JET PROPULSION LABORATORY INTEROFFICE MEMORANDUM

SFB: 363-89-005

17 August 1989

TO: Time Warp Folks

FROM: Steven Bellenot

Why is Pucks Lazy?

The application/benchmark "Pucks" performs significantly better using lazy cancellation than using aggressive cancellation. This memo attempts to explain why. The reasons are partly that Pucks takes better advantage of lazy cancellation than either STB88 or Warp Net and partly that Pucks with aggressive cancellation is overloading the message passing system in TWOS. Some conjectures as to why lazy cancellation works so well on the Pucks simulation are given at the end. Our format is a large number of figures (at the end) with (mostly) just a few words about each.

How Lazy?

That Pucks under lazy was roughly twice as fast as Pucks under aggressive on the 40 to 75 node range was documented in my second memo on Cancellation Policies (JPL IOM SFB: 363-89-004). Table 1 shows the relative ratio aggressive run time over lazy run time for two collection of runs. The top row are from the memo cited above and the bottom row were made after tuning the queueing parameters for aggressive cancellation. Figure 12 contains the execution times for the bottom row of Table 1 and Figure 13 shows this data as a speed-up curve. Lazy is twice as fast as aggressive on 75 nodes, even after the parameters have been tuned for the aggressive policy.

Number of Nodes	40	45	50	55	60	65	70	75
ma = 2 / rcvq = 30	1.71	1.97	2.19	1.94	1.93	2.00	2.49	2.70
ma = 1 / rcvq =100	1.49	1.59	1.62	1.52	1.56	1.62	1.84	1.98

Table 1. (the aggressive execution time with parameters given) divided by (the lazy execution time with the parameters ma=2 rcvq = 30)

Tuning The Parameters

The parameters we varied were (ma) max_acks and (rcvq) the receive queue length (also called (mna) max_neg_acks), also we varied the radius of the pucks which is a simulation parameter. The baseline was with ma=2, rcvq = 30 and radius = 6. This choice of ma and mna came from a study of the behavior of aggressive cancellation on STB88. (See my first memo on cancellation policies (JPL IOM SFB: 363-89-002).) We ran tests with 70 nodes and 30 nodes (35 nodes for the radii test).

Max Acks

The first set of tests varied max_acks. In this version of TWOS on the butterfly there are 64 buffers on each node for sending off-node messages. Those buffers numbered less than max_acks are used for positive messages and the others are used for negative messages and system messages. When max_acks is small, an off-node positive message may have to wait for a buffer to free before it is transmitted. The effect of a small max_acks is to limit the number of positive messages in the FIFO DualQueue. A small max_acks doesn't limit negative messages.

The 70 node tests indicate that max_acks should be as small as possible for the aggressive case (see Figure 1). Figure 1 also shows max_acks had hardly any effect on the execution time for the lazy case. (Actually for the lazy case, max_acks = 1 was slower than max_acks = 2. Also max_acks = 4 might be slightly faster than max_acks = 2 in the lazy case.) Figure 2 shows the number of negative messages sent for these runs. Note that the run times are proportional to the number of anti-messages sent for the aggressive case. The max_acks = 32 data point is missing for aggressive since we didn't obtain a successful run.

We checked the other end of our range with a 30 node test. Again (see Figure 3) max_acks = 1 was the best for the aggressive case. The range is not as great by the execution time again follows the number of anti-messages sent (see Figure 4). The lazy case is the same as for 70 nodes. Tests with max_acks larger than eight were not run on 30 nodes.

Receive Queue Length

The rcvq is the maximum number of messages which can be waiting to be enqueued into the Time Warp queues. Each time through the main loop, Time Warp reads everything off the DualQueue until this limit is reached. In this version, but not TWOS 2.1 or TWOS 2.2, the receive queue is in timestamped order. At least one message from the receive queue is enqueued into the Time Warp queue. Messages from the receive queue are enqueued into the Time Warp queues until there is a object with a strictly smaller timestamp ready to run (or the receive queue is empty.) Note that up to rcvq - 1 messages could be left in the receive queue when TWOS starts running an object, and hence the next time through the main loop Time Warp would dequeue at most one message off the DualQueue. (TWOS currently puts anti-messages ahead of positive messages in the receive queue.)

A large rcvq will tend to empty the FIFO DualQueue and order the messages by the better Time Warp order. Unfortunately, the receive queue compleat for memory with everything else. A large rcvq on a small number of nodes can cause memory exhaustion. A small rcvq can leave more important messages on the DualQueue. If each node has one message in transit to node A, then the length of the receive queue must be the number of nodes minus one in order to always get the message furthest behind. In practice, a much smaller number will do for the length of the receive queue. However nodes can have several messages in transit to node A.

The results of the 70 node tests for rcvq are rather boring (see Figure 5.) It looks like the rcvq is not important at 70 nodes. (We will see later that 100 is just too small when max_acks is 2.) For the lazy case, rcvq = 30 is better than rcvq = 10, but rcvq = 50 might be slightly faster than both. The results for the 30 node case (Figure 6) show that rcvq can be too large for the aggressive case (at least with ma = 2 and a small number of nodes), but for the aggressive case, the run times are decreasing with increasing rcvq. Eventually these tests were re-run with max_acks set to 1. The results (Figures 7 and 8) show a decreasing run time for aggressive pucks as the length of the receive queue increases. For unimportant reasons, there is a maximal limit of 100 on rcvq, which is the best for the aggressive case. Clearly there are a lot longer receive queues for the aggressive case, TWOS is not consuming the messages in the DualQueue fast enough.

Puck Radius

The configuration file used for the test runs has 128 sectors and 128 pucks. It was suggested that perhaps there were not enough collusions in this configuration of Pucks to be a good test of the lazy vs aggressive issue. The effect of increasing the radius (a config file parameter) would be to increase the number of collusions. It was thought that this would improve the relative performance of aggressive vs lazy. The tests were run with ma = 2 and rcvq = 30. Figure 9 (for 70 nodes) and Figure 10 (for 35 nodes) show that aggressive gets worse as the radius increases. Figure 11 shows the number of committed events and committed event messages for the various radii. Some caution for the radius = 12 or 24 case is needed. The puck must be told about all the sectors it starts in, otherwise it can get lost. The configuration files for radius 12 and 24 didn't check for this. Also at the beginning, each puck checks to see if it is completely on the board. If part of the puck "sticks out", the "puck" screams a message to standard error. The radius = 24 case had a few of these screams and these standard error messages add about 10 seconds to each run time.

Timing Runs

Thus ma = 1 and rcvq = 100 (and radius = 6) were used in our timing runs. As Table 1 shows, this tuning of parameters did greatly aid the aggressive run time of pucks. Figure 12 shows the run time data and Figure 13 shows the data as a speed-up curve. (The sequential run time of 2299.01 seconds was taken from the 2.1 benchmarks which was a (slightly?) different version of Pucks.) As before each data point is a average of three runs, differences of a second or two are not significant. Thus we have:

Conclusion 1: Pucks runs much faster with lazy cancellation than with aggressive cancellation.

Conclusion 2: Part of this difference is due the message system being overloaded with aggressive cancellation.

Pucks is More Lazy than STB88 or Warp Net

Another way to check how lazy Pucks can be is to compare it to our other benchmarks, namely STB88 and Warp Net. Table 2 contains the sequential run time of these simulations and the number of committed messages sent. We see that Warp Net sends far fewer messages. The committed messages per second is nearly the same for Pucks and STB88, but they reverse order between the sequential run and the 40 node runs. (The 40 node times were done with lazy cancellation.)

Application	seq-run-time	cemsgs	msg/sec	40 node time	msg/sec
Pucks	2299 seconds	416673	181.24	194 seconds	2147.8
STB88	3756 seconds	603472	160.67	228 seconds	2646.8
Warp Net	2381 seconds	45319	19.03	100 seconds	453.19

Table 2. Comparing applications

We also normalized the number of messages sent for each application by expressing them as a ratio with the number of committed event messages. Figure 14 shows lazy anti-message production for the three applications. Warp Net and STB88 are roughly the same with Pucks being around 50% higher than the other two. Figure 15 shows lazy unsent messages, the ratio of "lazy hits". Again STB88 and Warp Net are closer to each other than to Pucks. Indeed Pucks is about twice the size of the other two. Figure 16 shows aggressive anti-message production, here Pucks has more than twice the ratio of negative messages sent than either STB88 or Warp Net. (Note Warp Net is also more lazy than STB88 by this measure.) Thus we have empirically shown:

Conclusion 3: Pucks takes better advantage of lazy cancellation than does STB88 or Warp Net.

The Lazy Simulation features of Pucks

Just what feature of Pucks makes it so lazy? This section of the memo is conjecture.

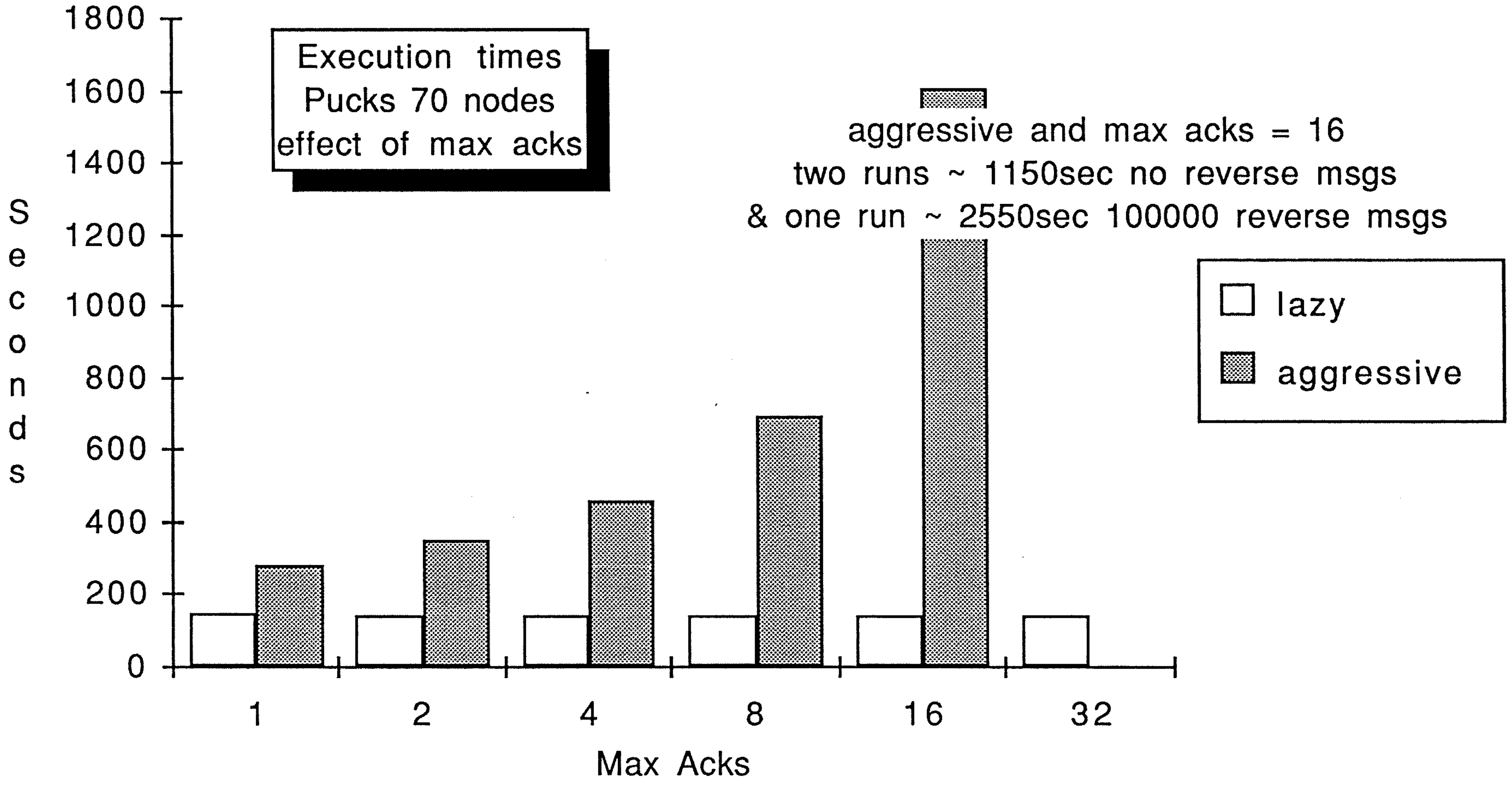
When two pucks are in the same sector at the same time, they will not collide more often than they will collide. In this case if one of the pucks is delayed, the other puck correctly races ahead. This kind of simulation independence is a win for lazy cancellation. It was this idea which lead to the radius test. By increasing the radius, it is more likely that the pucks will collide. Unfortunately, also by increasing the radius it is more likely that non-colliding pucks will share the same sector. We do not have the data to determine which dominates.

Another way in which Pucks is lazy is the large number of message "sinks". Most messages do not cause more messages, the main exceptions being the "change velocity" and "enter sector" messages for a sector. A puck generally will enter at most one of the neighbors of its sector, and as before, collisions between pucks is the exception and not the rule. If a new trajectory message is wrong, most nearby objects will not go down wrong paths.

A third way in which Pucks is lazy is the way in which it uses cancellation. Even if puck A thinks it will collide with puck B at time now + 10.0, it will schedule a collision with puck C at time now + 20.0. The collusion with puck B will cancel the collision with C. The new trajectory message from B with which A schedules the collision for now + 10.0 must arrive after now + 20.0 (and not now + 10.0) in order to generate incorrect results. Thus events can arrive a bit out of order without incorrect work being done.

What Next?

Two paths for further study suggest themselves. First Time Warp could be changed to react faster to messages to remove the queuing effects. It seems that this would require some serious re-thinking, receive queue lengths of more than 100 are likely to kill performance. Second Pucks could collect statistics to see if any of the conjectures above are true.



1800
1600
1400
1200
1000
800
600
400
200
0
Seconds
Execution times
Pucks 70 nodes
effect of max acks
aggressive and max acks = 16
two runs ~ 1150sec no reverse msgs
& one run ~ 2550sec 100000 reverse msgs
lazy
aggressive
1
2
4
8
16
32
Max Acks

Figure 1

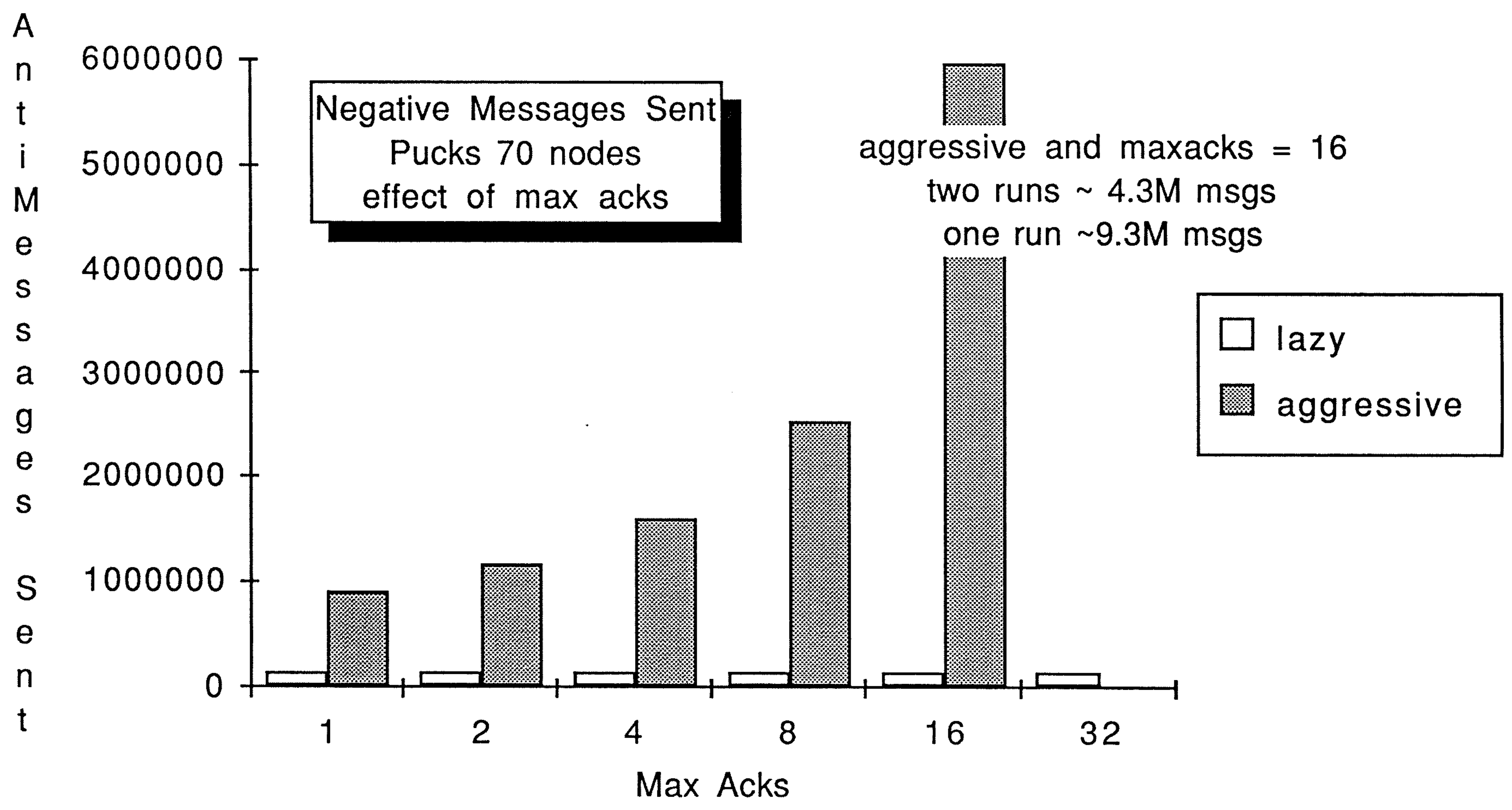
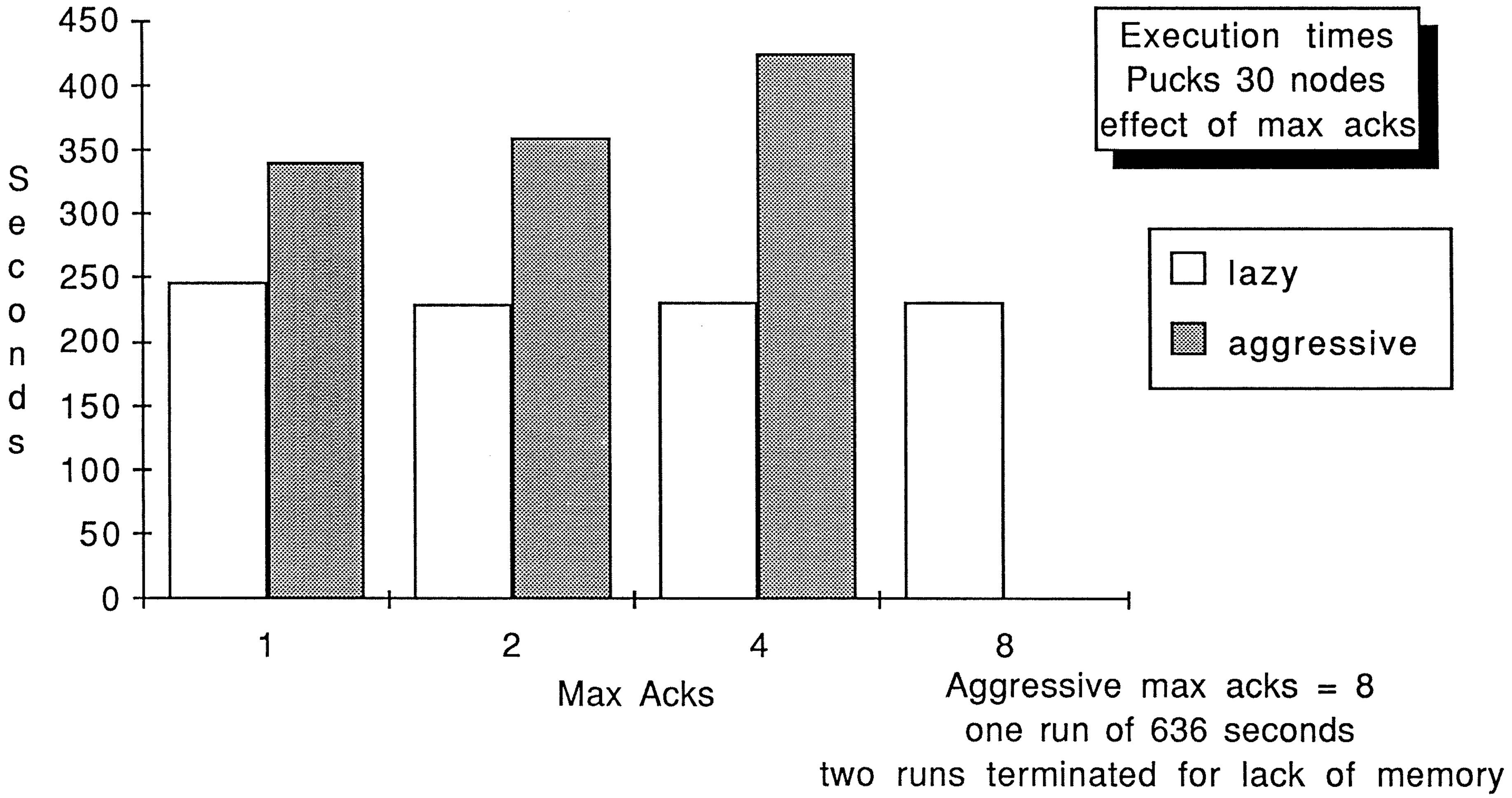

Anti Messages Sent
6000000
5000000
4000000
3000000
2000000
1000000
0
Negative Messages Sent
Pucks 70 nodes
effect of max acks
aggressive and maxacks = 16
two runs ~ 4.3M msgs
one run ~9.3M msgs
lazy
aggressive
1
2
4
8
16
32
Max Acks

Figure 2



Execution times
Pucks 30 nodes
effect of max acks
□ lazy
▨ aggressive
Seconds
450
400
350
300
250
200
150
100
50
0
1
2
4
8
Max Acks
Aggressive max acks = 8
one run of 636 seconds
two runs terminated for lack of memory

Figure 3

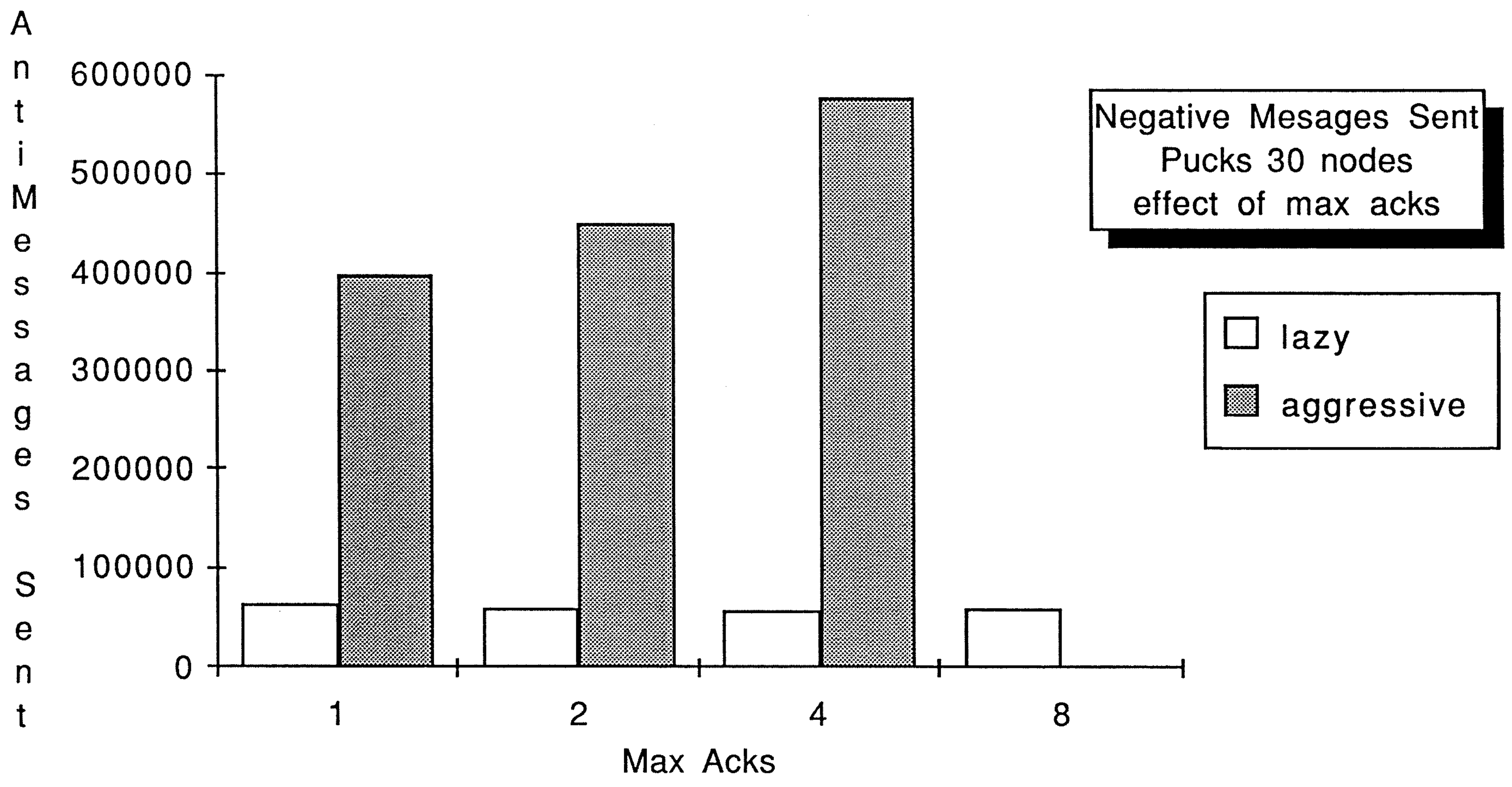
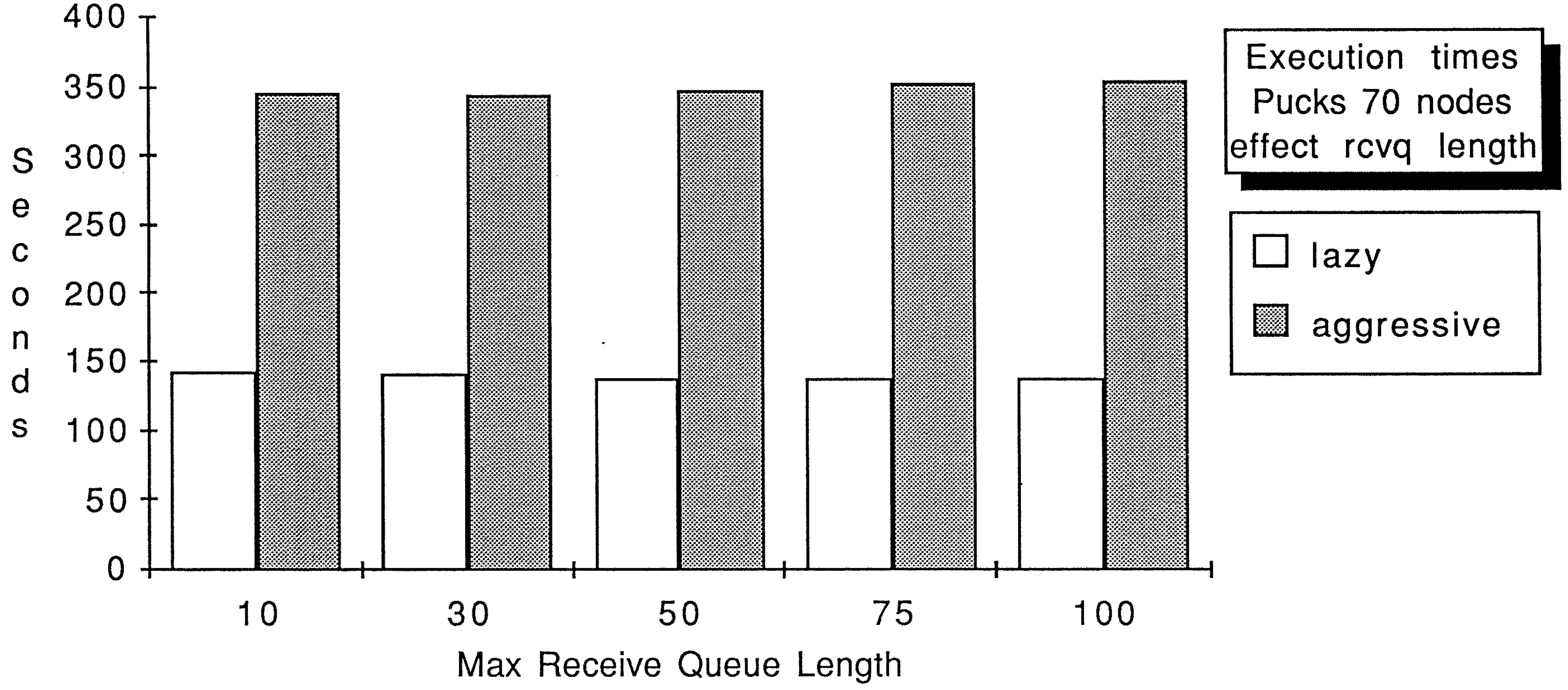

Anti Messages Sent
600000
500000
400000
300000
200000
100000
0
1
2
4
8
Max Acks
Negative Mesages Sent
Pucks 30 nodes
effect of max acks
lazy
aggressive

Figure 4



Execution times
Pucks 70 nodes
effect rcvq length
□ lazy
▨ aggressive
Seconds
0
50
100
150
200
250
300
350
400
10
30
50
75
100
Max Receive Queue Length

Figure 5

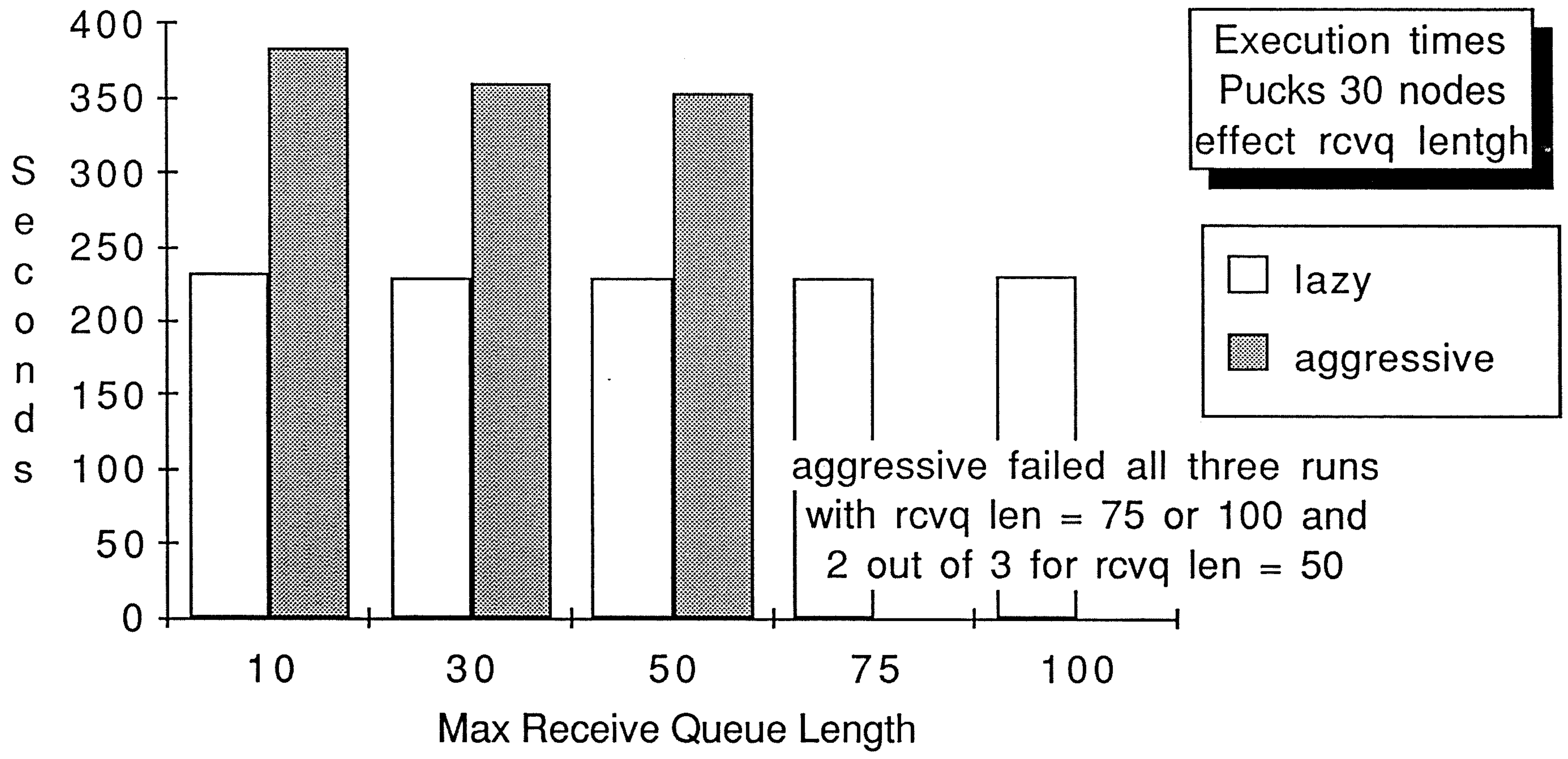

Execution times
Pucks 30 nodes
effect rcvq lentgh
lazy
aggressive
Seconds
400
350
300
250
200
150
100
50
0
10
30
50
75
100
Max Receive Queue Length
aggressive failed all three runs
with rcvq len = 75 or 100 and
2 out of 3 for rcvq len = 50

Figure 6

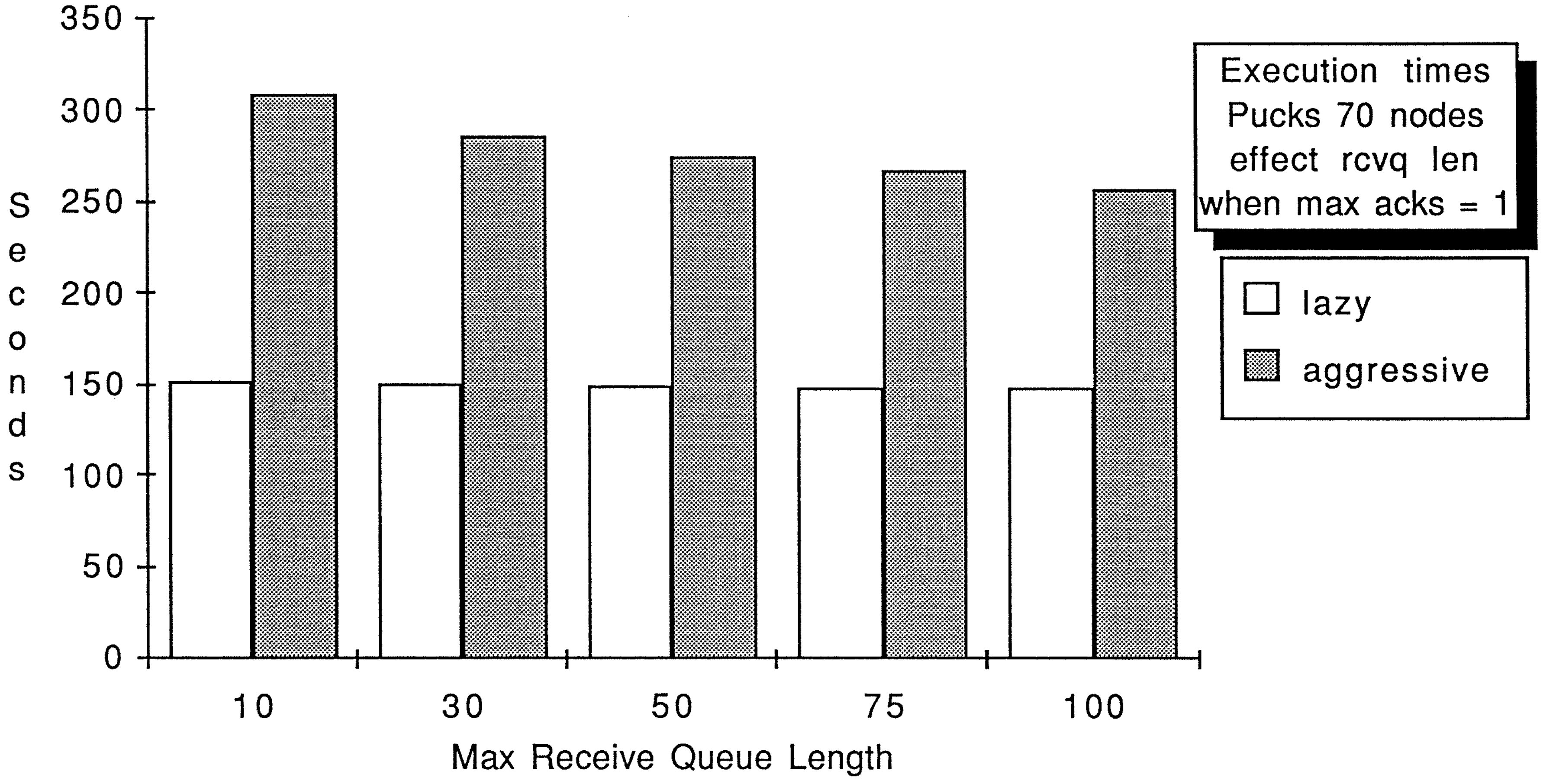

Execution times
Pucks 70 nodes
effect rcvq len
when max acks = 1
lazy
aggressive
Seconds
350
300
250
200
150
100
50
0
10
30
50
75
100
Max Receive Queue Length

Figure 7

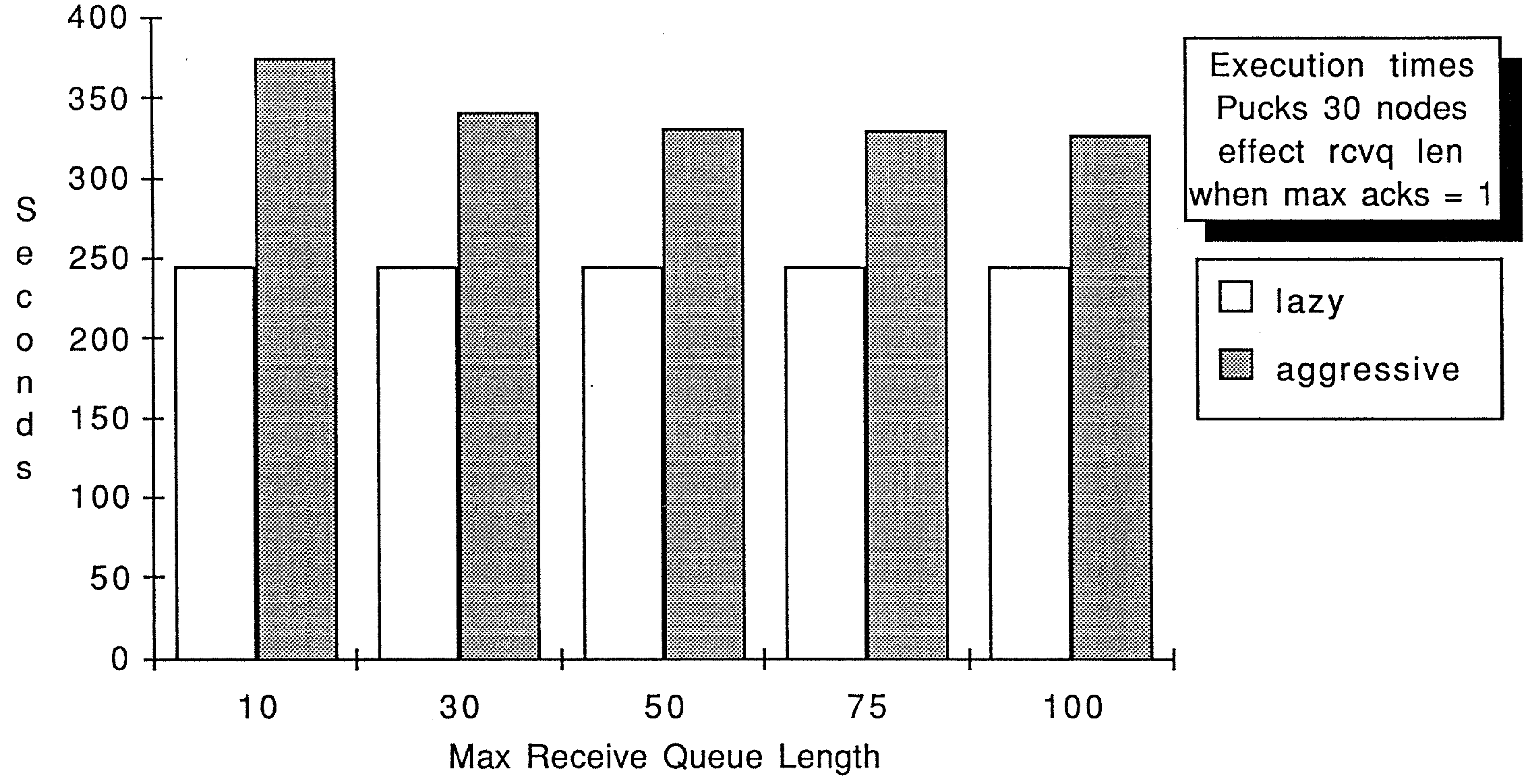

400
350
300
250
200
150
100
50
0
Seconds
10
30
50
75
100
Max Receive Queue Length
Execution times
Pucks 30 nodes
effect rcvq len
when max acks = 1
□ lazy
▨ aggressive

Figure 8

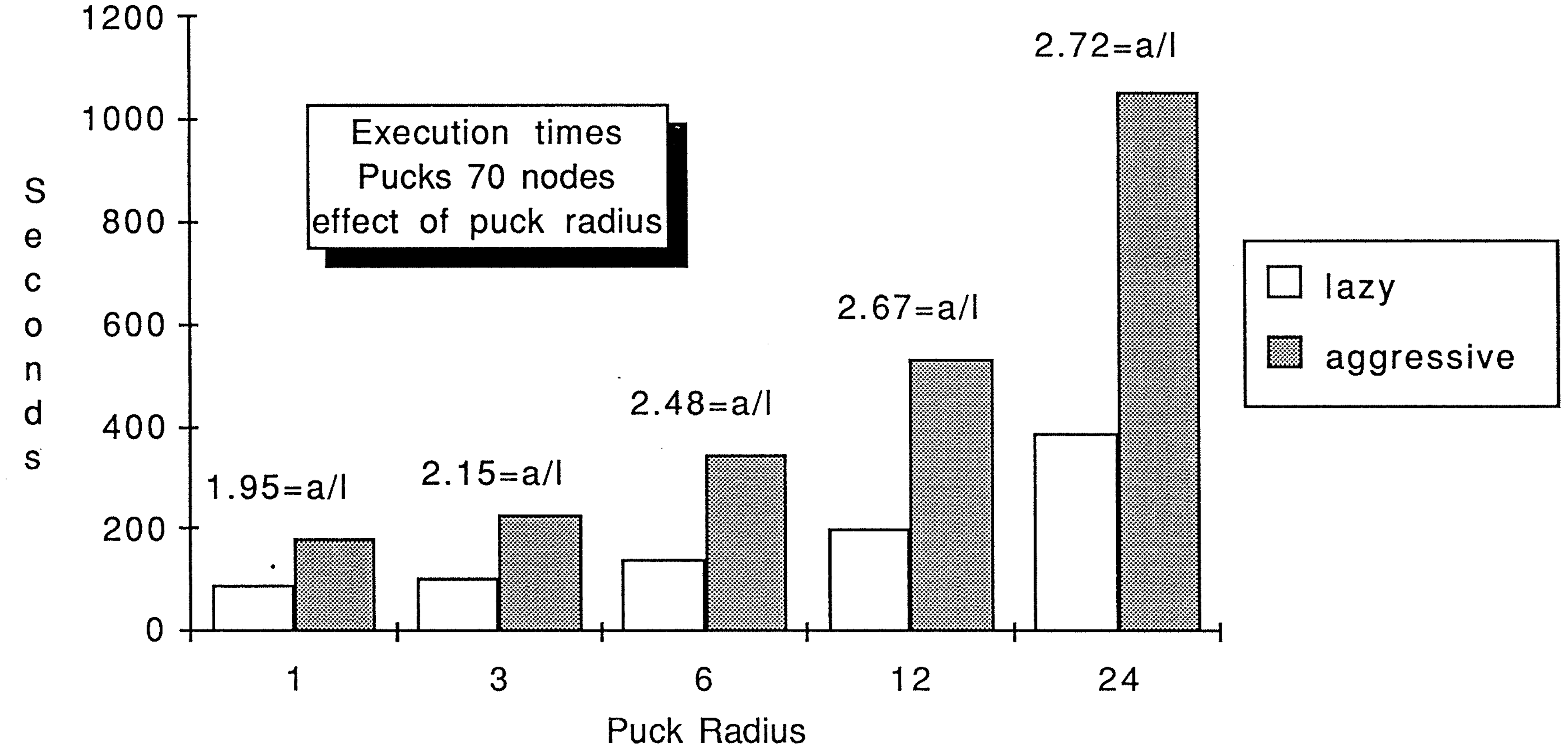

1200
1000
800
600
400
200
0
Seconds
Execution times
Pucks 70 nodes
effect of puck radius
1.95=a/l
2.15=a/l
2.48=a/l
2.67=a/l
2.72=a/l
□ lazy
▨ aggressive
1
3
6
12
24
Puck Radius

Figure 9

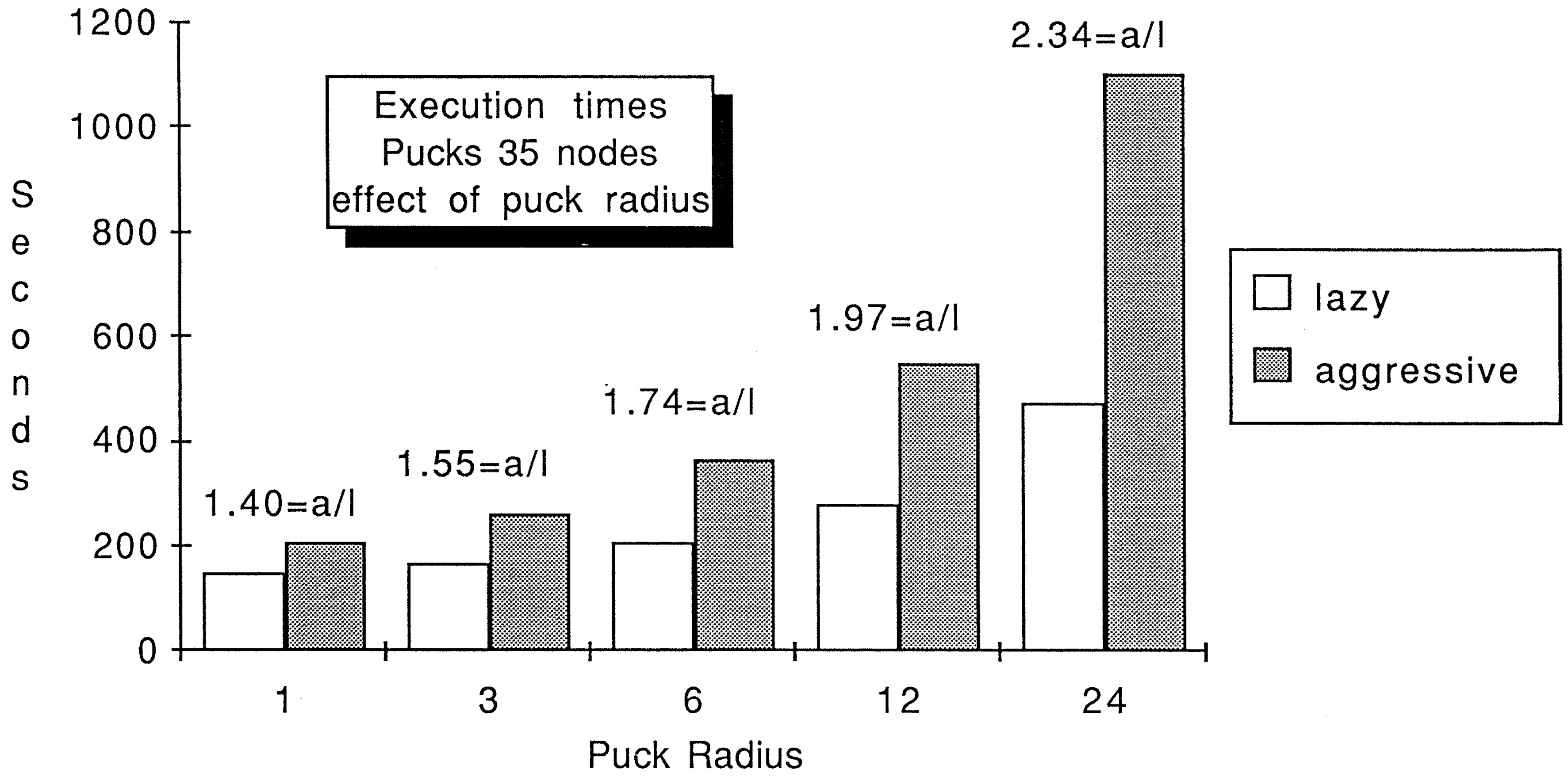

Execution times
Pucks 35 nodes
effect of puck radius
Seconds
1200
1000
800
600
400
200
0
1.40=a/l
1.55=a/l
1.74=a/l
1.97=a/l
2.34=a/l
1
3
6
12
24
Puck Radius
lazy
aggressive

Figure 10

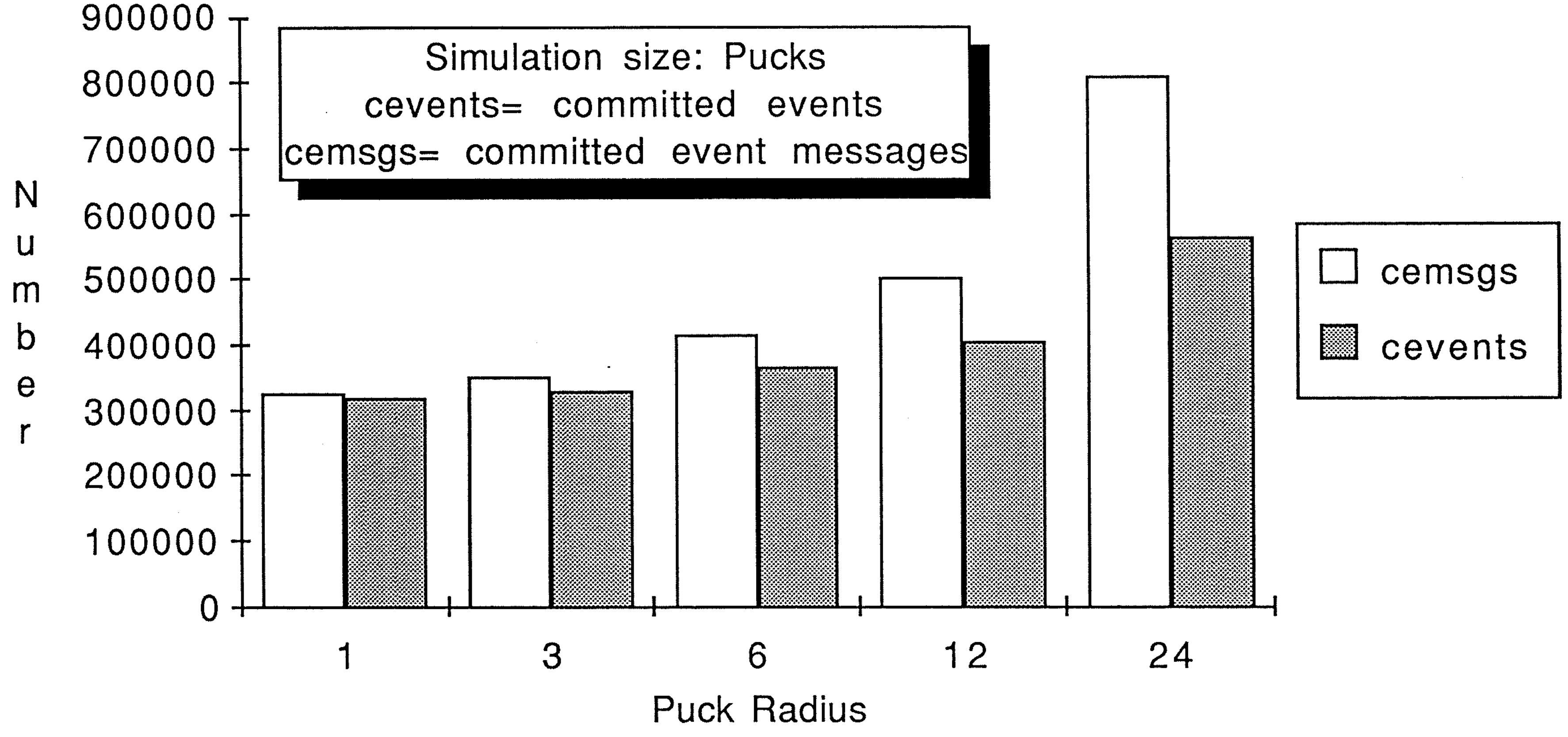

Simulation size: Pucks
cevents= committed events
cemsgs= committed event messages
cemsgs
cevents
Number
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
1
3
6
12
24
Puck Radius

Figure 11

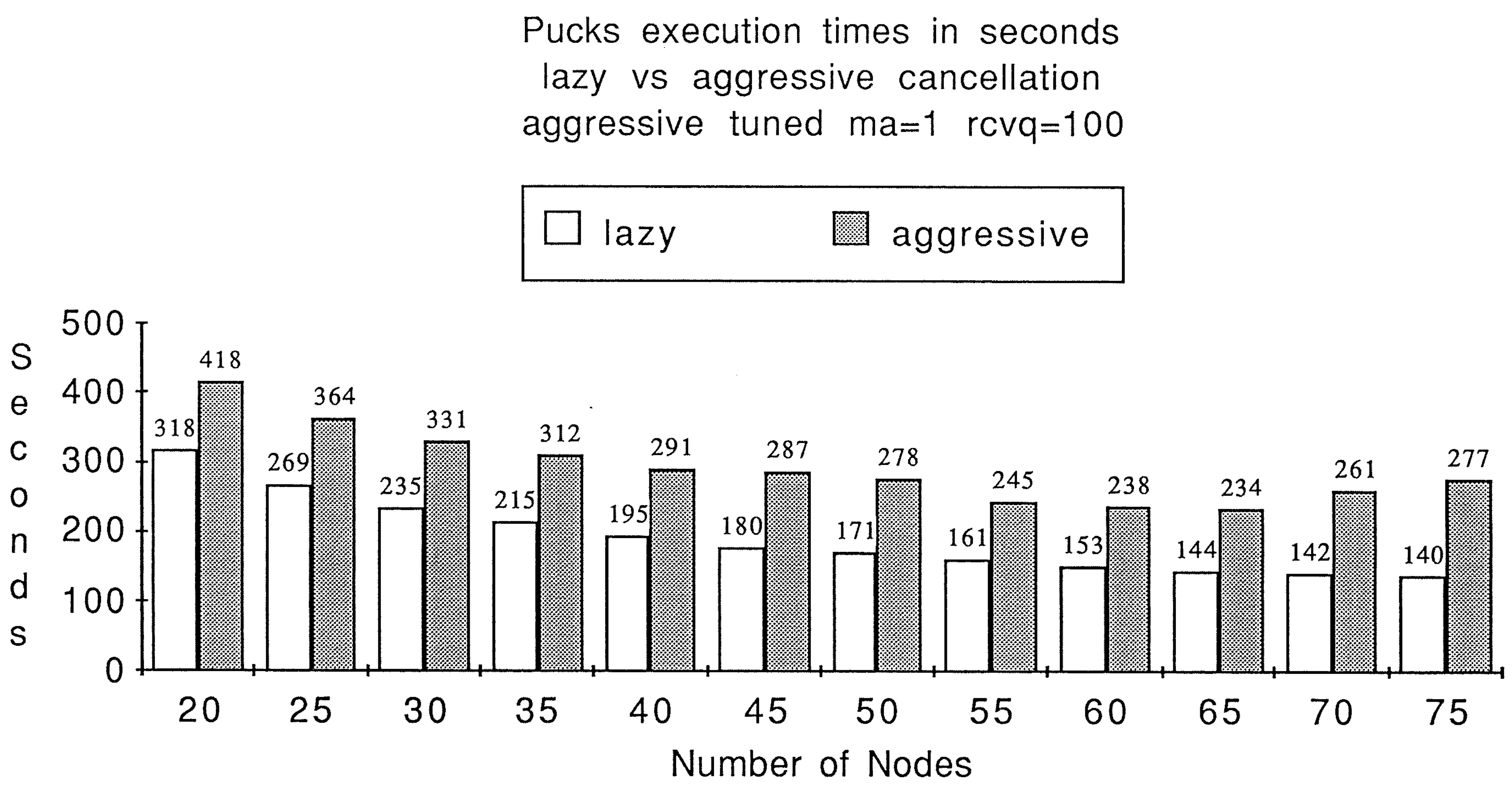

Pucks execution times in seconds
lazy vs aggressive cancellation
aggressive tuned ma=1 rcvq=100
lazy
aggressive
Seconds
0
100
200
300
400
500
318
418
269
364
235
331
215
312
195
291
180
287
171
278
161
245
153
238
144
234
142
261
140
277
20
25
30
35
40
45
50
55
60
65
70
75
Number of Nodes

Figure 12

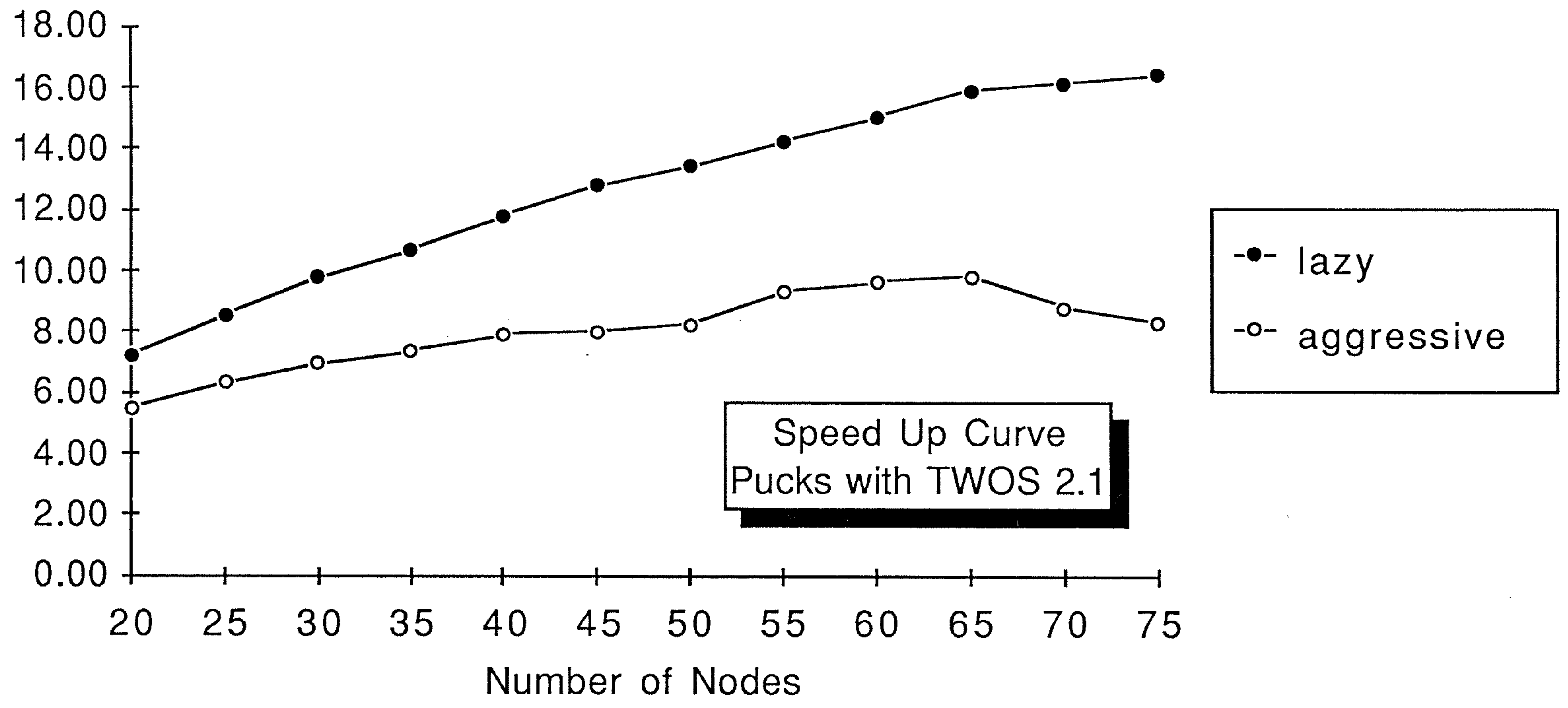

18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
20
25
30
35
40
45
50
55
60
65
70
75
Number of Nodes
Speed Up Curve
Pucks with TWOS 2.1
-●- lazy
-○- aggressive

Figure 13

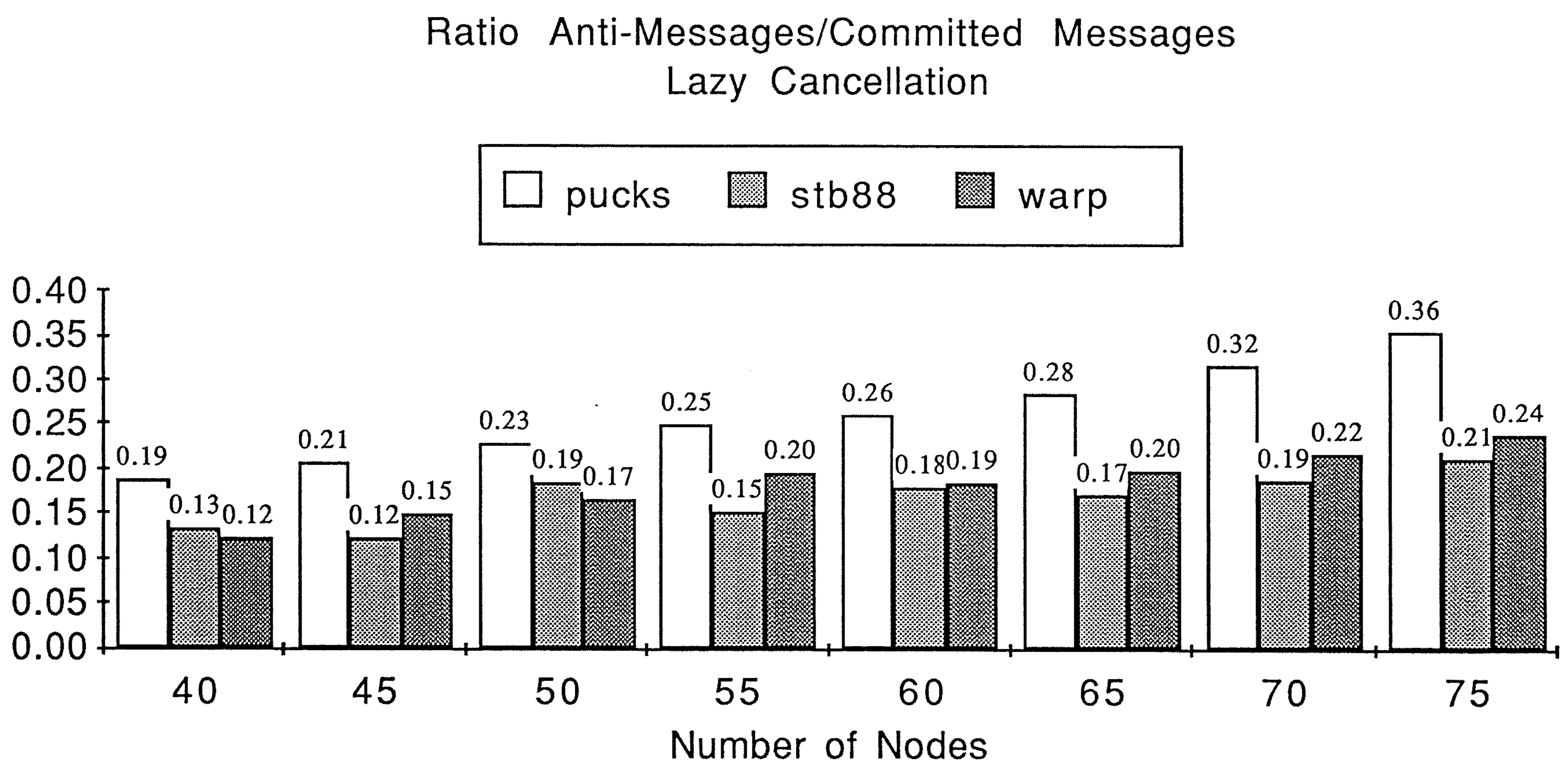

Ratio Anti-Messages/Committed Messages
Lazy Cancellation
pucks
stb88
warp
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.19
0.13
0.12
0.21
0.12
0.15
0.23
0.19
0.17
0.25
0.15
0.20
0.26
0.18
0.19
0.28
0.17
0.20
0.32
0.19
0.22
0.36
0.21
0.24
40
45
50
55
60
65
70
75
Number of Nodes

Figure 14

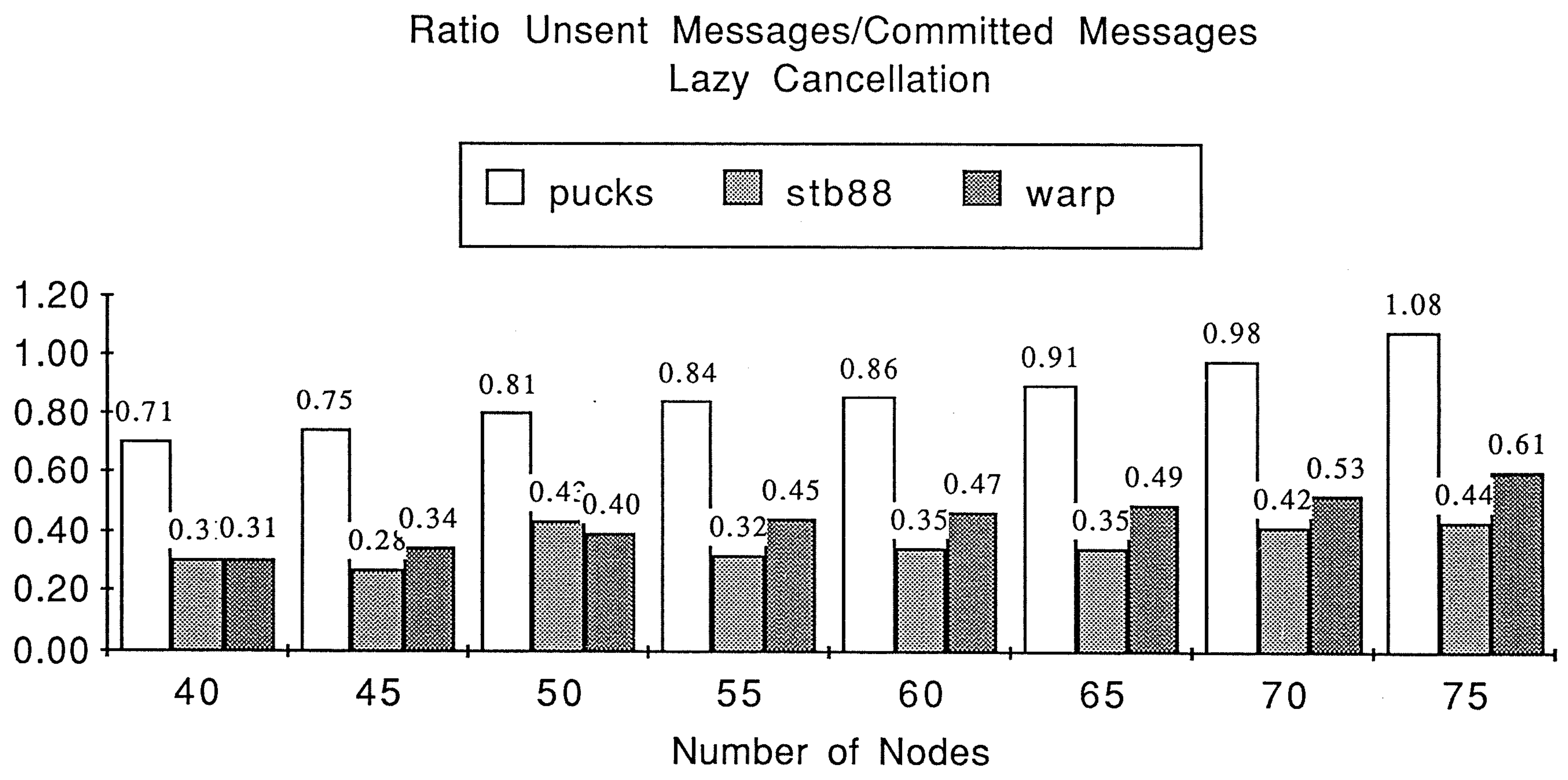

Ratio Unsent Messages/Committed Messages
Lazy Cancellation
pucks
stb88
warp
1.20
1.00
0.80
0.60
0.40
0.20
0.00
0.71
0.3
0.31
0.75
0.28
0.34
0.81
0.43
0.40
0.84
0.32
0.45
0.86
0.35
0.47
0.91
0.35
0.49
0.98
0.42
0.53
1.08
0.44
0.61
40
45
50
55
60
65
70
75
Number of Nodes

Figure 15

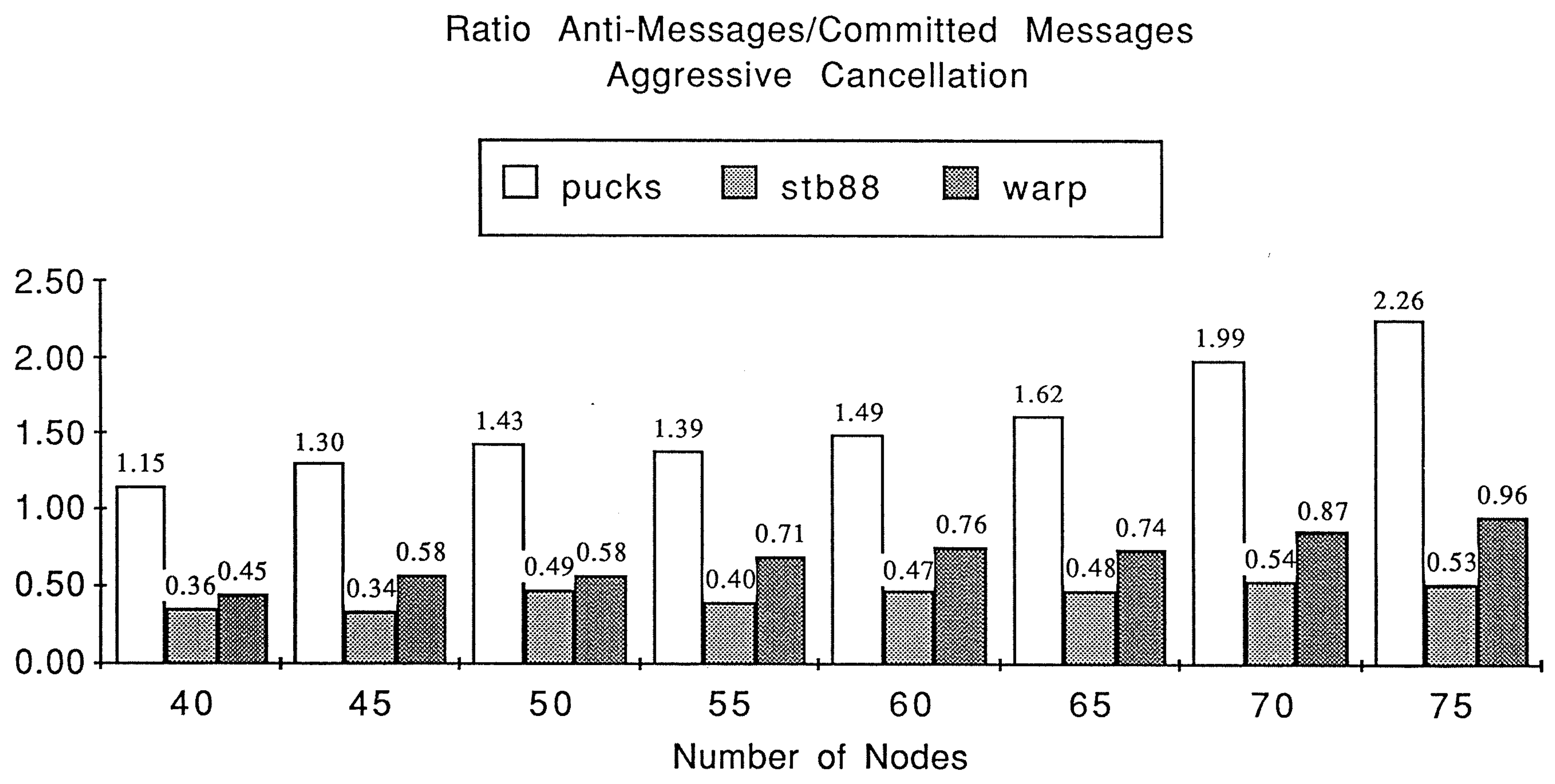

Ratio Anti-Messages/Committed Messages
Aggressive Cancellation
pucks
stb88
warp
2.50
2.00
1.50
1.00
0.50
0.00
1.15
0.36
0.45
1.30
0.34
0.58
1.43
0.49
0.58
1.39
0.40
0.71
1.49
0.47
0.76
1.62
0.48
0.74
1.99
0.54
0.87
2.26
0.53
0.96
40
45
50
55
60
65
70
75
Number of Nodes

Figure 16