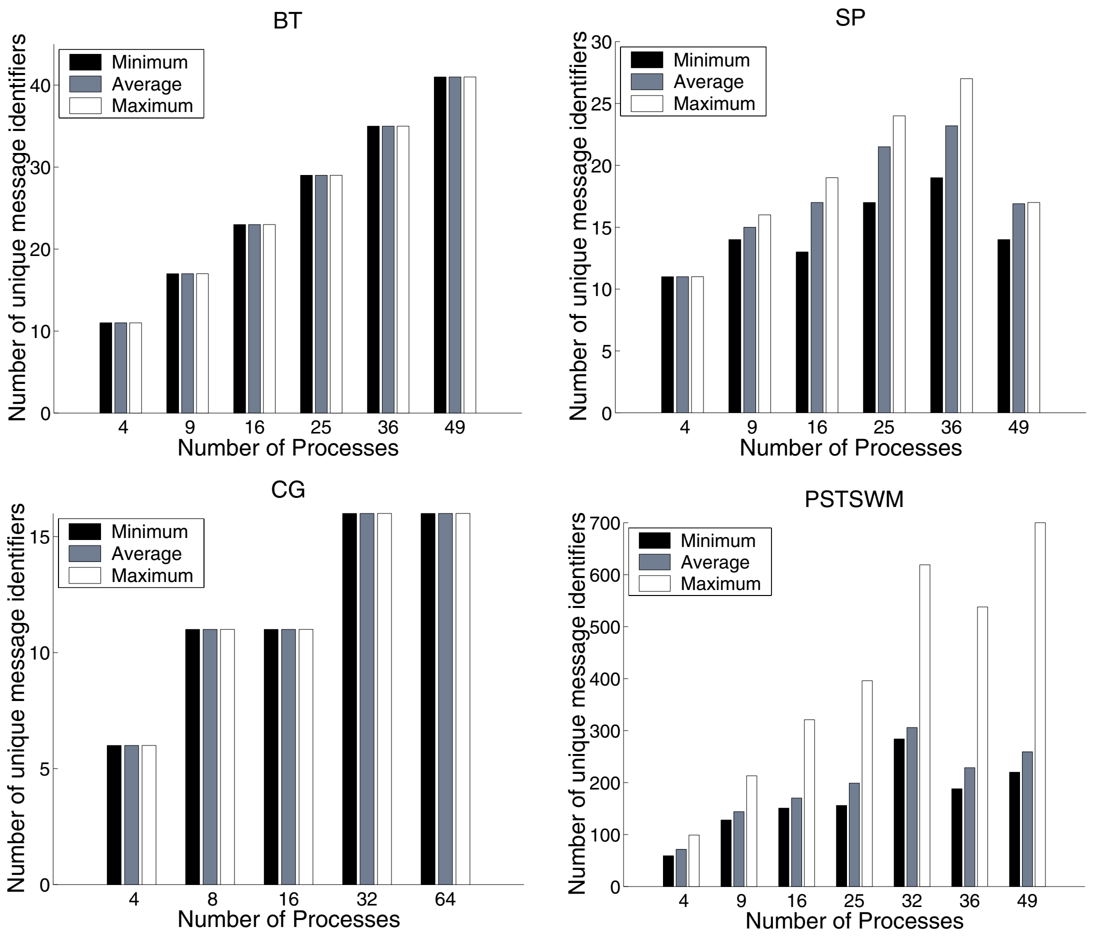

Figure 3. The number of unique message identifiers per process in the applications under different system sizes.

Figure 4 shows the distribution of each unique message identifier for process zero of the applications when the number of processes is 64 for CG and 49 for the other applications. We chose process zero because this process has the largest number of unique message identifiers among all processes, and is also responsible for distributing data and verifying the results of the computation. As is shown in Figure 3, the message identifiers are evenly distributed in BT. However, the distributions 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.

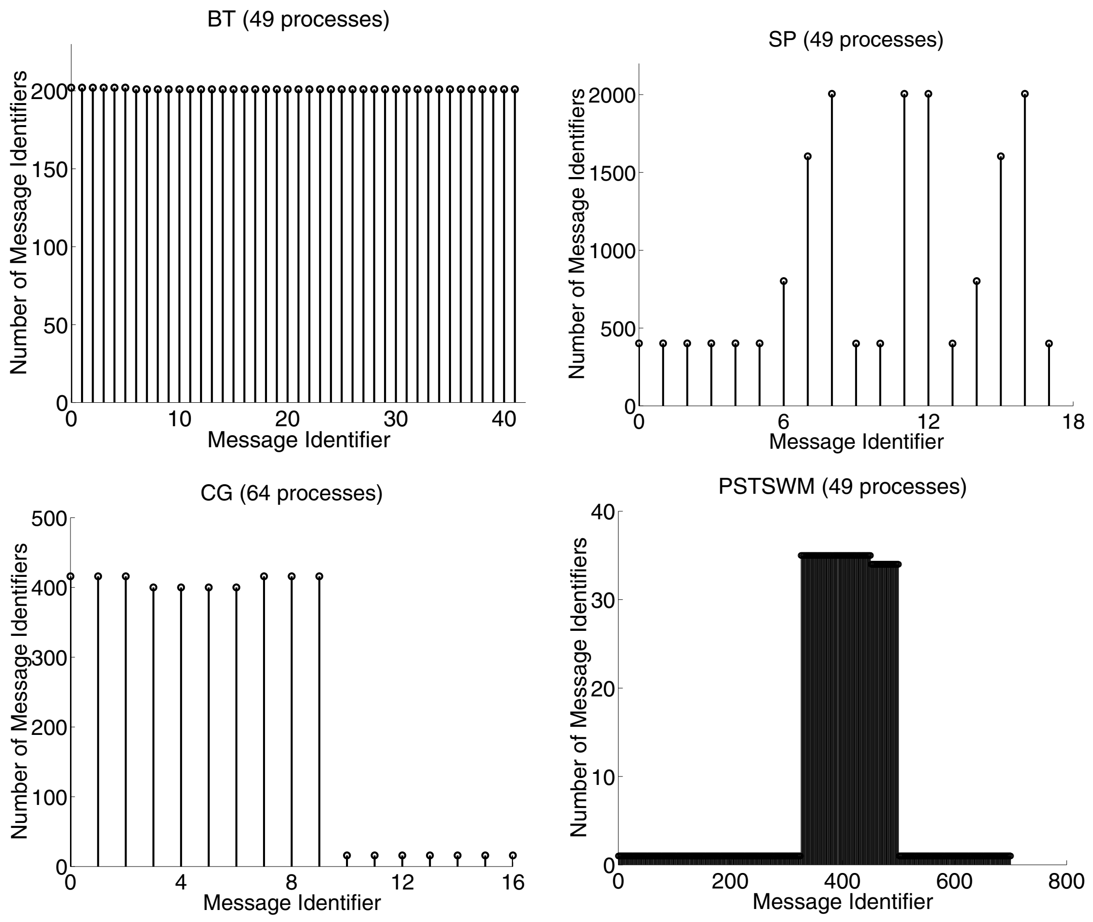

Figure 4. The distribution of the unique message identifiers for process zero in the applications.

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 [15,30–32]. Kim and Lilja [30] 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 an earlier work [15], we have shown the communication locality of message-passing applications in terms of message destination locality.

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 section, 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 the 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 average hit ratios of the LRU, FIFO, and LFU heuristics on the application benchmarks when the number of processes is 64 for CG and 49 for all other applications (taken over all processes of the applications). The plots start at the window size of one and stop at the window size equal to the maximum number of message identifiers for each application. It is clear that the hit ratios in all benchmarks approach one as the window size increases. The LRU and FIFO heuristics have a zero hit ratio on the BT, CG, and PSTSWM applications up to a certain window size. The reason for this is that these applications, after their initialization phase, cycle through a number of message reception calls and then enter their ending phase (however, the SP application does not follow this). The LRU and FIFO heuristics have a zero hit ratio if the window size is less than the cycle length. The performance of the FIFO algorithm is the same as that of the LRU for the BT and PSTSWM benchmarks, and almost the same as that of 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. It is interesting to see that a real application like PSTSWM needs window sizes of greater than 150 to achieve a good performance (hit ratios above 80%) under the LFU policy. Similar performance results for the LRU, FIFO, and LFU heuristics on other system sizes can be found in [33].

Essentially, the LRU, FIFO and LFU heuristics do not predict the next receive call exactly, but show the probability that the next receive call might be in the set. For instance, the SP benchmark shows a nearly 60% hit ratio for a window size of five under the LRU heuristic. This means that for 60% of the time one of the five most recently issued calls 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 needs 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 in having predictors that can predict the next receive call with a high probability. In Section 6, we introduce our novel message predictors which employ different heuristics and evaluate their performance.

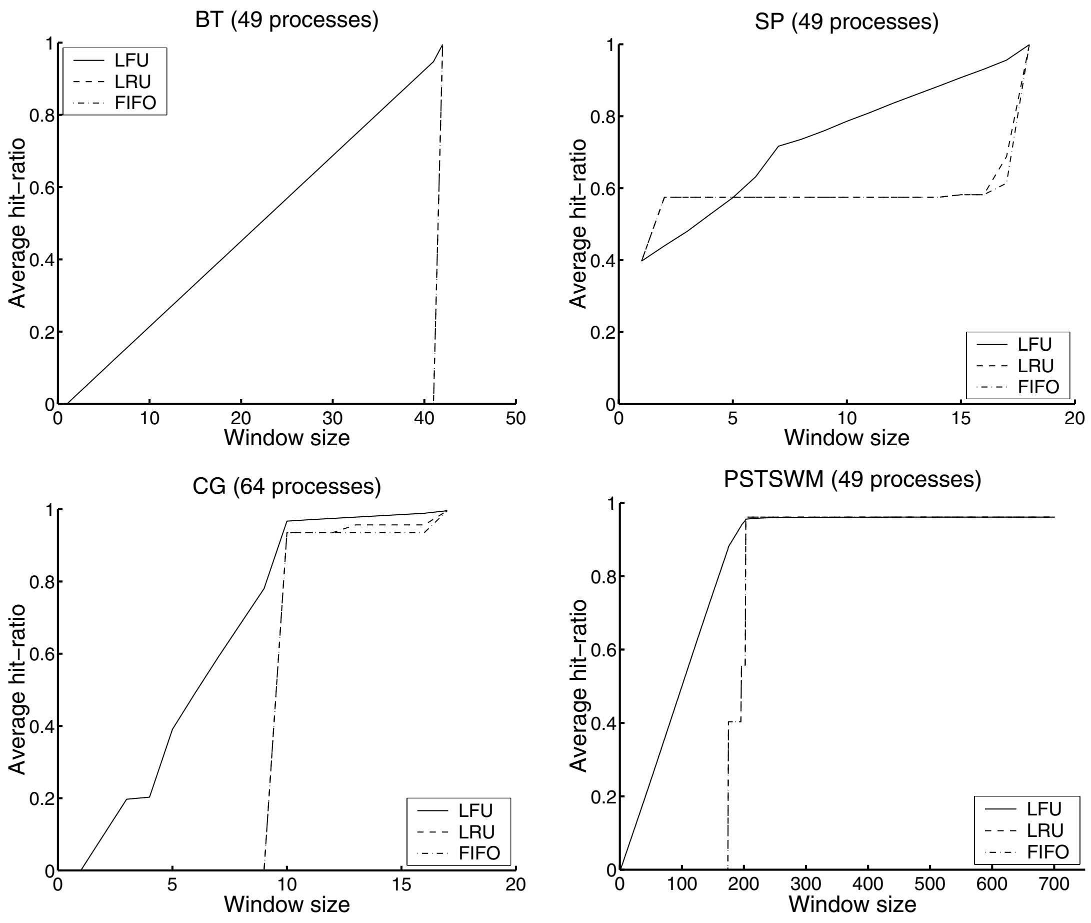

Figure 5. The effects of the LRU, FIFO, and LFU heuristics on the applications.

6. MESSAGE PREDICTORS

The set of predictors proposed in this section predicts the subsequent receive calls based on the past history of communication patterns on a per process basis. We propose two types of predictors in this work: the *Single-cycle* predictor, which is purely a dynamic predictor, and *Tag-based* predictors, which are static/dynamic predictors. In the Single-cycle predictor, predictions are done dynamically at the network interface without any help from the programmer or compiler. In the Tag-based predictors, *Tagging*, *Tag-cycle*, and *Tag-bettercycle*, predictions are done dynamically at the network interface as well, but they require some information to be passed from the program to the network interface. This can be done with the help of the programmer and/or the compiler through inserting instructions

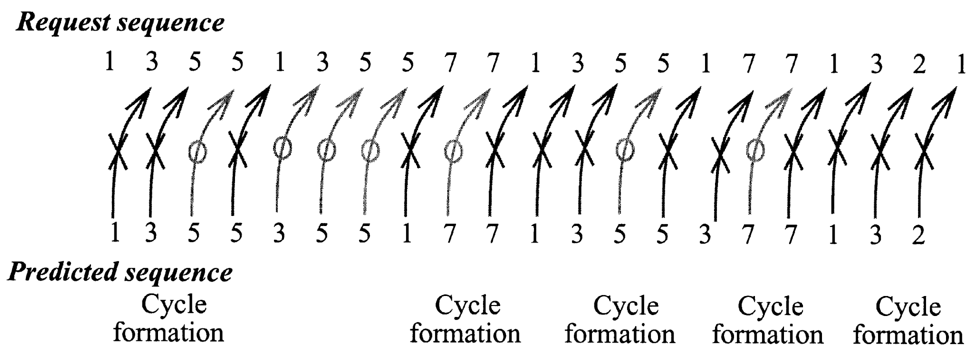

Figure 6. The operation of the Single-cycle predictor on the sample request sequence.

such as *pre-receive (tag)* in the program. The Tag-based predictors can be pure dynamic predictors if another level of prediction is done on the tags themselves at the network interface. In this way, there is no need for the program to pass pre-connect (tag) information to the network interface. We leave this approach for future research. These heuristics were originally proposed in [15] to predict the destination target of subsequent communication requests at the send side of communications. However, in this paper, we have added an 'initialization phase' to the *Single-cycle*, *Tag-cycle*, and *Tag-bettercycle* predictors to remove the initialization patterns of communications, if any [14]. We explain this initialization phase in Section 6.1.1.

It is worth mentioning that the message re-ordering effect [23] (messages from different processes may arrive out-of-order even if messages from the same processes arrive in order in most networks) has no effect on the predictions as the predictors predict the next receive calls based on the patterns of the receive calls in the program that runs on the same process and not on the arriving messages, unless the order of receive calls depends on the order of message arrival. Note that in the following figures, by average, minimum, and maximum, we mean the average, minimum, and maximum hit ratios taken over all processes of each application.

6.1. The single-cycle predictor

The Single-cycle predictor consists of an 'initialization phase' followed by a 'prediction phase'. The 'prediction phase' 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. Figure 6 illustrates an example for the operation of 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 those 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).

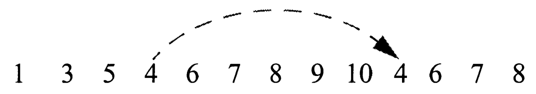

Figure 7. The message reception call sequence (from left to right).

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. Note that, during cycle formation, the previously requested message destination is offered as the predicted message destination. 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.

6.1.1. Initialization phase

We have added an initialization phase to the Single-cycle, Tag-cycle, and Tag-bettercycle predictors proposed in [13,15]. As stated earlier, the prediction phase of the Single-cycle predictor (and all other predictors) starts with the first message reception call as the cycle-head. However, it is possible that we never find this cycle-head again to establish a cycle. Thus, the predictor fails with a zero prediction hit ratio. This may also happen if we skip the first receive call and start forming the cycles with the second receive call as the cycle-head. In other words, applications may have a sequence of receive calls at the beginning that may never come back to them and hence no cycle is formed. This is illustrated in the example in Figure 7. It is clear that no cycle will be formed with the message cycle-heads 1, 3, and 5. However, if we skip ahead we see that a cycle is formed with the message cycle-head 4. This cycle consists of message calls 4, 6, 7, 8, 9, and 10. The first three message reception calls 1, 3, and 5, are considered to be the initialization pattern. The new Single-cycle predictor (along with the Tag-cycle and Tag-bettercycle predictors) proposed in this paper starts with the 'initialization phase', and then after forming the first cycle switches to the 'prediction phase'. It is evident that misses are added until the first cycle is formed. Note that in the 'initialization phase' of the predictors, we looked for a cycle with a length of greater than 5. This was chosen arbitrarily so as not to form a small cycle.

The performance of the Single-cycle predictor is shown in Figure 8. Its performance is consistently very high (hit ratios of more than 0.9).

6.2. The Tagging predictor

The Tagging 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

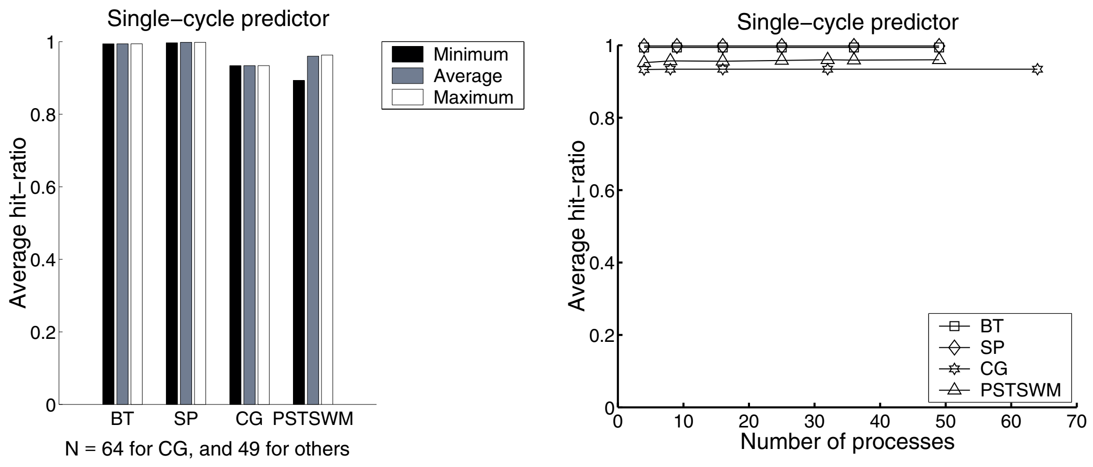

Figure 8. The effects of the Single-cycle predictor on the applications.

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 Tagging predictor is shown in Figure 9. It is evident that this predictor does not 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 processes increases.

6.3. The Tag-cycle predictor

The Tagging predictor did not 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 Tagging predictor. Therefore, we combine the Tagging algorithm with the Single-cycle algorithm and call it the Tag-cycle predictor.

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 the *tagcycle-head* (this is the first receive call that is requested at this tag, or the call that causes a miss). We log the sequence of the requests at this tag until the tagcycle-head 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 10. The Tag-cycle predictor performs well on all benchmarks.

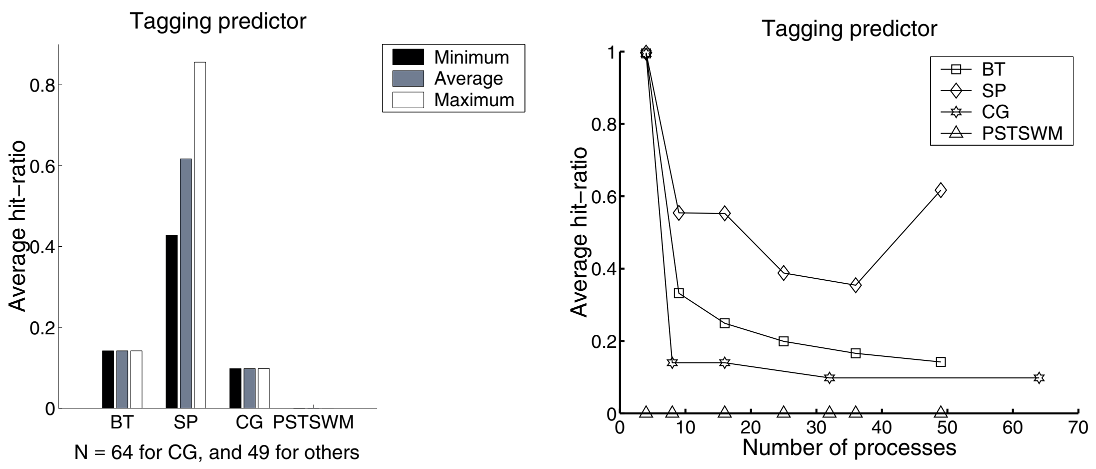

Figure 9. The effects of the Tagging predictor on the applications.

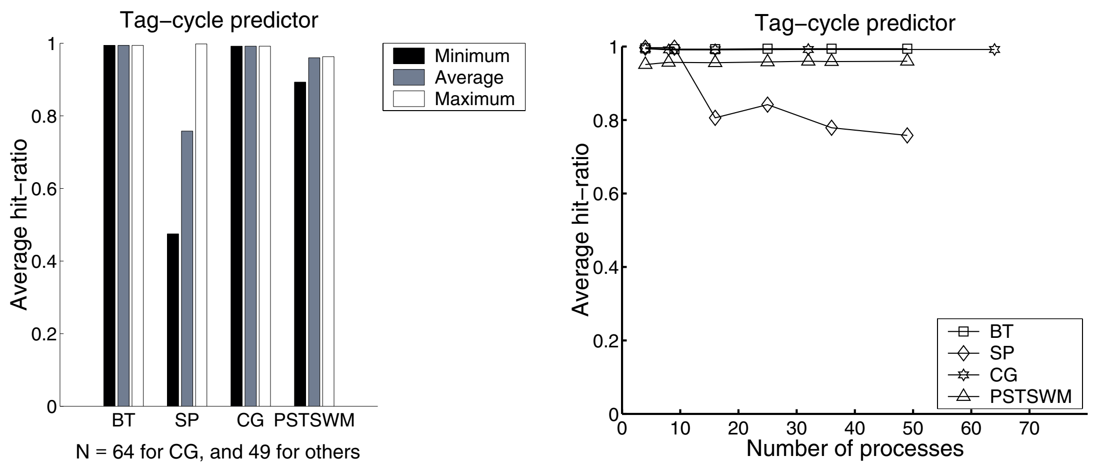

Figure 10. The effects of the Tag-cycle predictor on the applications.

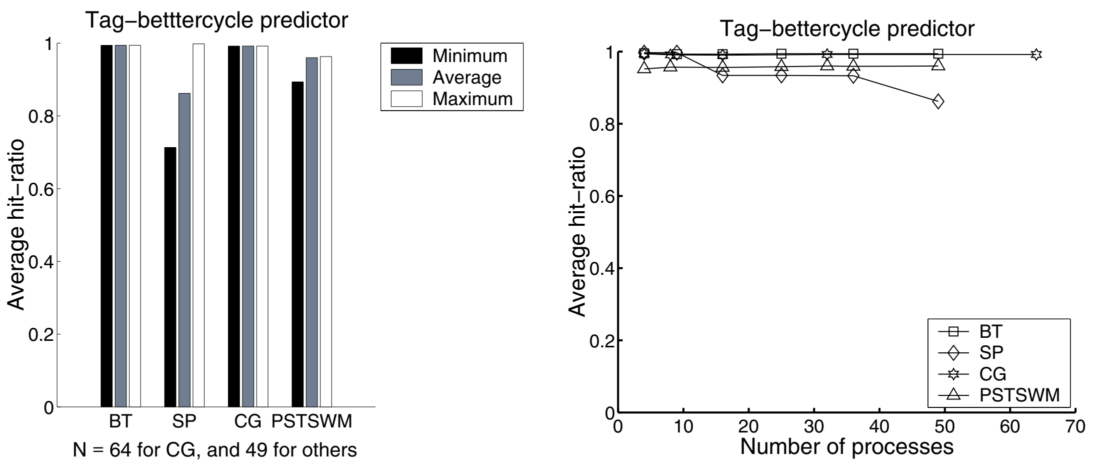

Figure 11. The effects of the Tag-bettercycle predictor on the applications.

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.

6.4. The Tag-bettercycle predictor

In the Tag-cycle predictor, 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 process. This means that when a cycle breaks we maintain this cycle in memory for later references. If we have not 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 11. 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 the 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. We also found that the applications have very small numbers of tagbettercycle-heads (at most two) under the Tag-bettercycle predictor and different system sizes.

6.5. Message predictors' comparison

Figure 12 presents a comparison of the performance of the predictors presented in this paper when the number of processes 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

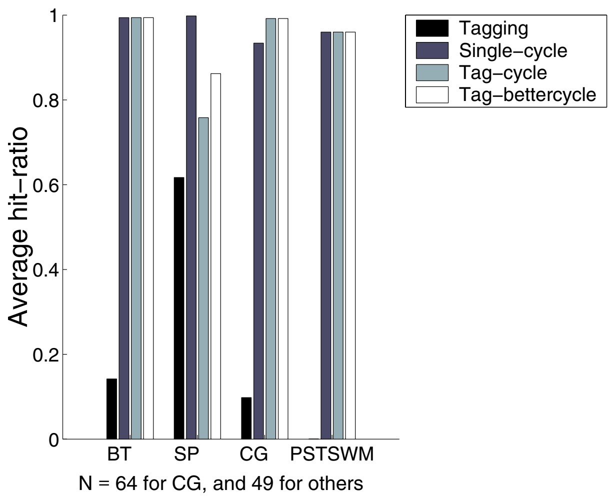

Figure 12. Comparison of the performances of the predictors on the applications.

Table I. Memory requirements (in 6-tuple sets) for the predictors when $N = 64$ for CG, and $N = 49$ for BT, SP, and PSTSWM.

	BT	SP	CG	PSTSWM
Tagging	12	12	10	7
Single-cycle	43	43	138	204
Tag-cycle	60	72	40	693
Tag-bettercycle	60	108	40	693

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 [33].

6.5.1. Predictor's memory requirements

Table I compares the memory requirements of the message predictors on the application benchmarks when the number of processes is 64 for CG and 49 for BT, SP, and PSTSWM. We have found that the memory requirements of the predictors decrease gradually when the number of processes decreases. The numbers in the table are the multiplication factor for the amount of storage needed to maintain the message 6-tuple sets. It is quite clear that the memory requirements of the predictors is low. This makes them very attractive for the implementation at the network interface. Comparatively, predictors (Single-cycle, Tag-cycle, and Tag-bettercycle) need higher memory requirements for the PSTSWM application. Although the classical LRU, LFU, and FIFO heuristics need less memory requirements, but as stated earlier the beauty of the predictors lies in the fact that they predict with high accuracy and transfer only one message to the cache which should dramatically reduce the cache pollution effect, if any. This should also bring down the software cost of the implementation.

The storage requirements of the predictors have been found using the following formulae:

$$\mathit{Mem}_{\text{Single-cycle}} = \text{Maximum cycle length} \tag{1}$$

$$\mathit{Mem}_{\text{Tagging}} = \text{Maximum number of tags} \tag{2}$$

$$\mathit{Mem}_{\text{Tag-cycle}} = \mathit{Mem}_{\text{Tagging}} \times \text{Maximum cycle length of each tag} \tag{3}$$

$$\mathit{Mem}_{\text{Tag-bettercycle}} = \mathit{Mem}_{\text{Tag-cycle}} \times \text{Maximum number of tagbettercycle-heads} \tag{4}$$

6.6. Sensitivity analysis of the message predictors

In this section, we are interested in discovering the prediction sensitivity of the Single-cycle, Tagcycle, and Tag-bettercycle message predictors to the varying starting message reception calls. That is, what would be the difference in the predictor's performance if we begin the predictors with the first message reception call, second message reception call, and so on. In fact, we applied the predictors on the benchmarks starting with each of the first 100 message reception calls. Figure 13 compares the average hit ratios of the proposed predictors in this paper (that is with the initialization phase) over the starting first 100 message reception calls with the predictors without the initialization phase as proposed in [13]. It is clear that the revised predictors proposed in this work perform better than the previously proposed predictors in [13] for the PSTSWM application. However, they have the same average performance on the BT, SP, and CG benchmarks. Meanwhile, the average performances of the predictors, with varying starting message reception calls, are more than 95% on the BT, CG, and PSTSWM benchmarks, and more than 75% on the SP benchmark.

7. INTEGRATING MESSAGE PREDICTORS WITH THE NETWORK INTERFACE

In this section, we briefly discuss how a message predictor can be used and integrated into the network interface. Predictors would reside beside the communication assist or network interface. They monitor the message reception call patterns of their host process and make a prediction according to their prediction algorithms. Then, the network interface uses the predictions to move the early arrival messages into the cache.

As stated above, the predictors would execute on the communication assist of each node of the parallel machine, and predict the message reception calls based on the past history of communications. The Single-cycle predictor does not need any help from the compiler or programmer. However, the Tag-based predictors (Tagging, Tag-cycle, and Tag-bettercycle) require an interface to pass some information from the program to the network interface. With a simple help from the programmer or compiler, this can be done through inserting *pre-receive (tag)* instructions in the program well above each specific receive communication operation, but evidently after the previous receive communication operation. The Tag-based predictors can be pure dynamic predictors if another level of prediction is

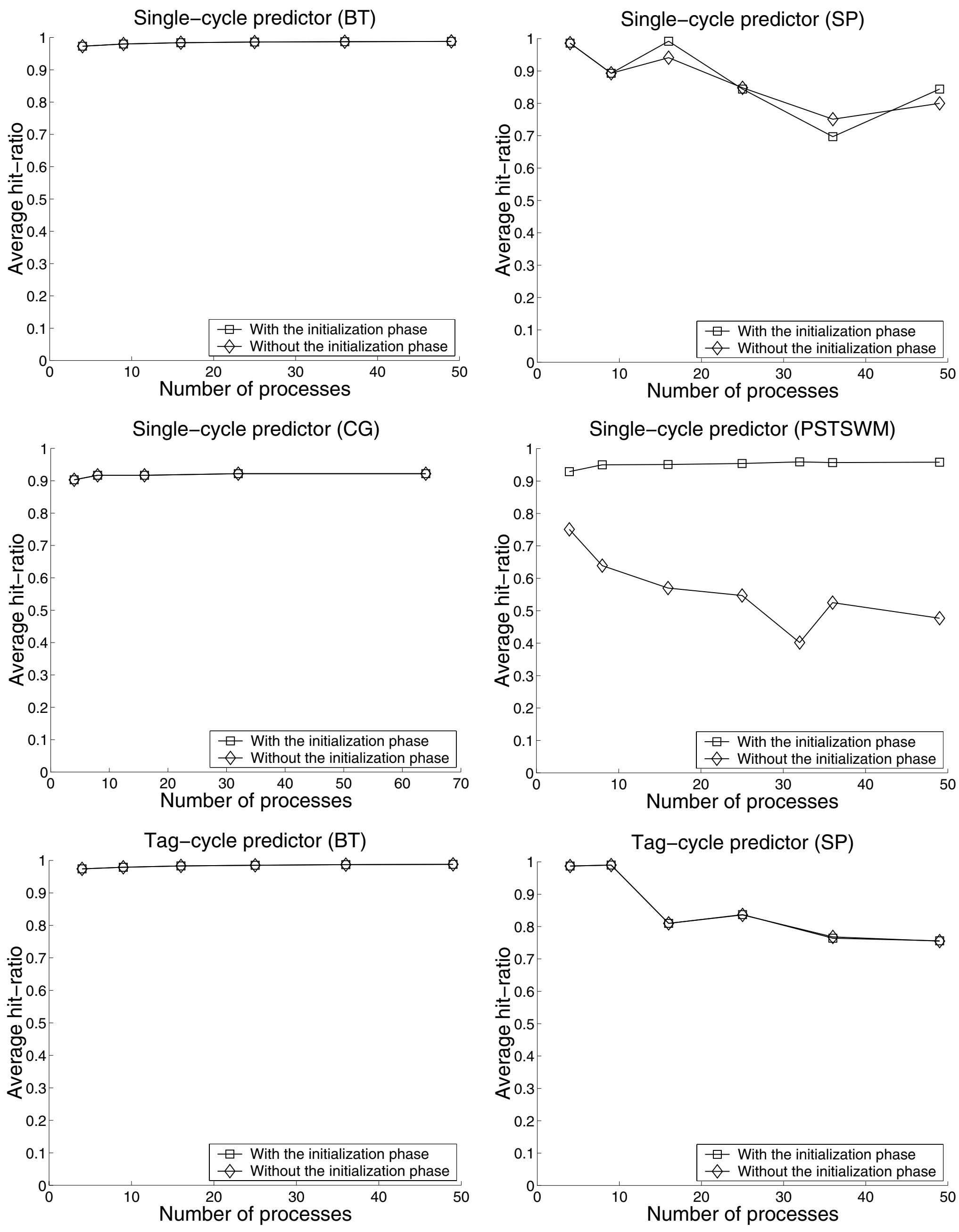

Figure 13. Comparison of the performances of the predictors with the initialization phase with the predictors without the initialization phase over the starting first 100 message reception calls.

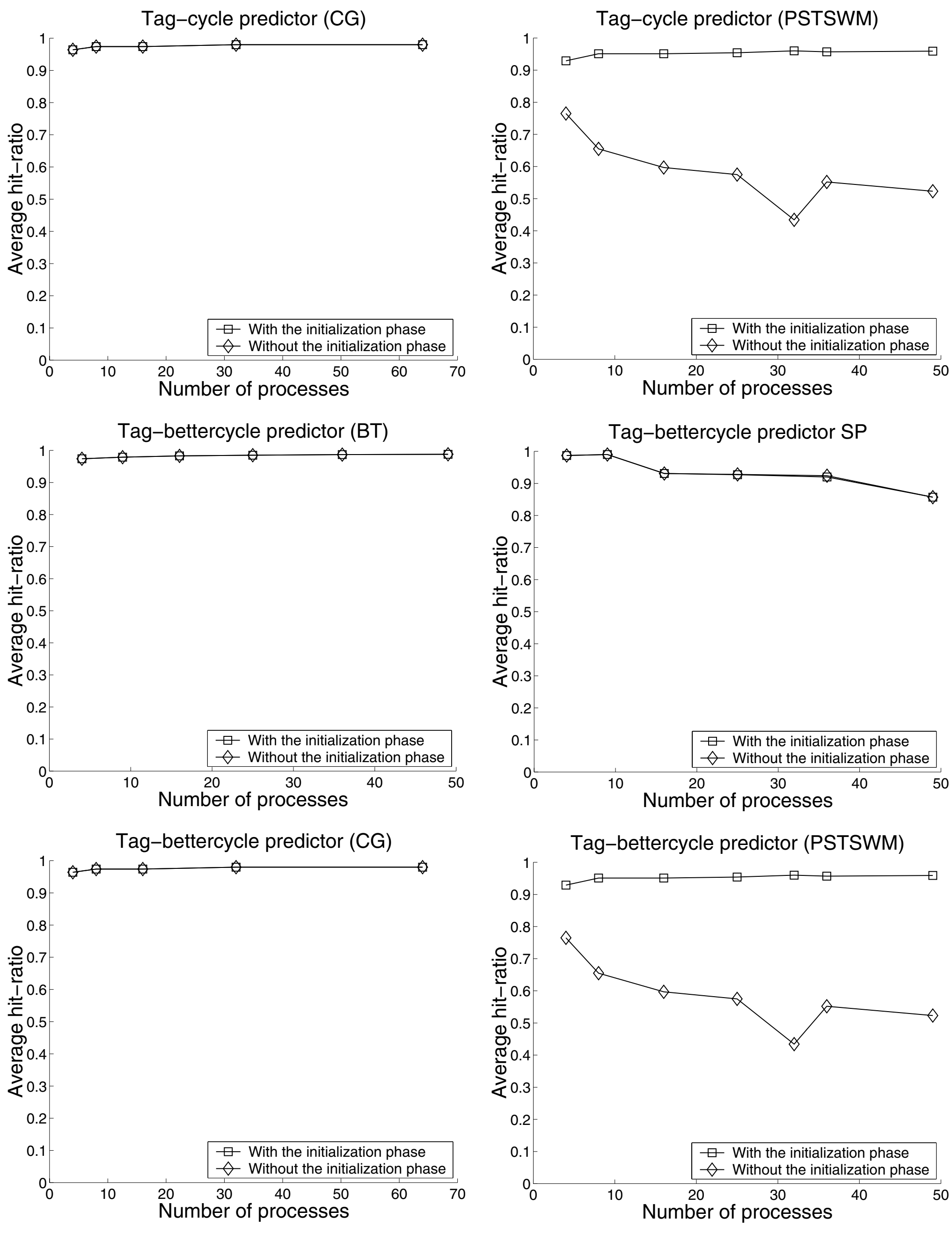

Figure 13. *Continued.*

done on the tags themselves at the network interface. This way, there is no need for the program to pass pre-receive (tag) information to the network interface. However, the performance of these *two-level Tag-based* prediction techniques has not yet been evaluated.

8. CONCLUSIONS

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 intermediate buffering. However, this is not always possible as a message may arrive at the destination where the corresponding receive call has not yet been issued. 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 have presented the performance of these predictors on some parallel applications. The performance results are quite promising and justify more work in this area. We have also presented that these predictors are not sensitive to the starting message reception call and can capture the patterns of communications for the correct prediction.

We envision that these predictors will 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 the 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.

ACKNOWLEDGEMENTS

This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada (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, and the computer staff at the University of Victoria for the access to the University of Victoria IBM SP2. Finally, we would like to thank the anonymous referees for their insightful comments and suggestions.

REFERENCES

1. Boden NJ, Cohen D, Felderman RE, Kulawik AE, Seitz CL, Seizovic JN, Su W-K. Myrinet: A gigabit-per-second local area network. *IEEE Micro* 1995; **15**(1):29–36.
2. Gigabit Ethernet Alliance. Running 1000BASE-T: Gigabit ethernet over copper. *White Paper*, September 1999. http://www.10gea.org/.

3. Eicken TV, Culler DE, Goldstein SC, Schauser KE. Active messages: A mechanism for integrated communication and computation. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Queensland, Australia, May 1992. ACM Press, 1992; 256–265.
4. Lauria M, Pakin S, Chien AA. Efficient layering for high speed communication: The MPI over fast message (FM) experience. *Cluster Computing Journal* 1999; **2**:107–116.
5. Dubnicki C, Bilas A, Chen Y, Damianakis S, Li K. VMMC-2: Efficient support for reliable, connection-oriented communication. *Proceedings Hot Interconnect'97*, Stanford, CA, USA. IEEE Computer Society Press, 1997.
6. Eicken TV, Basu A, Buch V, Vogels W. U-Net: A user-level network interface for parallel and distributed computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain resort, Colorado, USA. ACM Press, 1995; 40–53.
7. Dunning D, Regnier G, McAlpine G, Cameron D, Shubert B, Berry F, Merritt AM, Gronke E, Dodd C. The virtual interface architecture. *IEEE Micro* 1998; **18**(2):66–76.
8. Tezuka H, O'Carroll F, Hori A, Ishikawa Y. Pin-down cache: A virtual memory management technique for zero-copy communication. *First Merged Symposium IPPS/SPDP 1998 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, Orlando, FL, USA. IEEE Computer Society Press, 1998; 308–314.
9. Rodrigues SH, Anderson TE, Culler DE. High-performance local area communication with fast sockets. *USENIX 1997 Annual Technical Conference*, Anaheim, CA, USA, January 1997. USENIX, 1997; 257–274.
10. Banikazemi M, Govindaraju RK, Blackmore R, Panda DK. 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*, San Juan, Puerto Rico, April 1999. IEEE Computer Society Press, 1999; 183–190.
11. Takahashi T, O'Carrol F, Tezuka H, Hori A, Sumimoto S, Harada H, Ishikawa Y, Beckman PH. Implementation and evaluation of MPI on an SMP cluster. *PC-NOW99, International Workshop on Personal Computer based Networks Of Workstations, held in conjunction with IPPS/SPDP'99*, San Juan, Puerto Rico (*Lecture Notes in Computer Science*, vol. 1586). Springer: Berlin, 1999; 1178–1192.
12. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. Version 1.1, June 1995.
13. Afsahi A, Dimopoulos NJ. Efficient communication using message prediction for cluster of multiprocessors. *Proceedings of the CANPC'00, Fourth Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, held in conjunction with HPCA-6*, Toulouse, France (*Lecture Notes in Computer Science*, vol. 1797), January 2000. Springer: Berlin, 2000; 162–178.
14. Afsahi A, Dimopoulos NJ. Communication prediction in message-passing systems. *14th Annual International Symposium on High Performance Computing Systems and Applications, HPCS'2000. High Performance Computing Systems and Applications*, Victoria, Canada, January 2000. Kluwer: Dordrecht, 2002.
15. Afsahi A, Dimopoulos NJ. Hiding communication latency in reconfigurable message-passing environments. *Proceedings IPPS/SPDP 1999, 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico. IEEE Computer Society Press, 1999; 55–60.
16. Mukherjee SS, Hill MD. Using prediction to accelerate coherence protocols. *Proceedings 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain. ACM/IEEE Computer Society Press, 1998; 179–191.
17. Blumrich M, Li K, Alpert R, Dubnicki C, Felten E, Sandberg J. A virtual memory mapped network interface for the SHRIMP multicomputer. *Proceedings 21st Annual International Symposium on Computer Architecture*, Chicago, IL, USA. IEEE Computer Society Press, 1994; 142–153.
18. Geoffray P, Prylli L, Tourancheau B. BIP-SMP: High performance message passing over a cluster of commodity SMPs. *SC'99: High Performance Networking and Computing Conference*, Portland, OR, USA, November 1999. ACM/IEEE Computer Society Press, 1999.
19. Chu H. Zero-copy TCP in Solaris. *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, USA. USENIX, 1996; 253–263.
20. Druschel P, Peterson LL. Fbufs: A high-bandwidth cross-domain transfer facility. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, NC, USA. ACM Press, 1993; 189–202.
21. Fahringer T, Mehofer E. Buffer-safe and cost-driven communication optimization. *Journal of Parallel and Distributed Computing* 1999; **57**(1):33–63.
22. Gupta M, Schonberg E, Srinivasan H. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems* 1996; **7**(7):689–704.
23. Lai A-C, Falsafi B. Memory sharing predictor: The key to a speculative coherent DSM. *Proceedings 26th Annual International Symposium on Computer Architectures*, Atlanta, Georgia, USA. ACM/IEEE Computer Society Press, 1999; 172–183.
24. Kaxiras S, Goodman JR. Improving CC-NUMA performance using instruction-based prediction. *International Symposium on High Performance Computer Architecture*, Orlando, FL, USA. ACM/IEEE Computer Society Press, 1999; 161–171.
25. Zhang Z, Torrellas J. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group

prefetching. *Proceedings 22nd Annual International Symposium on Computer Architectures*, Santa Margherita, Italy. ACM/IEEE Computer Society Press, 1995; 188–199.

26. Dahlgren F, Dubois M, Stenström P. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1995; **6**(7):733–746.
27. Mowry T, Gupta A. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing* 1991; **12**(2):87–106.
28. Bailey DH, Harsis T, Saphir W, der Wijngaart RV, Woo A, Yarrow M. The NAS Parallel Benchmarks 2.0. *Report NAS-95-020*, NASA Ames Research Center, December 1995.
29. Worley PH, Foster IT. Parallel spectral transform shallow water model: A runtime-tunable parallel benchmark code. *Proceedings Scalable High Performance Computing Conference*, Knoxville, TN, USA. IEEE Computer Society Press, 1994; 207–214.
30. Kim J, Lilja DJ. Characterization of communication patterns in message-passing parallel scientific application programs. *Proceedings of the CANPC'98, Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, held in conjunction with HPCA'98*, Las Vegas, Nevada, USA (*Lecture Notes in Computer Science*, vol. 1362), February 1998. Springer: Berlin, 1998; 202–216.
31. Dao BV, Yalamanchili S, Duato J. Architectural support for reducing communication overhead in multiprocessor interconnection networks. *Proceedings Third International Symposium on High Performance Computer Architecture*, San Antonio, TX, USA. IEEE Computer Society Press, 1997; 343–352.
32. Chodnekar S, Srinivasan V, Vaidya A, Sivasubramaniam A, Das C. Towards a communication characterization methodology for parallel applications. *Proceedings Third International Symposium on High Performance Computer Architecture*, San Antonio, TX, USA. IEEE Computer Society Press, 1997; 310–321.
33. Afsahi A. Design and evaluation of communication latency hiding/reduction techniques for message-passing environments. *PhD Dissertation*, Department of Electrical and Computer Engineering, University of Victoria, April 2000.