Hiding Communication Latency in Reconfigurable Message-Passing Environments

Ahmad Afsahi Nikitas J. Dimopoulos

Technical Report ECE-99-3

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

January, 1999

(Revised March, 1999)

Abstract. *Message-passing multicomputers are composed of a large number of computing nodes that communicate with each other by exchanging messages through their interconnection networks. The communication overhead is one of the most important factors affecting their performance. In the first part of our work, we present evidence (through the analysis of several parallel benchmarks) that there exists communications locality, and that this locality is "structured". We have utilized this structure in a number of heuristics that can be used to "predict" the target of subsequent communication requests. This technique, can be applied directly to reconfigurable optical interconnects to hide the communications latency by reconfiguring the communications network concurrently to the computation. This method has applicability to conventional interconnection networks as well. In the second part of out work, we show the minimum execution time of the computation phases of these parallel benchmarks on an IBM SP2 using its high-performance switch and the user space mode. By comparing the inter-communication computation times of these parallel benchmarks with some specific reconfiguration times, we show that, most of the time, we are able to fully utilize these computation times for the concurrent reconfiguration of the interconnect when we know, in advance, the next target using one of the proposed high hit-ratio target prediction algorithms. We also present the performance enhancements of the proposed heuristics on the benchmarks for the total reconfiguration time (reconfiguration speedup) and the application speedup.*

1.0 Introduction

Message-passing multicomputers are composed of a large number of computing nodes that communicate with each other by exchanging messages through their interconnection networks. As the communication overhead is one of the most important factors affecting its performance, there has been a great deal of interest in the design of the interconnection networks.

Optics is ideally suited for implementing interconnection networks because of its superior characteristics over electronics [22,17,9] such as higher interconnection density, higher bandwidth, suitability for reconfigurable interconnects, larger number of fan-in and fan-out, lower error rate, freedom from planar constraints (light beams can easily cross each other), immunity from electromagnetic field and ground loops, lower signal crosstalk etc. It is foreseen that future massively parallel machines will employ optical interconnections in an ever increasing manner.

In free-space optical interconnects, optical signals can propagate very close to each other and pass each other without interaction. One of the main features of the optical interconnects is the capability to reconfigure on demand which is very suitable for the construction of 3-D VLSI computers [14]. Free-space optical interconnects use free-space (vacuum, air or glass) for optical signal propagation and include two basic components: one is the optoelectronic device for photon generation or modulation such as VCSELs and SEEDs, while the other component is the optical beam router to redirect or distribute optical beams. Optical beam routing in a free-space optical interconnection network often employs external optical elements such as holograms, mirrors, prisms, lenses etc.

Research in massively parallel computers using optical interconnections is focusing in two major directions; One is to implement traditional, and/or design new, point-to-point interconnection networks [13] but utilizing the advantages of the new emerging optical technologies to overcome the constraints of metal-based communications, while the second direction is to design new interconnection networks [4] based on the constraints of optical devices, such as optical passive star coupler (OPS) interconnection networks which can be modeled by hypergraphs. Along the first direction, we define an abstract model [1] for our complete reconfigurable free-space optical interconnection network for massively parallel computers, and discuss its characteristics.

Definition A *reconfigurable optical network*, *RON* (*k, N*), consists of *N* computing nodes with their own local memory. A node is capable of connecting directly to any other node. A node can establish *k* simultaneous connections. These connections are established dynamically by reconfiguring the optical interconnect. The links remain established until they are explicitly destroyed.

A simplified block diagram of the network is shown in Figure 1. Messages are sent using circuit-switching. That is, a connection must be established between the source and destination pair before the message is sent. Each node has the ability to simultaneously send and receive *k* messages on its *k* links (the *k-port* model), or exactly one message on one of its links (the *single-port* model). Full-duplex communication where a node can send and receive messages at the same time is supported. Various implementation technologies exist to embody the above abstract model. Such technologies include computer generated holograms and deformable mirrors for switching, frequency hoping for coding, wavelength tuning for transceivers, VCSELs and SEEDs for photon generation or modulation. In this paper, we are particularly interested in the abstract model of our proposed network and do not address these implementation issues. We shall assume that one or more of the technologies outlined above will be used to implement the proposed interconnect. Under such an implementation, the

various overheads associated with the reconfiguration of the network (such as beam steering, setting up the computer-generated holograms, tuning the transmitters, or sending the frequency code in a frequency hoping implementation etc.) are lumped together as the reconfiguration delay *d*.

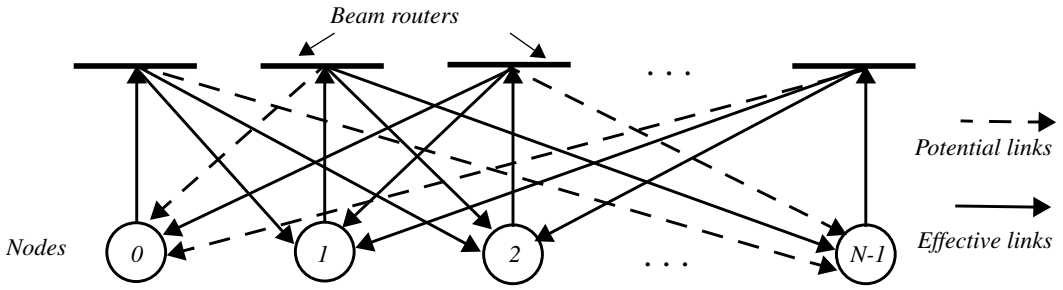


FIGURE 1. *RON (k, N),* a massively parallel computer interconnected by a complete free-space optical interconnection network

An important concern is to model the communication time *T* required to send a message from one node to another. In the linear model, the communication time depends, among other things, on the length of the message, and it is formulated as $T = t_s + l_m\tau$ where l_m is the length of the message, τ is the per unit transmission time, and t_s, the setup time, is the time required to prepare the message, such as adding a header, a trailer, memory copying etc. and logically is the time to send a zero byte message. For our case, we amend the linear model by explicitly including the reconfiguration delay *d* that is necessary for a node to configure a link that would connect directly to its target node. The transmission time then becomes $T = d + t_s + l_m\tau$. The time on the fly, $l_m\,\tau$, is negligible compared to the setup time, t_s, and the reconfiguration delay, *d.* In the current generation of parallel computer systems, the setup time is several 10 microseconds [7]. Several researchers are working to minimize this cost by user-level messaging techniques such as active messages [8] and fast messages [18]. However, in this work we are particularly interested in the techniques that hide the reconfiguration delay, *d*.

We shall assume that a node sends a message to another node by first establishing a link to the target (hence the reconfiguration delay *d*) and then sending the actual message over the established link. It is obvious that if the link is already in place, then the configuration phase does not enter the picture with a commensurate savings in the message transmission time. The main objective of this work is therefore to establish efficient algorithms where the link establishment costs are minimized. The stated objective can be accomplished, if the target of the communication operation can be "predicted" before the message itself is available. In this way, the communication pathway can be established and be ready to be used as soon as the message to be sent becomes available.

There are several ways of accomplishing this. If the communication operation is regular and known, then it is possible that one can determine the destinations and the instances that these shall be used. We have developed such algorithms for broadcasting and have reported these in [1]. However, if the algorithm is not known, regular or simple, the approach mentioned above cannot be used.

In the context of the shared memory programming, there are several works on hardware-controlled and software-controlled prefetching of the next shared data request [16,20,24]. T. Mowry and A. Gupta [16] have used prefetching, multithreading, and caching to hide/reduce the latency in shared memory multiprocessors. M. F. Sakr et al. [20] have used time series and neural networks for the pre-

diction of the next memory requests in shared memory multiprocessors using reconfigurable interconnects.

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 the *communications locality* properties of parallel applications [11,12]. J. Kim and D. J. Lilja [11] have recently 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.

We need to define the terms *communications locality* and *caching* in conjunction with this work. By communications locality we mean that if a certain source-destination pair 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. If communications locality exists in parallel applications, then it is possible to *cache* the configuration that a previous communication request has made and reuse it at a later stage. Caching in the context of this discussion will mean that when a communication channel is established it will remain established until it is explicitly destroyed. As we already mentioned, in the context of free-space optical interconnect maintaining a communication channel established does not interfere with communications that are in progress in other parts of the network.

This paper focuses on two parts. The first part of our work which is an extension of our work in [2] is to explore the effect that a number of heuristics has in predicting the target of a communication request. For these studies, we have utilized a number of parallel benchmarks, and extracted the communication traces on which we applied our heuristics. These heuristics can be used either dynamically in real-time at the communication assist/network interface and/or off-line in the compilation phase. It is worth mentioning that these heuristics can be used in any circuit-switched networks including the works proposed in [6,23] in which the authors exploit the communications locality to improve the performance of parallel computers in wormhole/circuit-switched electronic (wave switching), and circuit-switched optical interconnection networks, respectively.

The second part considers the execution time of the computation phases of these parallel benchmarks on an IBM SP2 using its high performance switch under the user space mode. We present that these computation times, are most of the time, sufficiently large for the reconfiguration that proceeds concurrently with the computation to terminate before the computation. We also present the performance enhancements of these heuristics on the total reconfiguration delay and on the benchmarks by using the obtained computation/communication traces and the heuristics prediction profiles under different reconfigurable costs and processor speeds.

In the following section, we shall discuss the benchmarks studied in this paper. Section 3 explains the proposed target prediction heuristics and presents their performance in terms of the parallel benchmarks. In section 4, we compare the inter-send computation times of the benchmarks on the SP2 with different reconfiguration costs, and present the performance enhancements of the proposed heuristics. Finally, we conclude with section 5.

2.0 Parallel Benchmarks

We have used some well-known parallel benchmarks form the NAS parallel benchmarks suite (NPB) [3], the Parallel Spectral Transform Shallow Water Model (PSTSWM) parallel benchmark [21], and the pure QCD Monte Carlo Simulation Code with MPI (QCDMPI) parallel benchmark [10]. The NAS parallel benchmarks suite have been developed at the NASA Research Center to study the performance of parallel supercomputers and networks of workstations. The NAS parallel Benchmarks (NPB) are a set of eight benchmark problems, each of which focuses on some important aspect of highly parallel supercomputing for aerophysics applications. The NPB consists of five "kernels", and three "simulated computational fluid dynamic (CFD) applications". In this paper, we are only interested in the patterns of the point-to-point communications between pair-wise nodes. Therefore the EP, FT, and IS kernels are not suitable for our study. EP and FT use only collective communication operations while each node in the IS kernel always communicates with a specific node.

The conjugate gradient kernel, CG, tests the performance of the system for unstructured grid computations which by their nature require irregular communications which is a challenge for all kinds of parallel computers. In this kernel, the inverse power method is used to find an estimate of the largest eigenvalue of a symmetric positive-definite sparse matrix with a random pattern of non-zeros. The second kernel benchmark is a simplified multigrid kernel, MG, which solves a 3-D poisson PDE. This code is a good test of both short and long distance highly structured communication. The three simulated CFD application benchmarks, LU, SP, and BT are intended to accurately represent the principal computational and data movement requirements of modern CFD applications. The lower-upper diagonal benchmark, LU, employs a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse block 5×5 lower and upper triangular system. In the scalar pentadiagonal benchmark, SP, multiple independent systems of non-diagonally dominant, scalar pentadiagonal equations with a 5×5 block size are solved. In the block tridiagonal benchmark, BT, multiple independent systems of non-diagonally dominant, block tridiagonal equations with a 5×5 block size are solved.

PSTSWM [21] was developed by P. H. Worley of Oak Ridge National Laboratory and I. T. Foster of Argonne National Laboratory. PSTSWM is a message-passing benchmark code and parallel algorithm testbed that solves the nonlinear shallow water equations on a rotating sphere using the spectral transform method. PSTSWM was developed to evaluate parallel algorithms for the spectral transform method as it is used in global atmospheric circulation models.

QCDMPI [10], written by S. Hioki at Tezukayama University, is a pure Quantum Chromo Dynamics simulation code with MPI calls. It is a powerful tool to analyze the non-perturbative aspects of QCD. This program can be applied to any dimensional QCD like 3-dimensional QCD in which we can obtain the color and/or quark confinement mechanism.

We have used the MPI [15] implementation of the NAS benchmarks (version 2.3, W and A classes), the PSTSWM (version 6.2), and the QCDMPI (version 1.4) benchmarks on the IBM SP2. We wrote our own profiling codes using the wrapper facility of the MPI to gather the communication traces and the timing profiles of these applications. The timing results have been gathered using the high performance switch of the IBM SP2 under the user space mode when we had an exclusive access to the system. It is worth mentioning that gathering communication traces does not affect the communication patterns of these applications. Meanwhile, we excluded all timing overheads in our profiling codes

when getting the execution times of the computation and communication phases of the parallel benchmarks.

In the following section, we first review the performance of the LRU algorithm [11] on the benchmarks and then present new algorithms employing different heuristics to hide the communications latency on our reconfigurable optical interconnect. Although the derived results are for the above mentioned parallel benchmarks, however, we believe that these benchmarks are representative of the existing scientific and engineering parallel applications.

3.0 Latency hiding heuristics

The set of heuristics proposed in this section predict the destination node of a subsequent communication request based on a past history of communication patterns on a per source node basis. Our heuristics would execute on each node of the multicomputer, and predict the destination nodes for communications originating at the node on which they reside.

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 destination node was correct out of all communication requests. In the following figures, the hit-ratios shown for the performance of the heuristics are either the minimum, the average, or the maximum of the hit-ratios of all nodes.

3.1 The LRU, FIFO and LFU heuristics

The *Least Recently Used* (LRU) [11], *First-In-First-Out* (FIFO) and *Least Frequently Used* (LFU) heuristics, all maintain a set of k (k is the *window size*) message destinations. If the next message destination is already in the set, then a hit is recorded. Otherwise, a miss is recorded and the new destination replaces one of the destinations in the set according to which of the LRU, FIFO or LFU strategies is adopted. The window size, k, corresponds to the number of ports used. Figure 2, shows the results of the LRU, FIFO, and LFU heuristics on the benchmarks when the number of processors is 64. Figure 3, shows the size scalibiltiy of the these heuristics on the benchmarks. It is clear that the hit-ratios in all benchmarks approach 1 as the window size increases. The performance of the FIFO algorithm is almost the same as the LRU for all benchmarks. However, the LFU algorithm has a better performance than the LRU and FIFO heuristics, the exception is for the LU benchmark, when $k = 2$ and $N = 16$, 32, and 64.

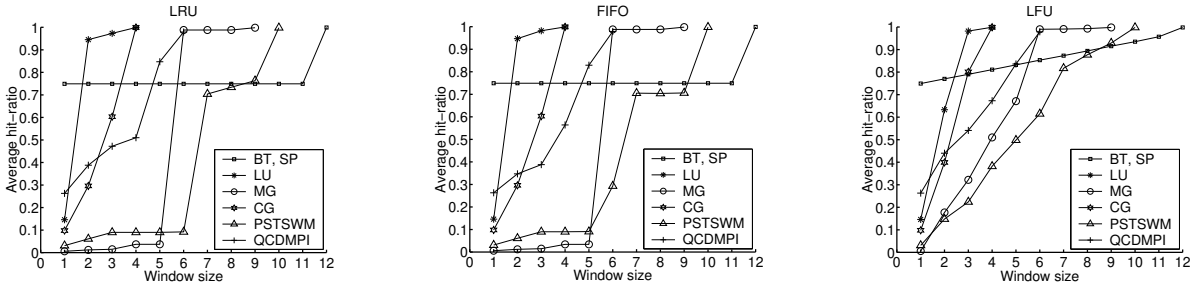


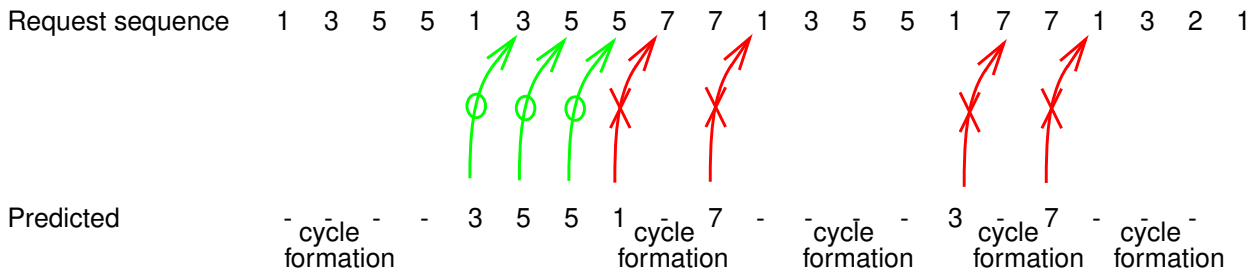
FIGURE 2. Effects of the LRU, FIFO, and LFU heuristics on the benchmarks, when N = 64

3.2 The Single-cycle heuristic

The LRU, FIFO, and the LFU heuristics perform better when k is sufficiently large. However, this adds to the hardware complexity as k links should be setup before the next message is ready to

take place. Therefore, we are looking for other heuristics that perform well under single-port modeling.

The Single-cycle heuristic is based on the fact that if a group of destinations are requested repeatedly in a cyclical fashion, then a single port can accommodate these requests by ensuring that the connection to the subsequent node in the cycle can be established as soon as the current request terminates. This heuristic implements a simple cycle discovery algorithm. Starting with a *cycle-head* node (this is the first node that is requested at start-up, or the node that causes a miss), we log the sequence of requests until the cycle-head node is requested again. This stored sequence constitutes a cycle, and can be used to predict the subsequent requests. If the predicted node coincides with the subsequent requested node, then we record a hit. If the requested node does not coincide with the predicted one, then we record a miss and the cycle formation stage commences with the cycle-head being the node that caused the miss. The following example illustrates the heuristic used. The top trace represents the sequence of requested destination nodes, while the bottom trace represents the predicted nodes set according to the Single-cycle heuristic. The arrows with the cross represent misses, while the ones with the circle represent hits. The "dash" in place of a predicted node indicates that a cycle is being formed, and therefore no predicted nodes are offered (note that this is also added to the misses).



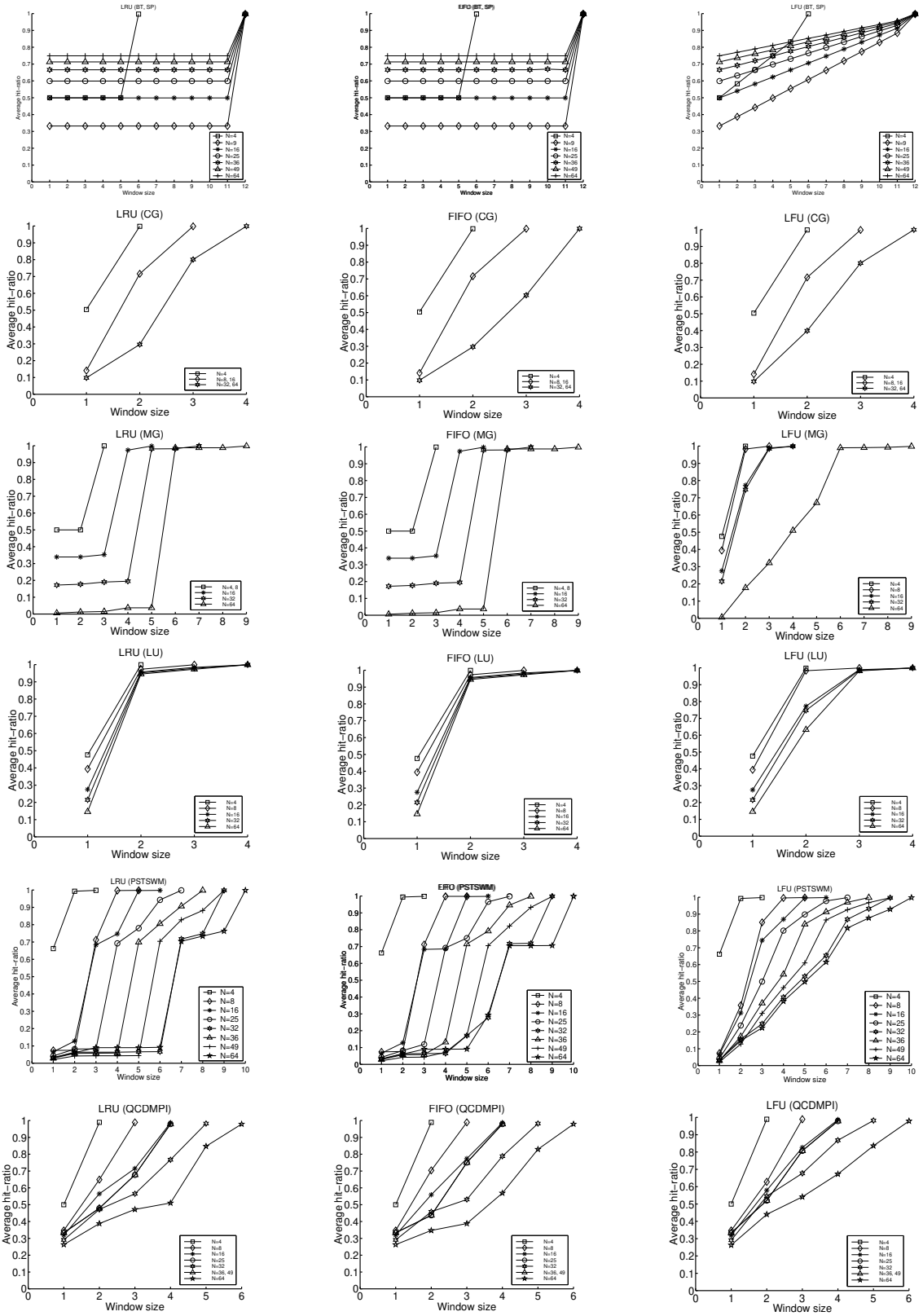


FIGURE 3. Effects of the scalibilty of the LRU, FIFO, and LFU heuristics on the benchmarks

Figure 4, shows the behavior of this algorithm. This algorithm behaves much better than the LRU, FIFO and LFU heuristics under single-port modeling for the LU and CG benchmarks, and for the MG, PSTSWM and QCDMPI benchmarks (except when $N = 4, 8$). However its performance is worse for the BT and SP benchmarks. One of the reasons is that in these benchmarks there exist cycles of length one (such as the one composed of node 7 in the example above). For these situations, there will always be two misses until the heuristic determines that there is a cycle of length 1. The Single-cycle2 heuristic presented next improves the performance for the BT and SP benchmarks.

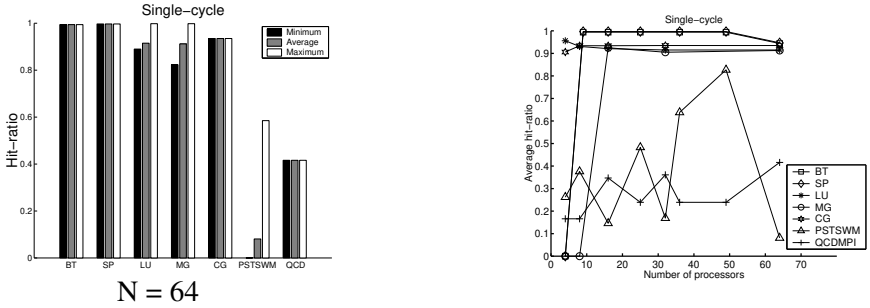
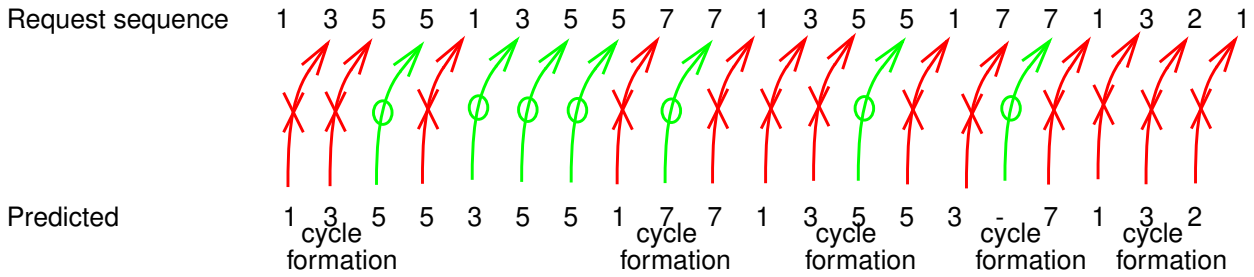


FIGURE 4. Effects of the Single-cycle heuristic on the benchmarks

3.3 The Single-cycle2 heuristic

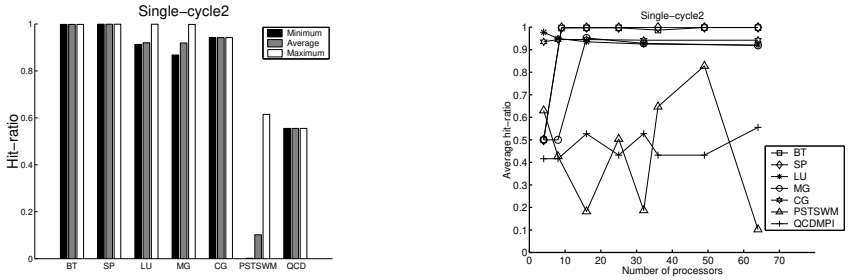
The *Single-cycle2* heuristic is identical to the single-cycle heuristic with the addition that during cycle formation, the previously requested node is offered as the predicted node. If a miss occurs during cycle formation, the formation phase continues until a cycle is formed. Then and only then misses cause a new cycle formation phase to begin. We applied the Single-cycle2 heuristic to the request sequence of the previous example. As it was expected, the Single-cycle2 heuristic reacts better to cycles of length 1.



This algorithm has better performance than the single-cycle algorithm especially for the BT and SP benchmarks, as shown in Figure 5. This heuristic performs much better than the LRU, FIFO, and LFU algorithms under single-port modeling for the LU, MG, and CG benchmarks, and for the PSTSWM and QCDMPI benchmarks (except when $N = 4$). However, its hit-ratios for the BT, and SP benchmarks is slightly less than the hit-ratios for the LRU, FIFO, and LFU algorithms.

3.4 The Better-cycle and Better-cycle2 heuristics

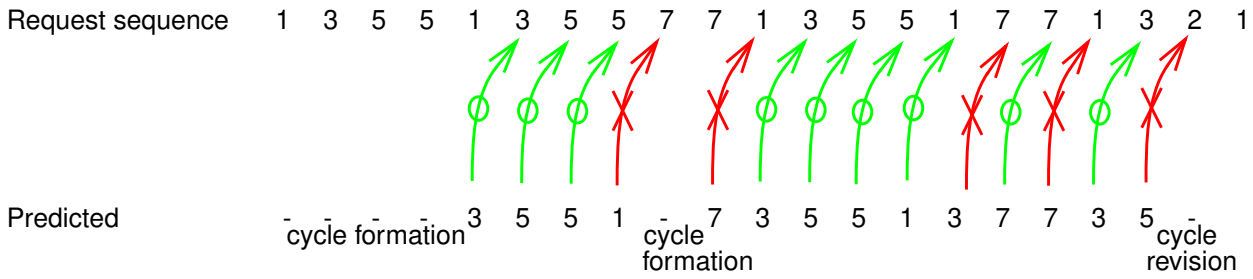
The problem with the Single-cycle and Single-cycle2 algorithms was that as soon as a destination node breaks a cycle we throw away the previous cycle and form a new cycle. Then we just rely only on the new cycle to predict the next communication request. In the *Better-cycle* heuristic, we keep the last cycle associated with each cycle-head encountered in the communication pattern of each



N = 64

FIGURE 5. Effects of the Single-cycle2 heuristic on the benchmarks

node. This means that when a cycle breaks we keep this cycle in memory for later references. If we haven't already seen the new cycle-head then we form a cycle for it, otherwise we predict the next communication request based on the member of the cycle associated with this new cycle-head that we have from the past in memory. If the predicted node coincides with the subsequent requested node, then we record a hit. If not, then we record a miss and revise the cycle for this cycle-head. The following example illustrates how the better-cycle heuristic works. It is clear that the first cycle associated with cycle-head 1 consists of destination nodes 1, 3, 5, and 5. However, in the fourth appearance of this cycle-head a revised cycle forms which contains destination nodes 1, 3, and 2.



The state diagram and the performance of this heuristic on the benchmarks are shown in Figure 6 and-Figure 7, respectively. It is evident that its performance is exceptionally better for all benchmarks compared to the Single-cycle heuristic.

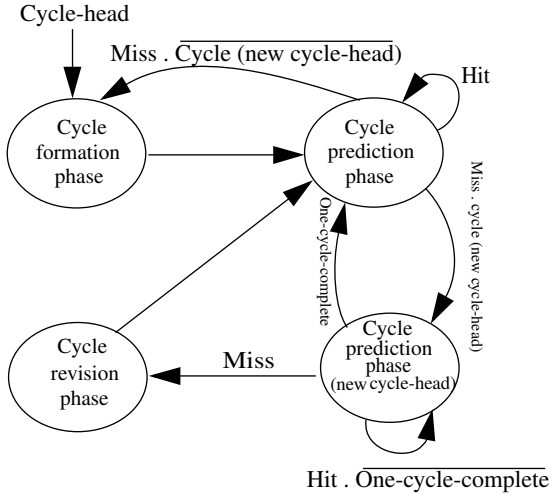
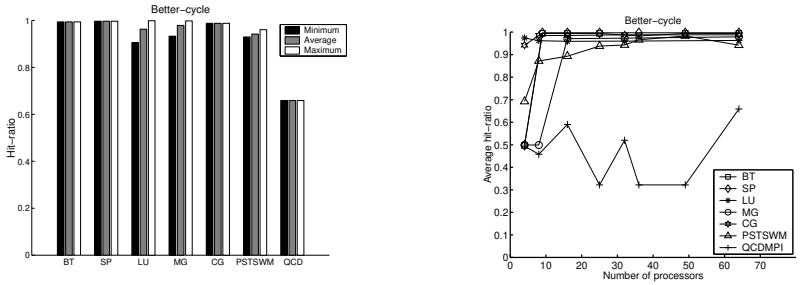


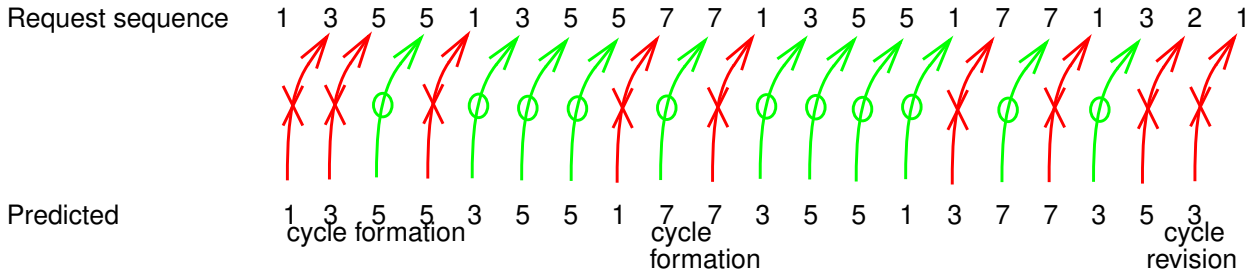
FIGURE 6. State diagram of the Better-cycle algorithm



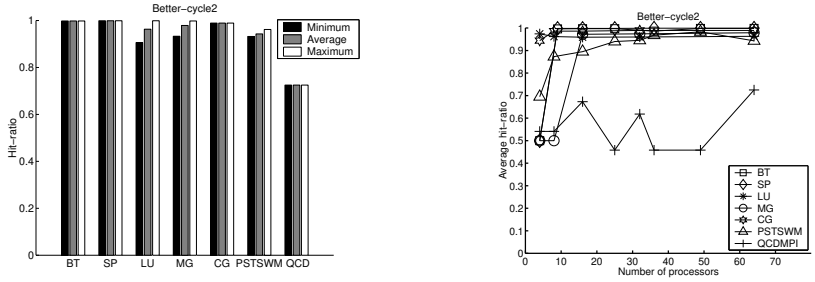
N = 64

FIGURE 7. Effects of the Better_cycle heuristic on the benchmarks

The *Better-cycle2* heuristic is identical to the Better-cycle heuristic with the addition that during cycle formation and cycle revision phases the previously requested node is offered as the predicted node. The following examples shows how Better-cycle2 works.



This heuristic has better performance than the Single-cycle and Single-cycle2 heuristics, and the Better-cycle heuristic for the BT, SP, and the QCDMPI benchmarks. The performance of this heuristic is shown in Figure 8.



N = 64

FIGURE 8. Effects of the Better_cycle2 heuristic on the benchmarks

3.5 The Tagging heuristic

This heuristic assumes a communications environment in the sense that a particular communications request (send) in a section of code, will be to the same target node with a large probability. Therefore, as the execution trace nears the section of code in question, it can cause the communications environment to establish the connection to the target node before the actual communications request is issued. This can be implemented with the help of the compiler or by the programmer through a *pre-connect(tag)* operation which will force the communication system to establish the

communications connection before the actual communications request is issued. This technique is similar to the prefetch operation advocated by T. Mowry and A. Gupta [16].

For our experiments, we attach a different tag to each of the communication requests found in the benchmarks. To this tag, we assign the requested target node. A hit is recorded if in subsequent encounters of the tag, the requested communication node is the same as the target already associated with the tag. Otherwise, a miss is recorded and the tag is assigned the newly requested target node. This tagging technique is similar to the technique used in the branch cache of the MC68060 [5].

The performance of the tagging heuristic is presented in Figure 9. As it can be seen, the tagging heuristic results in an excellent performance (hit ratios in the upper 90%) for all the benchmarks except the CG, PSTSWM, and the QCDMPI benchmarks. The reason is that these benchmarks include send operations with a target address calculated based on loop variables. Thus, the same section of code cycles through a number of different target addresses. As we have seen in the section 3.6, the Better-cycle and Better-cycle2 heuristics are excellent in discovering such cyclic occurrences for the CG and PSTSWM benchmarks. Meanwhile, the Better-cycle2 heuristic has better performance for the QCD-MPI benchmark compared to the Tagging and other cycle heuristics.

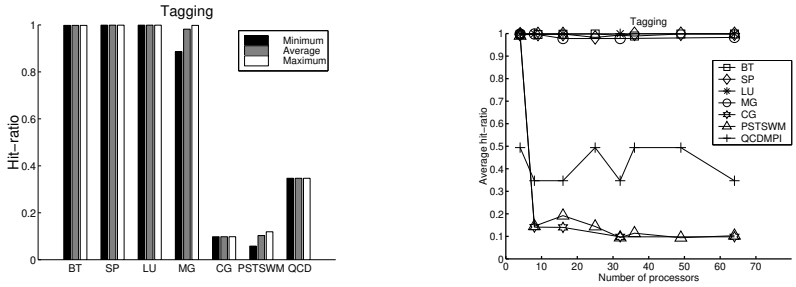
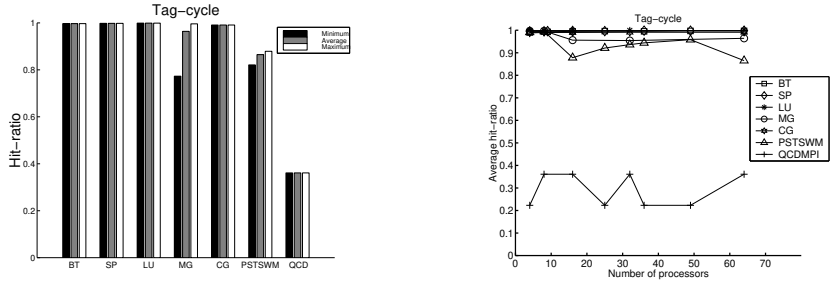


FIGURE 9. Effects of the Tagging heuristic on the benchmarks

3.6 The Tag-cycle and Tag-cycle2 heuristics

The tagging heuristic does not have a good performance on the CG, PSTSWM, and the QCD-MPI benchmarks while the Single-cycle and Single-cycle2 heuristics showed good results for the CG benchmark. We combine the Tagging algorithm with the Single-cycle algorithm and call it the Tag-cycle algorithm.

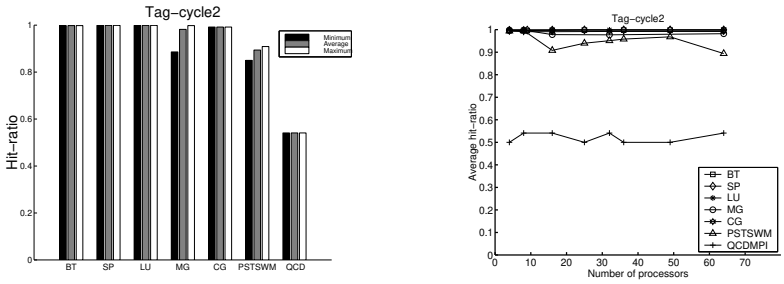
In the Tag-cycle heuristic, we attach a different tag to each of the communication requests found in the benchmarks and do a Single-cycle discovery algorithm on each tag. To this tag, we assign the requested target node, to be called *tagcycle-head* node (this is the first destination node that is requested 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. A hit is recorded if in subsequent encounter of the tag, the requested communication node is the same as the predicted node in the cycle. If not, then we record a miss and the cycle formation stage begins with the tagcycle-head being the node that caused the miss. The Tag-cycle heuristic perform exceptionally well across all the benchmarks except for the QCDMPI benchmark, as shown in Figure 10. Its performance is slightly worse for the BT benchmark compared to the Tagging heuristic.



N = 64

FIGURE 10. Effects of the Tag-cycle heuristic on the benchmarks

The Tag-cycle2 heuristic is identical to the Tag-cycle heuristic with the addition that during cycle formation, similar to the Single-cycle2 heuristic, the previously requested node is offered as the predicted node. The performance of Tag-cycle2 heuristic, as shown in Figure 11, is better than the Tagging and Tag-cycle heuristics for all benchmarks.

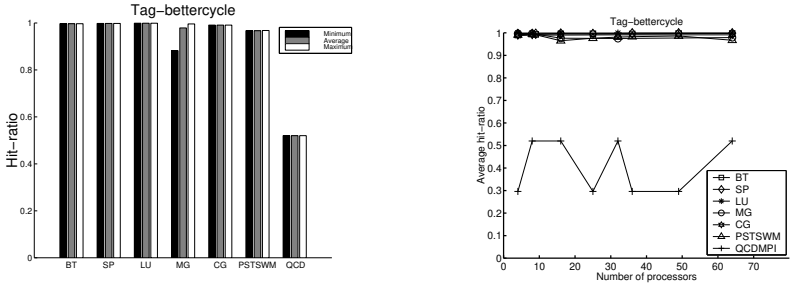


N = 64

FIGURE 11. Effects of the Tag-cycle2 heuristic on the benchmarks

3.7 The Tag-bettercycle and Tag-bettercycle2 heuristics

Better-cycle and Better-cycle2 algorithms have better performance on the parallel benchmarks than the Single-cycle and Single-cycle2 algorithms. Therefore, we combine the Better-cycle and Better-cycle2 algorithms with the tagging algorithms to get better performances than the Tag-cycle and Tag-cycle2 algorithms. We call them Tag-bettercycle and Tag-bettercycle2 heuristics. The performance of these two heuristics are shown in Figure 12, and Figure 13.

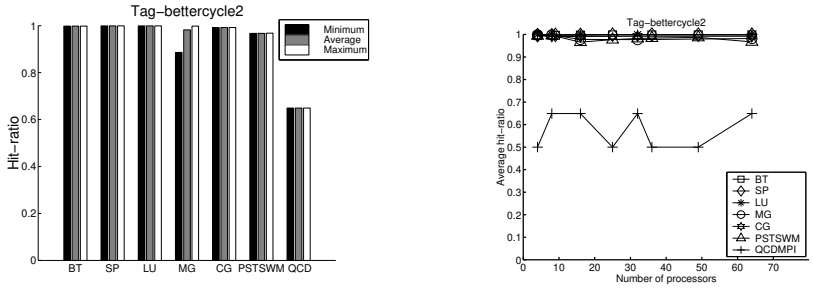


N = 64

FIGURE 12. Effects of the Tag-bettercycle heuristic on the benchmarks

In Tag-bettercycle heuristic, we attach a different tag to each of the communication requests found in the benchmarks and do a Better-cycle discovery algorithm on each tag. The Tag-bettercycle2 heuristic is identical to the Tag-bettercycle heuristic with the addition that during cycle formation, similar to the Better-cycle2 heuristic, the previously requested node is offered as the predicted node. The perfor-

mance of Tag-bettercycle for the QCDMPI benchmark is better than the Tag-cycle algorithm, but not better than the Tag-cycle2 heuristic. However, the Tag-bettercycle2 heuristic is superior to all other heuristics for all parallel benchmarks.



N = 64

FIGURE 13. Effects of the Tag-bettercycle2 heuristic on the benchmarks

Figure 14, presents a comparison of the performance of the heuristics presented in this work under single-port assumption when the number of processors is 64. That is only one communications channel is available at any time, and this is reconfigured on demand. As it can be seen, the Tag-bettercycle2 heuristic is the best for all benchmarks and its hit-ratio is consistently very high for all benchmarks considered here.

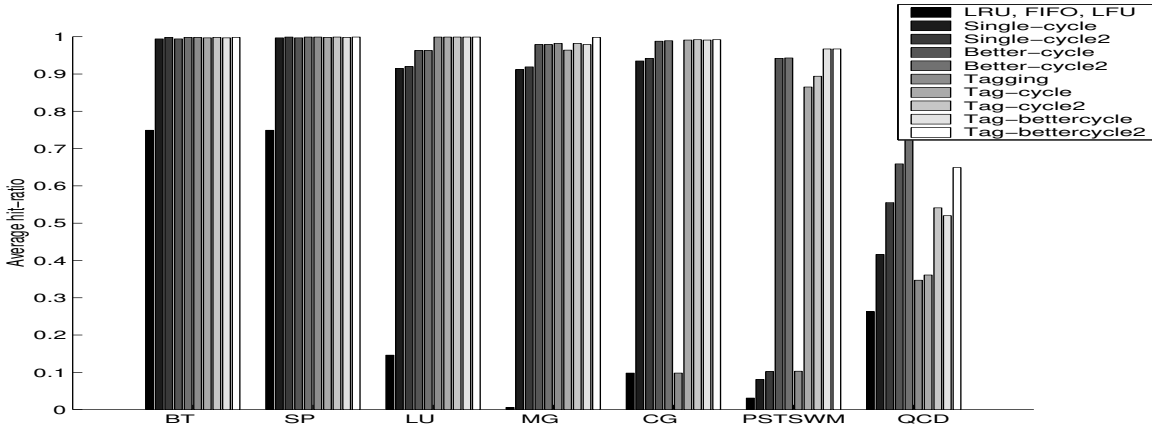


FIGURE 14. Comparison of the performance of the heuristics discussed in this work for the benchmarks under single-port modeling when number of processors is 64

4.0 Inter-send computation times and Speedup

To reconfigure the optical interconnect concurrently to the computation, two conditions are necessary: (1) An accurate prediction of the destination; (2) Enough lead time so that the reconfiguration of the interconnect be completed before the communication request arrives.

In the previous section of the work, we have presented a number of heuristics that can be used to accurately predict the destination of the subsequent communication request. In this section, we shall argue that, at least in the application benchmarks studied, there is enough computation preceding a communication request to effectively hide the reconfiguration cost.

We have used a four node IBM SP2 (160 MHZ P2SC thin nodes with 256MB RAM and second generation high performance switch) and run the suite of benchmarks at the user space, one process per node, and under an exclusive access to the node. This avoided any task switching that might have

affected our measurements. Our measurements determined a lower bound on the inter-send computation times (i.e. the time devoted to computation between two send requests). The inter-send computation measurements excluded any overhead associated with other communication primitives (e.g. receive) and it can thus be considered as a lower bound on the pure computation (please refer to the appendix B).

Table 1, shows the minimum pure computation times for each node of these benchmarks. The times for the CG benchmark and for some nodes of other benchmarks are quite large. However, the majority of minimum computation times are around 4 microseconds while for the PSTSWM is less than 1 microsecond. However, examining the distribution of the inter-send times, revealed that they are quite widely distributed. Figure 15, presents a summary of the distribution of the inter-send computation times.

Researchers in optical engineering are using different approaches to reduce the reconfiguration time of the optical interconnects [19]. In [19], the authors present a reconfiguration time of 25 microseconds for an experimental reconfigurable optical interconnect. Therefore, we compare the pure computation times of these benchmarks with this 25 microseconds reconfiguration time, and with reconfiguration times of 10, 5, and 1 microseconds as a measure of future advancements in this area, as shown in Figure 15. This figure presents the percentage of the number of computation times less than and more than 1, 5, 10, and 25 microseconds for each node of the SP2 and for each application. From the plots for the CG, QCDMPI and PSTSWM benchmarks, it can be easily seen that the computation times are nearly 100% sufficient for the reconfiguration to take place completely even when the reconfiguration time is 25 microseconds. For the MG benchmark, 75% of the computation times are above 25 microseconds. For the BT and SP benchmarks, near 70% of the computation times are above 10 microseconds. For the LU benchmark, near 50% of the computation times for nodes 0 and 3 are between 10 to 25 microseconds. It is evident that the majority of the reconfigurations can proceed in parallel with the computation and be readied before the end of the computation. For the cases where the computation time is not sufficiently long to completely hide the reconfiguration it effectively reduces the reconfiguration cost by the corresponding length of time.

We are in the process of getting further results on a larger machine to identify what would be the effect of increasing the number of cooperating nodes in the computation on the inter-communication computation times. We are also interested to see what would be the effect of larger problem size (A class relative to W class for the NAS benchmarks) on these computation times.

4.1 Total reconfiguration time and application speedup

In this section, we shall examine the effectiveness of the proposed heuristics. We shall quantify the ability of the proposed heuristics in hiding the reconfiguration delays. In addition, we shall determine the impact of the proposed heuristics on the application speedup.

We use two system models. Both system models assume a multicomputer with modes similar to the thin nodes of an IBM SP2 system but with a reconfigurable optical interconnect which has a reconfiguration delay d (d = 25, 10, 5, 1 μs).

TABLE 1. Minimum inter-send computation times of the parallel benchmarks when N = 4 (all times are in microsecond)

	Node 0	Node 1	Node 2	Node 3
BT (W class)	4.888	4.472	4.472	4.888
BT (A class)	4.992	4.368	4.264	4.472
SP (W class)	4.576	4.264	4.264	4.472
SP (A class)	4.576	4.264	4.264	4.576
LU (W class)	5.824	240.006	600.340	8.840
LU (A class)	5.824	981.734	2506.452	16.224
MG (W class)	5.928	6.240	6.240	7.072
CG (W class)	335.556	336.908	337.700	365.404
CG (A class)	680.316	685.204	684.476	731.796
PSTSWM	0.546	0.325	1.027	0.819
QCDMPI	6.448	1166.880	1167.816	1240.304

For the calculations used to quantify the reconfiguration hiding capabilities of our heuristics, we used the lower bound of the inter-send computation times. This allows us to compute the lower bound of the time that can be hidden. For the application speedup computations though, we have used the full benchmark trace obtained and the inter-send time includes overhead associated with the waiting on receive. The algorithm used to obtain the time spent in reconfiguring the interconnect with and without the prediction heuristics is given by the following pseudocode.

```
total_new_reconfiguration = 0.0;
total_original_reconfiguration = 0.0;

for each inter_send computation time {
   if (hit) then
     if (inter_send_computation < reconfiguration_delay) then
        total_new_reconfiguration += reconfiguration_delay - inter_send_computation;
   else total_new_reconfiguration += reconfiguration_delay;
   total_original_reconfiguration += reconfiguration_delay;

}
```

Figure 16, illustrates the average ratio of the total new reconfiguration delay over the total original reconfiguration delay for each benchmark for two different CPU speeds and four different reconfiguration delays. It is clear that the tag-based heuristics have the best performance enhancements. For the CG, LU, and PSTSWM benchmarks, the granularity of the applications are such that almost all of the reconfiguration delay can be hidden under tag-based heuristics. This is also true for CG and LU benchmarks under other heuristics (except for LRU, FIFO, and LFU). Also, cycle-based heuristics perform relatively well for other benchmarks. It is also evident that the ratios increase with a faster CPU for the same reconfiguration delay. However, the reconfiguration time may also decrease in future. In this respect, it is good to compare the bar graphs in Figure 16 under different reconfiguration delays and processor speeds. Not to mention that these results are still preliminary and we are looking forward to the results of a larger machine size.

In Table 2, we present the maximum application speedup that we can get by applying the heuristics on them. The application speedup has been computed for node 0. The results are consistent with the

fact that the total communication time is much smaller than the total computation time. This is perhaps because of the granularity of these applications and the fact that these results are for only 4 nodes. It is interesting to see what would be the application speedup with a larger machine size and problem size.

TABLE 2. Speedup of the benchmarks when number of processors is 4 and *d* = 25 microseconds (W class for NPB)

	BT	SP	LU	MG	CG	PSTSWM	QCDMPI
Speedup	1.0005	1.0003	1.002	1.006	1.004	1.0001	1.0001

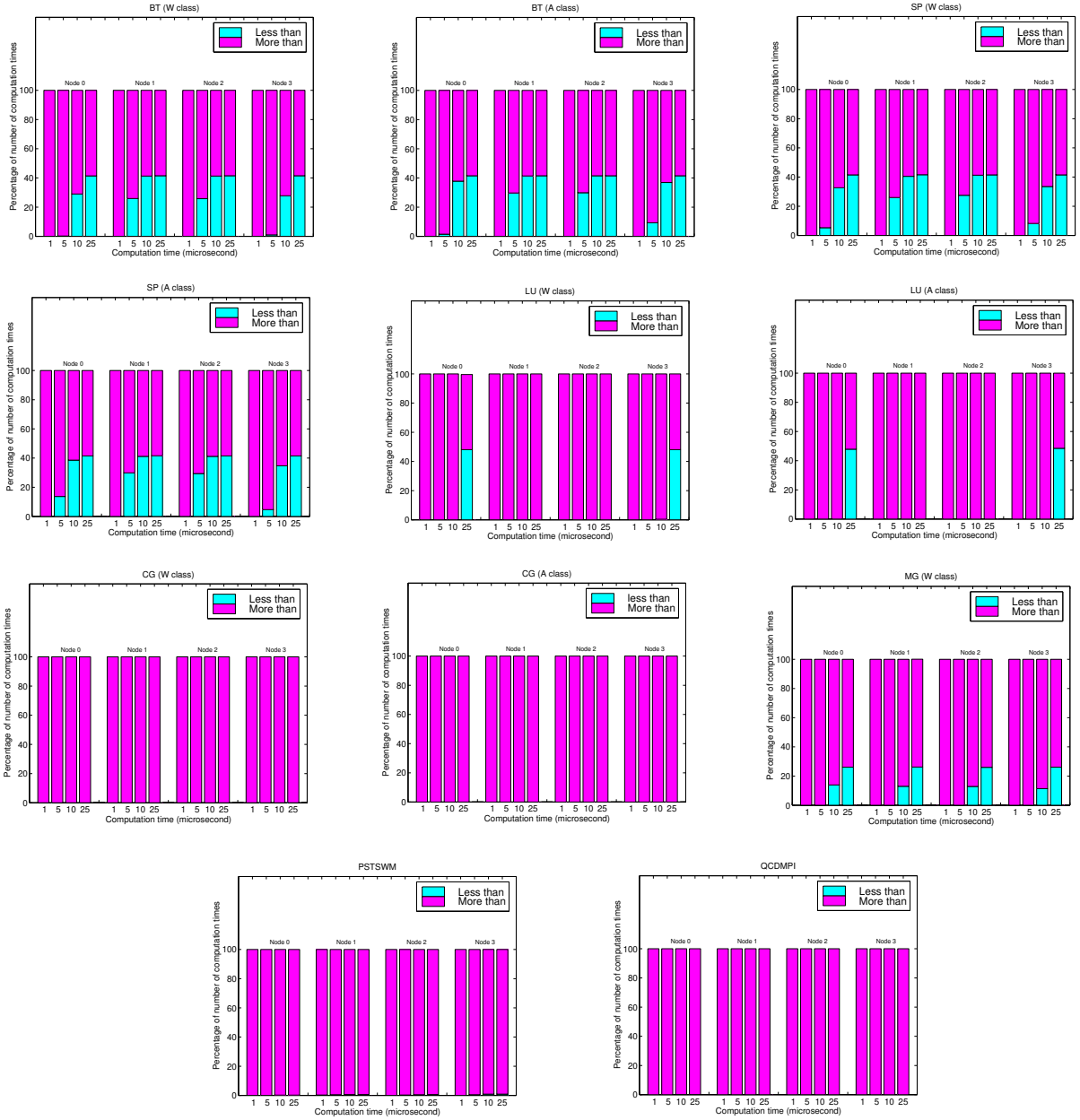


FIGURE 15. Percentage of the inter-send computation times for different benchmarks less than, and more than specific length of times when N = 4 (W and A are classes of NAS parallel benchmarks)

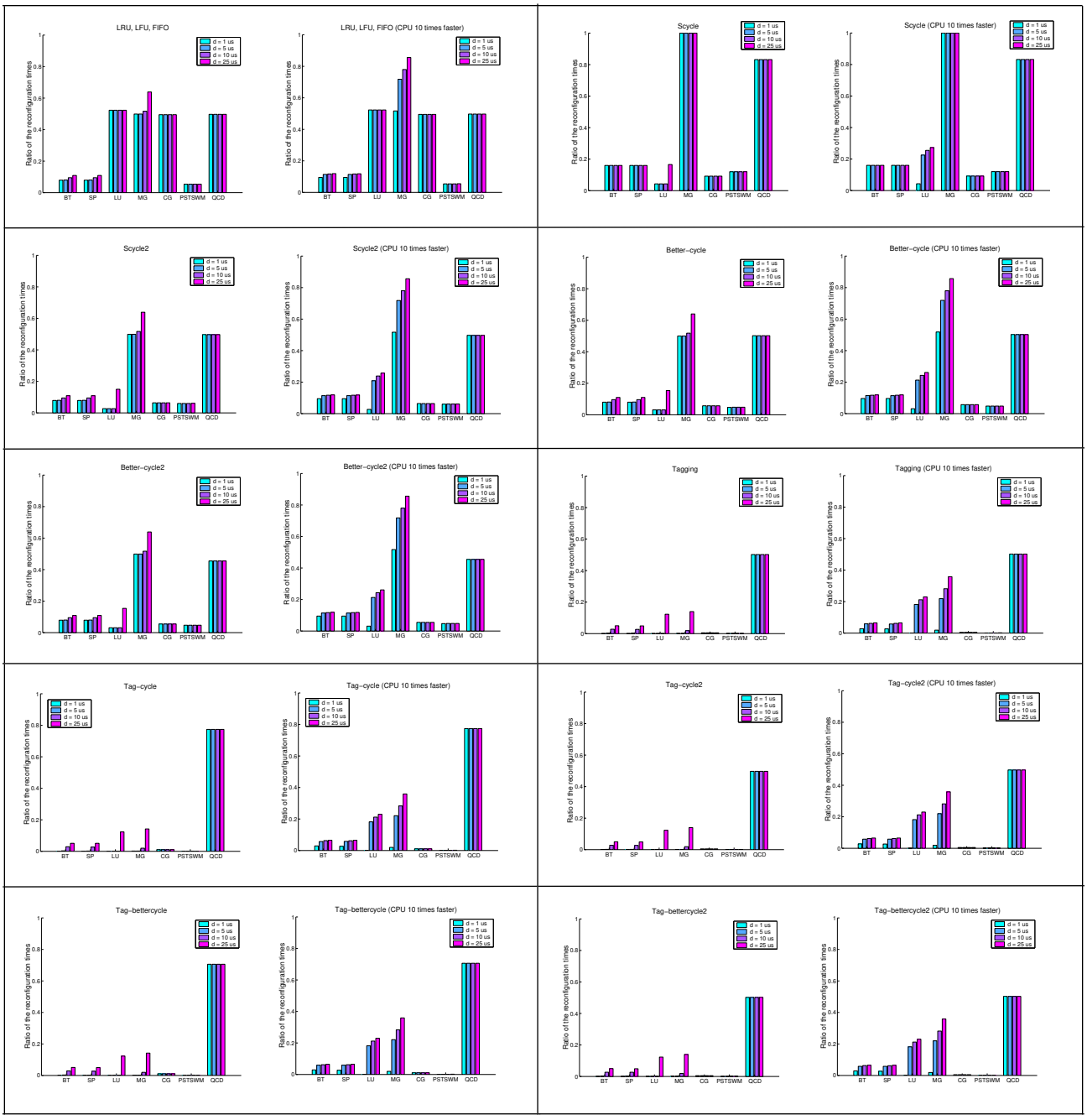


FIGURE 16. Average ratio of the total reconfiguration time after hiding over the total original reconfiguration time for different benchmarks with the current generation and a 10 times faster CPU when d = 1, 5, 10, and 25 microseconds, and N = 4 under single-port modeling

5.0 Conclusion

In the first part of this work, we presented a number of heuristics that can be used to "predict" the target of a communication request before the actual request is issued. These heuristics use the pattern of communications and are designed to extract dependencies which are embedded in these patterns. For these studies, we used the publicly available NAS suite, the PSTSWM and the QCDMPI parallel benchmarks. The results of our studies give strong evidence for the existence of "communications locality" at least in the benchmark programs studied. Our experiments also showed that the nodes in message passing parallel programs (at least in these benchmarks) tend to communicate with only a subset of the other nodes.

The heuristics proposed are only possible because of the existence of *communications locality* that can be used in establishing a communication pathway between a source and a destination before this pathway is to be used. This is a very desirable property since it allows us to effectively hide the cost of establishing such communications links, providing thus the application with the raw power of the underlying hardware (e.g. a reconfigurable optical interconnect).

In the second part of our work, we presented the execution times of the computation phases of these parallel benchmarks on the IBM SP2 using its high performance switch and the user space when we had exclusive access to four nodes of the system. In measuring the execution times of the computation phases we ensured that any system and communication overheads were excluded. In essence, the reported times are the lower bounds of the execution times of the computation times. We presented the percentage of times that these computation times are less than 1, 5, 10, and 25 microseconds. Even for current optical technology [19] (supposably, 25 microsecond case), the results show that we can use most of this time to hide the reconfiguration delay if we use one of the proposed high hit-ratio heuristics. We would like to see what would be the inter-send computation times of these benchmarks when we run them on a larger number of processors.

We also presented the performance enhancements of the proposed heuristics on the total reconfiguration time. For this, we used the obtained computation/communication traces and heuristics hit/miss profiles to determine the total reconfiguration time under different reconfigurable costs and processor speeds. The results indicated that the tag-cycle based heuristics have the best performance.

The speedup results are explainable since the four node system is heavily compute bound. Any improvements on the communication have minimal effect due to Amdahl's law. We are planning to extend our measurements to systems with more than four nodes where the computation and communication are balanced.

Finally, we are confident that the existence of "communications locality" and the resulting latency hiding techniques will usher a new era in interconnection technologies by allowing the use of reconfiguarbility and fast optical fabrics.

Acknowledgments

This work was supported by grants from NSERC and the University of Victoria. Special thanks to the staff of the computer center at the University of Victoria for their kind cooperation and help in accessing the university IBM SP2.

References

1. A. Afsahi and N. J. Dimopoulos, “Collective Communications on a Reconfigurable Optical Interconnect,” Proceedings of the International Conference on Principles of Distributed Systems, Dec., 1997, pp. 167-181
2. A. Afsahi and N. J. Dimopoulos, “Communications Latency Hiding Techniques for a Reconfigurable Optical Interconnect: Benchmark Studies,” Proceedings of the of PARA98, Fourth International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems, June, 1998
3. D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon, “NAS Parallel Benchmark Result 3-94,” Proceedings of the Scalable High-Performance Computing Conference, 1994, pp. 111- 120
4. H. Bourdin, A. Ferriera, and K. Marcus, “A Comparative Study of One-to-Many WDM Lightwave Interconnection Networks for Multiprocessors,” Proceedings of the Second International Conference on Massively Parallel Processing using Optical Interconnections, 1995, pp. 257-263
5. J. Circello, G. Edgington, D. McCarthy, S. Sullivan, D. Schimke, J. Gay, R. Duerden, C. Hinds, D. Marquette, L, Sood, A. Crouch, D. Chow, “The Superscalar Architecture of the MC68060,” IEEE Micro, Volume 15, Number 2 April 1995, pp. 10-21
6. 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
7. J. J. Dongarra and T. Dunigan, “Message-Passing Performance of Various Computers,” Concurrency, Vol. 9, No. 10, Dec. 1997, pp. 915-926
8. 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
9. M. W. Haney and M. P. Christensen, “Fundamental Geometric Advantages of Free-Space Optical Interconnect,” Proceedings of the Third International Conference on Massively Parallel Processing using Optical Interconnections, 1996, pp. 16-23
10. S. Hioki, Parallel Computing, 22-10 (1996), pp. 1335-1344.
11. 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, Feb. 1998, pp. 202-216
12. D. G. de Lahaut and C. Germain, “Static Communications in Parallel Scientific Programs” Proceedings of PARLE’94, Parallel Architecture and Languages, Athen, Greece, July 1994
13. A. Louri and H. K. Sung, "An Optical Multi-Mesh Hypercube: A Scalable Optical Interconnection Network for Massively Parallel Computing," Journal of Lightwave Technology, Vol. 12, No. 4, 1994, pp. 704-716
14. P. J. Marchand, A. V. Krishnamoorthy, G. I. Yayla, S. C. Esener, and U. Efron, “Optically Augmented 3-D Computer: System Technology and Architecture,” Journal of Parallel and Distributed Computing, Special Issue on Optical Interconnects, Feb. 25, 1997, pp. 20-35
15. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. Version 1.1 (June 1995)
16. T. Mowry and A. Gupta, “Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors,” Journal of Parallel and Distributed Computing, 12(2), 1991, pp. 87-106
17. R . A. Nordin, A. F. Levi, R. N. Nottenburg, J. O’Gorman, T. Tanbun-Ek, and R. A. Logan, “A System Perspective on Digital Interconnection Technology,” IEEE Journal of Lightwave Technology, Vol. 10, June 1992, pp. 801-827
18. 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
19. K. Panajotov, N. Nieuborg, A. Goulet, I. Veretennicoff and H. Thienpont, "A Free-space Reconfigurable Optical Interconnection based on Polarization-Switching VCSEL's and Polarization-Selective Diffractive Optical Channels," Proceedings of the Optics in Computing, 1998, pp. 151-154
20. 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
21. 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
22. G. I. Yayla, P. J. Marchand, and S. C. Esener, Speed and Energy Analysis of Digital Interconnections: Comparison of On-chip, Off-chip and Free-Space Technologies, " Applied Optics, Vol. 37, No. 2, Jan. 1998, pp. 205-227
23. X. Yuan, R. Melhem and R. Gupta, "Compiled Communication for All-Optical TDM Networks," Proceedings of the Supercomputing'96, 1996
24. Z. Zhang and J. Torrellas, "Speeding Up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching," Proceedings of the 22nd Annual Symposium on Computer Architecture, 1995, pp. 188-199

Appendix A (IBM Deep Blue results)

We have extended our experimentation on a larger machine to see the inter-send computation times and the effects of the heuristics on the total reconfiguration delay. For this, we used the IBM SP2 Deep Blue machine, a 30 node machine, and run the applications, one process on each node, when we were the only user of this machine. Table 3, shows the minimum inter-send computation times of the application benchmarks on up to 25 nodes of the IBM deep Blue machine. Figure 17, presents the percentage of the inter-send computation times for different benchmarks less than, and more than 1, 5, 10, 25 microseconds for N = 4, 8, 9, 16, and 25. Figure 18, presents the total reconfiguration time obtained while employing heuristics as a percentage of the total configuration time when no prediction heuristics were employed. Figure 19, shows a summary of the Tag-bettercycle2 heuristic. The results are shown for the best benchmarks of each category, that is the Tag-bettercycle2, better-cycle2 and the LRU/LFU/FIFO algorithms. The Benchmarks assumed a current generation and a 10 times faster CPU. The results are consistent with the fact that we can use most of the time to hide the reconfiguration delay with the help of one of the proposed high hit-ratio heuristics. Meanwhile, the Tag-bettercycle2 heuristic shows the best for the performance enhancement of the total reconfiguration time.

TABLE 3. Minimum inter_send computation times (all times in microseconds)

	4 nodes	**8 nodes**	**9 nodes**	**16 nodes**	**25 nodes**
	W A	W A	W A	W A	W A
BT	4.161 4.576	--------	4.161 4.472	4.161 4.472	4.161 4.576
SP	4.161 4.784	--------	4.161 4.472	4.161 4.472	4.161 4.368
LU	9.568 22.568	8.216 12.688	----------	8.112 9.672	---------
MG	6.344 7.592	5.720 7.176	----------	5.928 6.760	---------
CG	407.99 829.92	6.864 7.176	----------	7.384 6.656	---------
PSTSWM	7.176	6.240	----------	6.032	6.344
QCD	1392.352	695.344	----------	353.080	193.128

It is interesting to discover what would be the effect of the heuristics on the receiving sides. Using one of the high hit-ratio heuristics reduces the total reconfiguration delay. When these happen at the sender sides, most of the time the messages are delivered sooner at the receiver sides. If the receive calls have been issued after the corresponding send calls there would be no gain. However, if they are issued earlier then there would be performance enhancement on the receiving side and therefore on the whole execution time. We present the average percentage of the times that the receive calls are issued earlier than their corresponding send calls for the CG, SP, and PSTSWM benchmarks, in Figure 20. For these, we synchronized the timing traces of each node of these applications. These ratios present the worst case scenario for the receive side improvements. We would like to discover the performance enhancements of the heuristics (applying on the sender sides) for the receiver sides using an event-driven simulator.

Acknowledgments

We would like to thank Dr. Murray Campbell at the IBM TJ Watson Research Center and Greg Schick at the IBM Victoria for their kind help in accessing the IBM Deep Blue.

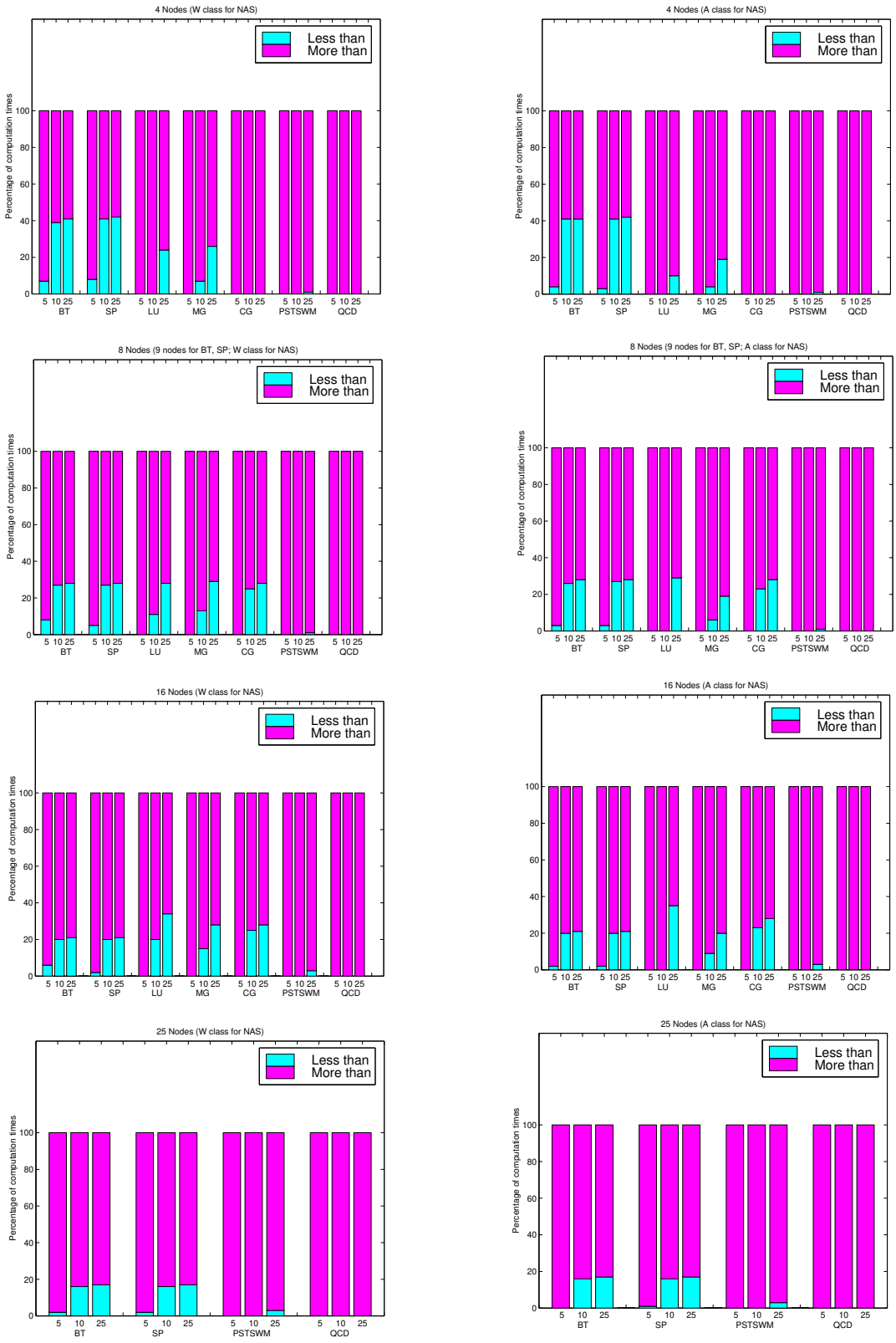


FIGURE 17. Percentage of the inter-send computation times for different benchmarks less than, and more than specific length of times when N = 4, 8, 9, 16, and 25.

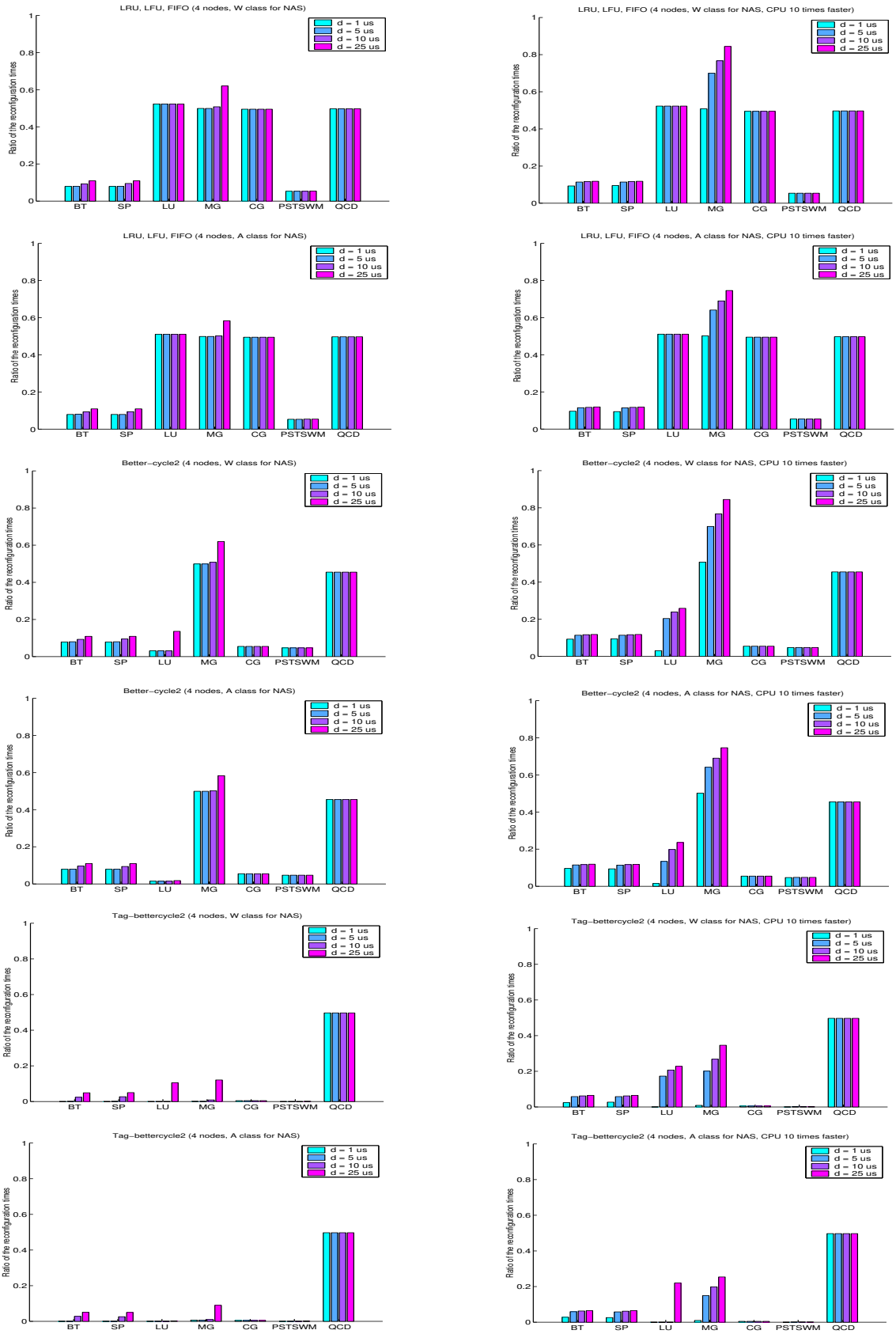


FIGURE 18. Average ratio of the total reconfiguration time after hiding over the total original reconfiguration time for different benchmarks with the current generation and a 10 times faster CPU when d = 1, 5, 10, and 25 microseconds

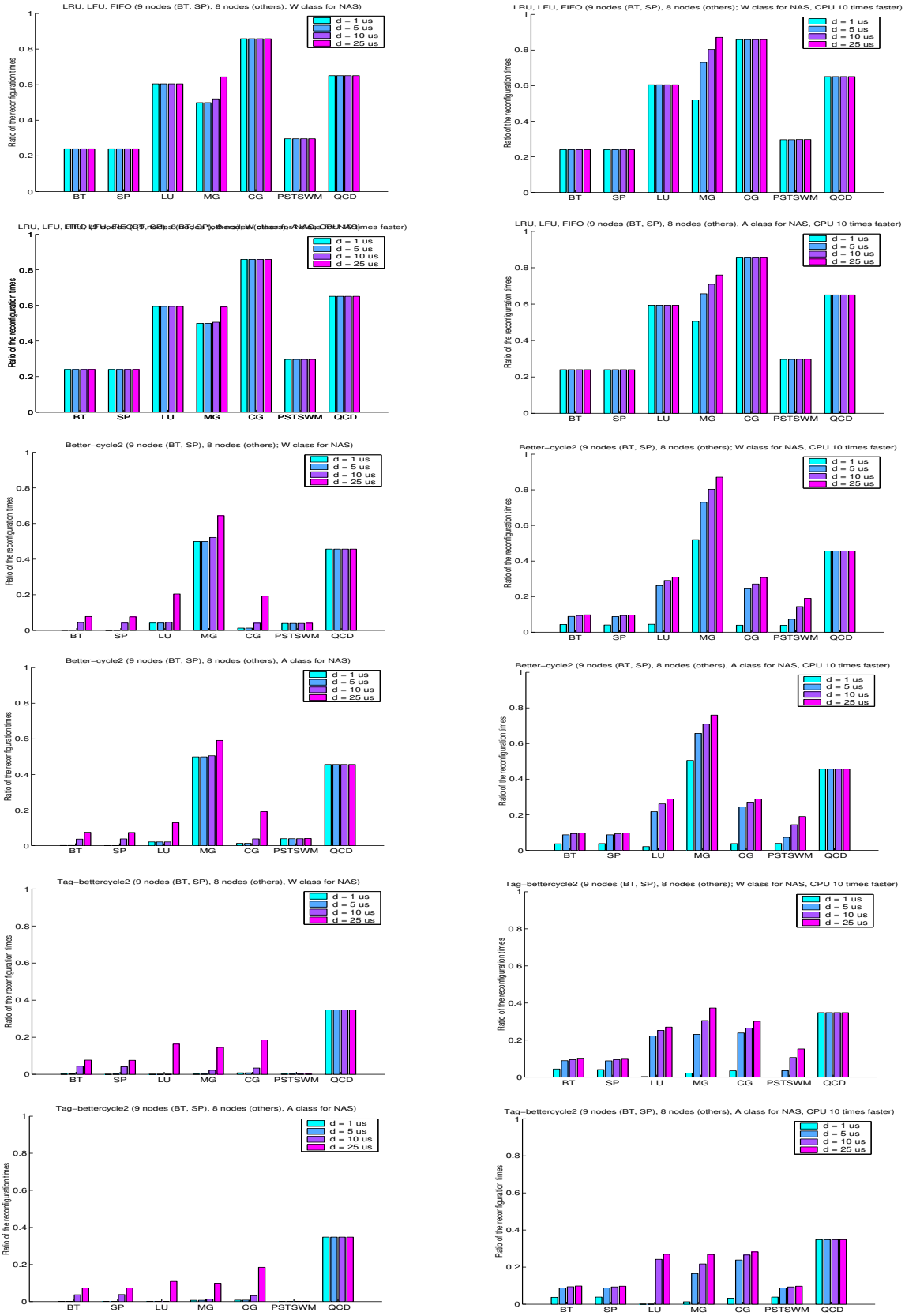


Figure 18 (continued)

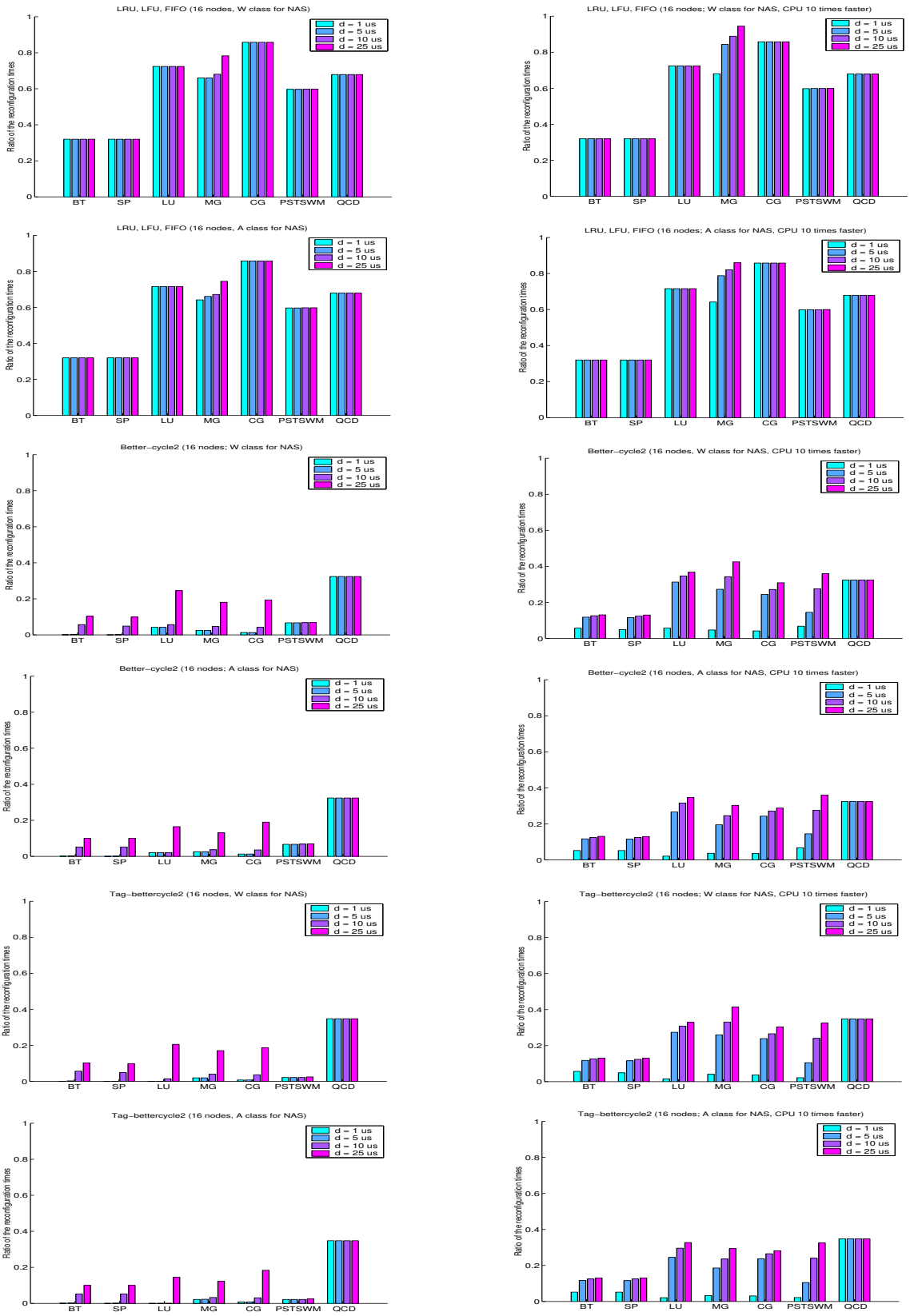


Figure 18 (continued)

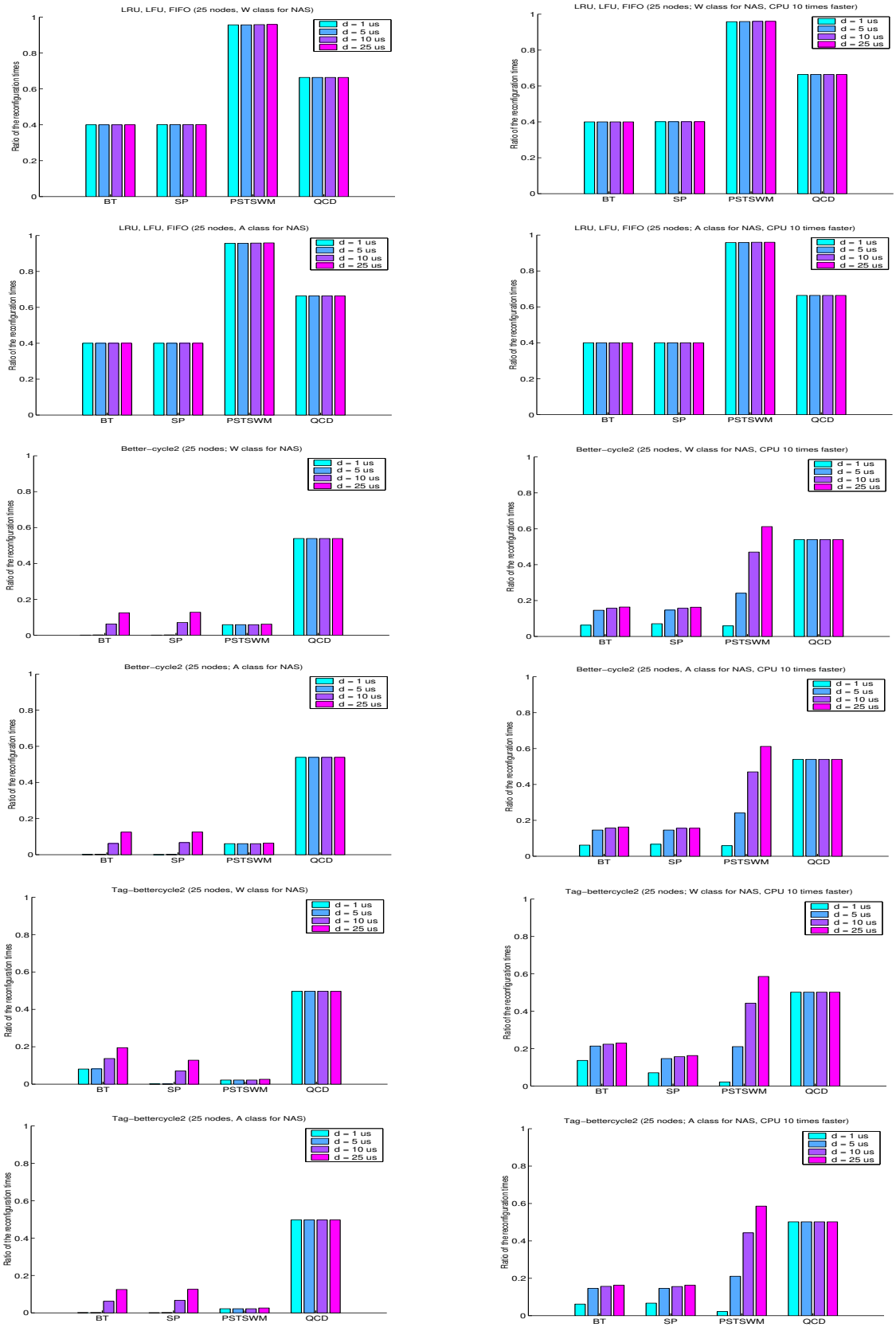


Figure 18 (continued)

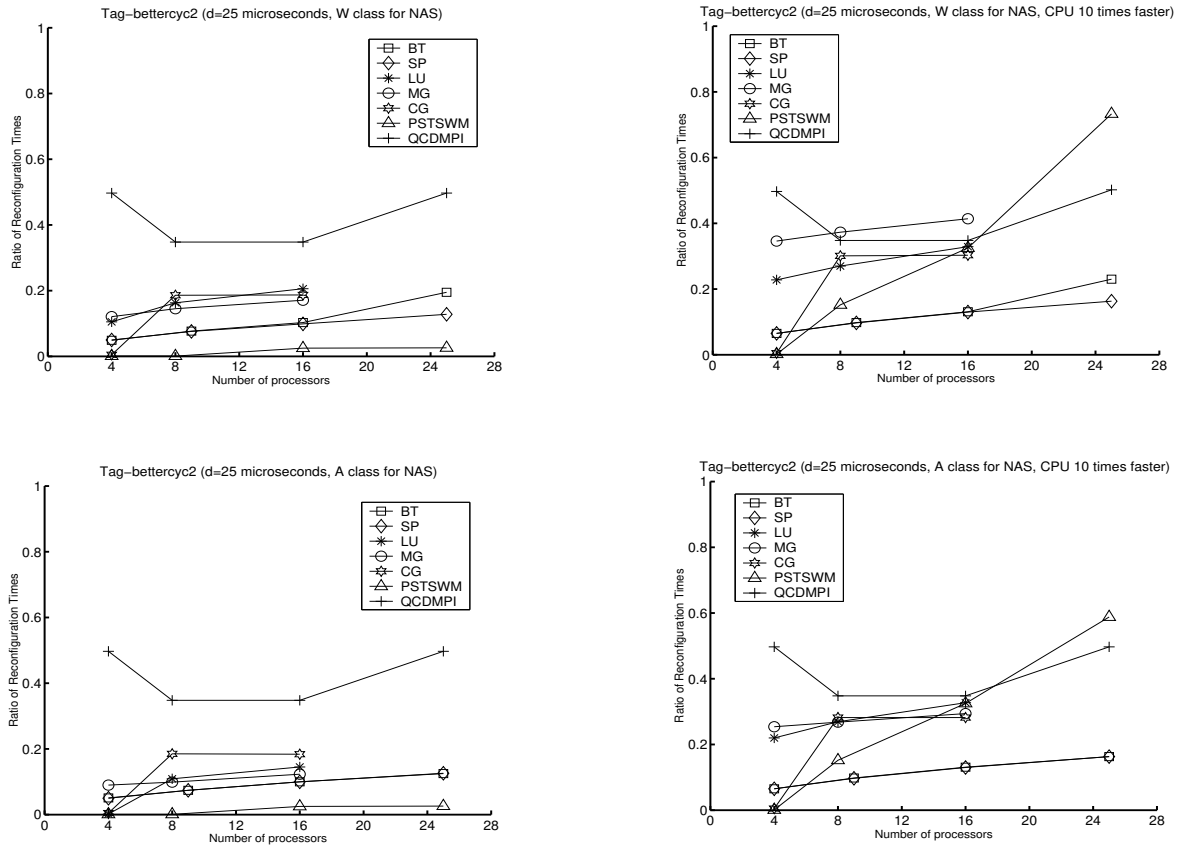


FIGURE 19. Summary of the ratio of the total reconfiguration times when applying the Tag-bettercycle2 heuristic on the benchmarks for $d = 25$ microseconds

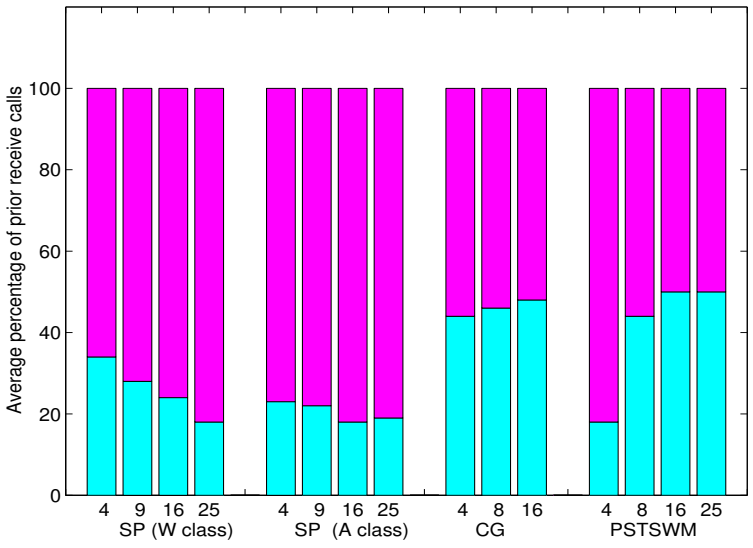


FIGURE 20. Average percentage of the times the receive calls are issued before the corresponding send calls

Appendix B (timing analysis)

In this section, we explain how the pure intercommunication computation times have been computed. Each inter-send computation time is the computation time between two succesive communication operations (send operations). In the following example, t_3 - t_2 is the computation time between two successive *mpi_send* operations where t_3 is the time just before the second call is issued while t_2 is the time just after the first send call finishes.

t_1 *mpi_send* (buff, length, dest, tag); t_2

... computation ...

t_3 *mpi_send* (buff, length, dest, tag); t_4

The example above has no other MPI calls between the two send primitives. In cases that other MPI calls exist between succesive send calls, we have to take out these extra times to obtain the pure inter_send computation times. In the following, two other MPI calls (*mpi_recv* and *mpi_wait*) exist. Therefore, the pure computation time is equal to $t_7 - t_2 - ((t_4 - t_3) + (t_6 - t_5))$.

t_1 *mpi_send* (buff, length, dest, tag); t_2

... computation ...

t_3 *mpi_recv* (buff, length, source, tag); t_4

... computation ...

t_5 *mpi_wait (tag);* t_6

... computation ...

t_7 *mpi_send* (buff, dest, length, tag); t_8

To get the pure inter-send computation times, we need to know the exact times before and after each MPI call. For these, we did not insert the *mpi_wtime* call in the source codes of the applications but instead we wrote our own profiling code to gather the timing traces. Hence, each MPI call in the applications calls its own profiling code, as shown in the following example for the *mpi_send*.

Application benchmark:

```
...
...
ta mpi_send (buff, length, dest, tag)  tf
...
```

Profiling code:

```
tb  start_time [index] = mpi_wtime(); tc
    pmpi_send (buff, length, dest, tag);
td  end_time [index] = mpi_wtime(); te
(i)   index++;
(ii)  label = k;
      return;
```

The *mpi_wtime* calls give the times before and after the profiling *mpi_send* (*pmpi_send*) call. It is clear that there are overheads entering and exiting the profiling code in adddition to the overhead of instructions *i* and *ii*. The times t_c - t_b and t_e - t_d are the times taken for the first and second *mpi_wtime* calls, respectively. The times t_b - t_a and t_e - t_f are the other overheads. Therefore, we need to exclude the sum of the overheads t_c - t_a and t_f - t_e associated with each MPI call.

The overhead for each type of MPI calls used in the benchmarks can be computed by using the corresponding two consecutive MPI calls and the following profiling code. The folllowing is an example for the *mpi_send* call.

Profiling code:

```
...                                   start_time [index] = mpi_wtime();  t1 , t3
mpi_send (buff, length, dest, tag)    end_time [index] = mpi_wtime();  t2 , t4
mpi_send (buff, length, dest, tag)    index++;
...                                   label = k;
                                      return;
```

The times t_1, t_2 and t_3, t_4 are the times returned by *mpi_wtime* calls for the first and second *mpi_send* calls, respectively. Hence, the extra overhead for the *mpi_send* call is equal to t_3 - t_2. We computed these extra overheads for each type of the MPI calls used in our applications and took them out to find the pure inter-send computation times.

We have used the following strategy for discovering the number of times that the receive calls are issued earlier than their corrsponding send calls. We have considered the times just before the send and receive calls are issued. In case of blocking and non-blocking send calls, the time just before the calls (*mpi_send* and *mpi_isend*) have been taken into account. That is the time that the message is ready to be sent over. For the blocking receive call (*mpi_recv*), we did the same. That is the time that the receiver is ready to get the message. However, for the non-blocking receive call (*mpi_irecv*), we consider the time when the wait call (*mpi_wait*) is issued for the corresponding receive call (*mpi_irecv*). This gives us the worst case scenario of the number of times the receive calls are issued before their corresponding send calls.