RELIABLE SCHEDULING IN A TMR DATABASE SYSTEM

Frank M. Pittelli
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, New Jersey 08544

CS-TR-028-86

April, 1986

Reliable Scheduling in a TMR Database System

Frank M. Pittelli
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, New Jersey 08544

ABSTRACT

A Triple Modular Redundant (TMR) system achieves high reliability by replicating data and all processing at three independent nodes. When TMR is used for database processing all non-faulty computers must execute the same sequence of transactions, and this is ensured by a collection of processes known as "schedulers". In this paper we study the implementation of efficient schedulers through analysis of various enhancements such as null transactions and message batching. The schedulers have been implemented in an experimental TMR system and the evaluation results are presented here.

April 4, 1986

Reliable Scheduling in a TMR Database System

Frank M. Pittelli
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, New Jersey 08544

1. Introduction

With the advent of powerful, yet inexpensive, micro-computers it is becoming practical to design reliable database systems using redundant processing techniques. One such technique, known as Triple Modular Redundancy (TMR), has long been used as the basis for reliable control systems (e.g., aircraft control). The basic idea is that every system task is executed concurrently by three separate processors. The three outputs are given to a voter that selects the result given by two out of three of the processors. The system can tolerate the *arbitrary* failure of one of the processors, and still give correct results. We emphasize the word "arbitrary" here: the failed node can act maliciously, erratically, or not at all. The correct result will still be output by the voter. This capability to mask out unpredictable failures is the major strength of the TMR approach.

In this paper we concentrate on a TMR system for database processing. There are a number of key differences with conventional "system control" types of TMR. In particular, the inputs are not signals (e.g., voltages) from hardware sensors; they are database transactions. Similarly, the outputs are not usually control signals (which can be voted on by a simple and very reliable hardware voter), but are possibly large subsets of the database. Furthermore, the order in which the transactions are executed at the nodes is critical, so the non-faulty (i.e., perfect) nodes should execute them in exactly the same sequence. Finally, control applications tend to have a relatively small amount of state information that must be remembered from one task to the next. In database processing, however, the entire database is part of the system state and recovery of this information after a failure is a challenging problem.

Inherently, TMR is expensive: three computers are doing the work of one. However, it does provide extremely high protection against hardware through a very simple approach. Specifically, within the TMR system, transaction processing is done by each node in an almost conventional fashion. That is, each node processes transactions without requesting remote locks and without sending backup data to remote nodes. Such simplicity is paramount in reliable designs to reduce the likelihood of software bugs.

It could be argued that it is impractical to replicate an entire database on three nodes. However, when we speak of "databases" we do not necessarily refer to large ones. In many systems a small, but critical, portion of a larger database must be maintained with high reliability. In such situations, the critical portion may be maintained by a TMR subsystem. (IBM's Highly Available System [Aghi83] protects certain system tables using a similar approach.) Hence, we can view the TMR system as the manager of such a critical database subsystem.

There are several options for TMR database processing. At one extreme we could build a computer with TMR built into each internal circuit. That is, there would be three copies of each chip (IC), and the outputs of these chips would go through a hardware voter. The main advantage of this approach is that the resulting computer would appear to the user as a conventional system. Thus, a conventional database system could be implemented directly, yet achieving high reliability.

In the middle ground, TMR processing can be used to provide highly-reliable stable storage. In particular, every read operation executed by a given processor could be done redundantly on three database copies, selecting the majority value. Furthermore, a database copy would be updated only if at least two out of three processors have requested the same update. Essentially, such a system consists of a TMR processor connected to a TMR storage device. This system's main advantage is that it can be programmed in a conventional style, while using off-the-shelf components. A complete discussion of this type of system can be found in [Schl83].)

At the other extreme, we can have three conventional computers and distribute the transactions to be executed through software. In this case, an agreement protocol (i.e., scheduling algorithm) is required to ensure that the same set of transactions is executed in the same order. Once a processor knows that it must execute a transaction, it executes it locally, using a relatively conventional database system. The output of the entire transaction is sent to the user, where the voting takes place. The major advantage of this strategy is the minimal interaction between the processing elements. In particular, the processors exchange transaction information before and after its execution, but not during its execution. Such a loosely-coupled asynchronous system may allow greater efficiency than systems built with chip-level voting, or those which provide reliable I/O operations. (The tradeoffs related to the choice of TMR database strategies are further discussed in [Garc86].)

The objective of this paper is to study transaction scheduling in the last TMR database approach. With this strategy, the scheduler is one of the most critical components in the system, in that it acts as a throttle for the entire system. If transactions can't be distributed fast enough and with moderate overhead, then the entire TMR system will have poor performance.

We start by presenting the scheduling problem in detail (Section 2) and the basic solutions proposed in the literature (Sections 3 and 4). Then we study three enhancements: premature scheduling, null transactions, and message batching (Sections 5, 6, 7). In Section 8 we discuss an experimental evaluation of all of the techniques. Our experiments were run on a network of SUN-2/120 running Berkeley UNIX version 4.2. The results then lead us to an improved scheduling algorithm that dynamically changes its policy as the load on the system varies.

Our motivation for the detailed study of scheduling algorithms is a full TMR database system being constructed. Since our goal is to find a very good scheduler for a *real* system, we emphasize in this paper "practical" issues and solutions. Readers interested in the "theory" behind the implementations can refer to the papers cited shortly. In particular, [Garc86] presents more formally our failure models and correctness criteria. A description of the full TMR system and some preliminary performance results (*not* using the optimal scheduler we present here) can be found in [Pitt86].

Finally, note that TMR techniques, including the ones presented in this paper, can be easily generalized to NMR techniques. However, we will restrict our discussion to triple redundancy, both because it is most likely in practice and because our experimental results are for such a level of replication.

2. The Scheduling Problem

One of the most important requirements of the TMR database system is that all perfect nodes (i.e., nodes that have not failed) must process the same transactions in the same order. The system components which enforce this operation are known as the transaction schedulers and their design and performance are the focus of this paper. (A complete discussion of the system operation is provided in [Pitt86].)

The normal processing of transactions in the TMR system is shown in Figure 1. A database user submits a transaction request to one of the node schedulers, which, in concert with the other schedulers, is responsible for distributing it to all perfect nodes. In particular, the schedulers must guarantee the following properties in the presence of node failures.

S1) If a user submits a transaction to a perfect node, then that transaction is "eventually" scheduled by all perfect nodes. Furthermore, if a user submits a transaction to a faulty node, then either all or none of the perfect nodes schedule that transaction.

S2) All perfect nodes schedule transactions in the same order.

Note, we do not require that "all" transactions be scheduled. Specifically, a transaction submitted to a faulty node may or may not be scheduled by the perfect nodes. Also, we do not require transactions to be scheduled "simultaneously" by all perfect nodes, only eventually. This allows some degree of asynchrony within the TMR system.

The first property is a version of the Byzantine Generals Problem [Lamp82] and has many proposed solutions [Dole82, Lync82, Peas80]. The second property essentially requires that all "actions" on a replicated "state" be "synchronized". Two approaches have been proposed to guarantee such a property. Lamport has described a "state machine approach" that makes use of reliable broadcasts and synchronized real clocks to cope with arbitrary node failures [Lamp84]. To the same end, Schneider makes use of reliable broadcasts and logical clocks, implemented via acknowledgement messages [Schn82]. Both approaches are very general, and can be applied to many different distributed applications. Consequently, the design of the TMR system will incorporate ideas from both.

As will be discussed later, the following assumptions are required for the design of the schedulers.

A1) Any message transmitted between any two perfect schedulers is received and processed in less than M_{delay} seconds. That is, there is a maximum bound on the time to transmit a message. (This implicitly assumes that messages are not lost between perfect nodes.) We do not, however, assume that all messages are processed in the order sent.

A2) All perfect schedulers have synchronized clocks that differ by at most C_{diff} units. (Many different algorithms have been proposed to provide synchronized clocks [Lund84, Lamp85, Halp84].)

A3) All messages are authenticated. That is, all messages are signed by the sender in such a way that the receiver can determine unequivocally who sent it and what it contained. (In practice, the signature is an encryption of the entire message itself using a public-key encryption scheme [Diff76].)

3. Reliable Broadcast Algorithm

The basis for all of the scheduling algorithms in this paper is a reliable broadcast algorithm. Unlike those proposed in other papers, this reliable broadcast algorithm is designed without the use of "message" rounds. That is, any scheduler can initiate a reliable broadcast at any time and many different reliable broadcasts can proceed simultaneously. A pseudo-code description of the algorithm is presented in Appendix 1, while Figure 2 shows the exchange of messages for a single broadcast. At this time, we briefly describe the operation of the algorithm, followed by a discussion of its properties.

The algorithm makes use of two transaction sets at each node, the pending set and the final set. The goal of the reliable broadcast is to maintain the same "final set" of transactions on all perfect nodes (property S1). That is, after any number of simultaneous broadcasts have completed, (during which the pending sets will usually be different) all perfect nodes must have the same final set of transactions. (It remains for the scheduling algorithm to extract transactions from this set "in the same order" on every perfect node.)

As shown in Appendix 1, when a node receives a transaction from the user, it computes an expiration time for it equal to the current time plus a constant, S_{delay}. The plan is that *by* the expiration time, the transaction will be in the final set on all perfect nodes. The expiration time and a unique signature form a timestamp for the transaction. The transaction and its timestamp are broadcast to the other nodes in a "request" message. Furthermore, the sender would immediately add the transaction to its own final set, completing its role in the algorithm.

After verifying the timestamp of a request message (see Appendix 1), the other nodes store the entire message in their pending sets. (Essentially, the transaction will remain in each pending set until it is determined to be correct, as we will see shortly.) Additionally, the recipient appends its own signature to the message and sends this "confirmation" to the remaining node (see Figure 2). Similar processing is performed when a confirmation is received, except that no further messages are transmitted.

The final processing of requests and confirmations is triggered by internal events on each node. In particular, if corresponding request and confirmation messages exist in a node's pending set (i.e., their expiration times and contents are identical), then the transaction is added to that node's final set. Conversely, if the messages are different, then they are both ignored. Finally, if any message in the pending set has an expired timestamp (i.e. it's expiration time is less than the local clock time), then it is immediately added to the final set.

The correctness of the broadcast algorithm relies heavily on Assumptions A1-A3 and on the appropriate choice of S_{delay}. In particular, S_{delay} must be at least $2M_{delay} + 2C_{diff}$. That is, the expiration time for a given transaction must be far enough into the future so as to guarantee that there is enough time for all request and confirmation messages to be exchanged. Given this, and unforgeable signatures, it can be shown that the algorithm given in Appendix 1 satisfies Property S1, despite an arbitrary failure of a single scheduler (see Appendix 2).

Finally, the performance of the reliable broadcast algorithm must be restricted or "throttled" to prevent a crippling situation that can be caused by a faulty node. In particular, consider a faulty node that initiates a continuous stream of reliable broadcasts. Since the algorithm requires the other nodes to process two messages for every broadcast initiated, the faulty node can easily overwhelm them and they won't have time to initiate their own broadcasts. Essentially, in this situation, a single faulty node has crippled the system by causing it to perform too much work. (Note, the broadcasts initiated by the faulty node may have nothing to do with the system tasks being executed.) Since we can never prevent such faulty behavior, the best we can do is to restrict *all* nodes to some pre-defined limit. That is, nodes will not be allowed to initiate more than a certain number of reliable broadcasts in a fixed amount of time. In this way, perfect nodes will always be able to initiate their own broadcasts.

In the experimental system, the restriction is implemented by forcing each node to assign expiration times separated by at least 50 ms. If any node receives two request messages from the same origin that have expiration times too close together, then the second is ignored. This limits the overall system to a throughput of 60 reliable broadcasts per second, which, as will be shown in Section 8, is greater than the average performance of the schedulers in normal operation. It does, however, restrict a faulty node that could otherwise initiate broadcasts every 20 ms.

4. Basic Scheduling Algorithm

Given that the reliable broadcast algorithm guarantees property S1 it remains to design an algorithm that provides property S2. That is, we must insure that all perfect nodes extract transactions for execution from their final transaction sets in the same order. We call this extraction procedure "transaction scheduling".

A simple method for transaction scheduling, given reliable broadcasts, has been proposed by Lamport [Lamp84]. Appendix 3 presents a pseudo-code description of such an algorithm implemented in an asynchronous fashion. That is, each node performs actions triggered by the receipt of messages and internal events. In particular, when a node receives a transaction request from the user, it initiates a reliable broadcast of the transaction. Furthermore, each node extracts or "schedules for execution" those transactions in the final set that have an expired timestamp. (Remember, the reliable broadcast algorithm guarantees that a transaction is added to the final set "no later" than its expiration time; it may be added earlier.) It is important to note that the pseudo-code algorithm states that a transaction is scheduled "immediately" when its timestamp expires. In a real implementation, processing delays may cause the final set to contain more than one such transaction. Accordingly, the real algorithm must extract the transactions using a total ordering based on their expiration times and the ID of the node which initiated them to guarantee property S2.

Although the basic scheduling algorithm is straight-forward, it possesses a number of useful properties. First, transactions are scheduled strictly by their expiration times, which are based on real clocks. Consequently, users can be allowed to "select" an appropriate expiration time. In this way, the user can initiate transactions to be scheduled at specific times (i.e. during overnight slack periods.) A similar benefit is gained by the system during reliable broadcast processing. Specifically, if a scheduler knows that a given transaction won't be scheduled in the immediate future, it may decide to postpone request and/or confirmation messages, thereby allowing higher priority transactions to proceed. Finally, since transaction distribution and scheduling can be effectively separated, the system can be designed to distribute all transactions during one phase, followed by all scheduling and execution. For example, consider a database that is completely replicated on three ships. The tactical situation may restrict ship-to-ship communication to specific time periods. During these periods, the distribution of transactions can be performed. In between these periods, all ships can execute the accumulated transaction set knowing that the same results will be achieved.

5. Premature Scheduling

The basic scheduling algorithm made use of a fixed scheduling delay. That is, each transaction was assigned an expiration time which was at least $S_{delay} = 2M_{delay} + 2C_{diff}$ seconds in the future. Unfortunately, since M_{delay} is the "maximum" message delay, most request and confirmation messages are processed well before S_{delay} seconds have elapsed. (In our implementation the maximum message delay is 500 ms, while the average time to transmit a message is approximately 11 ms. This large difference is usually caused by the operating system, not the network hardware.) Also, depending on the implementation used, C_{diff} may or may not accurately represent the average difference between node clocks. (Recall, C_{diff} is the maximum difference allowed.) Consequently, it may be advantageous to "prematurely" schedule transactions to reduce the average scheduling time. That is, if all nodes are perfect and all messages are transmitted correctly, then a given transaction may be scheduled before its expiration time.

The basic scheduling algorithm can be modified to allow premature scheduling through the use of sequence numbers (see Appendix 4). First, each scheduler adds a sequence number to the timestamp of every request message that it initiates. In normal processing (no-failure periods), the sequence number is monotonically increasing for each scheduler. Additionally, each scheduler must keep track of the sequence numbers used by every scheduler. A given transaction, T_x, with sequence number, N, may be prematurely scheduled if all of the following conditions hold.

1) T_x bears the earliest expiration time of all transactions in the transaction set.

2) N is the "next" sequence number expected from the scheduler that submitted it. That is, all transactions, numbers 0 through $N-1$, from T_x's originator have been scheduled.

3) For *every* scheduler, the transaction set contains the "next" transaction from that scheduler.

The following example illustrates the reasons for these conditions.

Figure 3 shows the transaction set and sequence number table maintained by scheduler S_2. Since T_1 is the earliest transaction, it satisfies condition 1. Also, S_2 can determine that T_1 is the next transaction from S_1, using the sequence number table. That is, there are no earlier transactions from S_1 which have not been scheduled by S_2 (condition 2). (If such a transaction does arrive it is simply ignored because S_1 is incorrectly generating duplicate sequence numbers.) Similarly, S_2 knows that it has no outstanding transactions. However, at this point, S_2 can't prematurely schedule T_1, because the "next" transaction from S_3 hasn't been scheduled and it may have an expiration time earlier than time 1300. If it does arrive and has an expiration time greater than time 1300, condition 3 is satisfied and T_1 can be scheduled. (In Schneider's terminology, T_1 is "fully-acknowledged" by all schedulers [Schn82].) Similar, but not identical, processing will occur at the other nodes if no failures occur. Keep in mind, if S_2's clock reaches time 1300 before the next transaction from S_3 is received, then T_1 would be scheduled automatically.

A faulty scheduler can always prevent premature scheduling simply by skipping sequence numbers. In such a case, no transaction can be prematurely scheduled because the "next" transaction from the faulty scheduler is never seen by any scheduler. (Unfortunately, there is no way to prevent such behavior.) However, we also state, without proof, that a faulty scheduler can not cause inconsistent schedules by using sequence numbers maliciously. This is because, in the worst case, each perfect scheduler will schedule a given transaction when its timestamp expires. For example, suppose that S_3 is faulty and relays a correct confirmation for T_1 to S_2, but sends no confirmation for T_2 to S_1. In this case, S_2 will prematurely schedule T_1, but S_1 will wait until T_1's expiration time. During this time, the two perfect schedulers (S_1 and S_2) will have different schedules, but "eventually" they will be the same, and that fulfills the scheduling requirements.

6. Null Transactions

The key idea behind premature scheduling is that a transaction can be scheduled when the next transaction from every scheduler is in the local final transaction set. However, during slack periods, a given scheduler may not have any transactions to submit. In such cases, a steady supply of transactions from that scheduler can be accomplished through the use of "null" transactions [Lamp84, Schn82].

Null transactions are processed like normal transactions with the exception that they are never scheduled. That is, null transactions remain in the transaction set until they are fully-acknowledged, at which time they are discarded. We illustrate the use of null transactions through an example.

Consider the situation, as seen by S_2, shown in Figure 4a. Without additional information, S_2 would have to wait until time 1300 before scheduling T_1. The use of null transactions eliminates this waiting period. In particular, S_2 and S_3 initiate null transactions that have expiration times greater than T_1's. Once these transactions are added to S_2's transaction set (Figure 4b) S_2 can prematurely schedule T_1. (Note, S_2 generated a null transaction to help the other schedulers.)

Null transactions are effective only if a given scheduler can determine "when" to generate them. At one extreme, a null transaction could be sent for every transaction added to the transaction set. In general, however, null transactions should be generated only when they are necessary. At this point, we present three conditions that should hold before a given scheduler, S_i, initiates a null transaction. Let T_l be the last transaction initiated by S_i, at time $s(T_l)$, with an expiration time, $e(T_l)$.

1) The final transaction set at S_i contains at least one transaction T_x, with expiration time $e(T_x)$, such that $e(T_l) \leq e(T_x)$.

2) $local\_time > s(T_l) + N_{delay}$, where N_{delay} is a fixed delay period such that $0 \leq N_{delay} < S_{delay}$.

3) Time $e(T_x)$ is not *too far* into the future. In particular, $e(T_x) \leq local\_time + S_{delay}$.

Now we discuss the reasons for these conditions. Keep in mind, S_i submits null transactions to help the "other" nodes schedule transactions prematurely.

The first condition guarantees there is at least one "worthy" transaction, T_x, that can benefit from a null transaction. That is, T_x has an expiration time greater than the last transaction submitted by S_i. In Figure 4a, T_1 is such a worthy transaction, from the point of view of node 2, assuming the last transaction from S_2 has an expiration timestamp $e(T_l) < 1300$. Note, any transaction that has an expiration time less than $e(T_l)$ doesn't require a null transaction because T_l itself can be used to prematurely schedule it.

The second condition avoids the generation of excessive nulls. If N_{delay} is set to 0, a null transaction will be generated as soon as a worthy transaction T_x exists. This may be the best thing to do from T_x's point of view, but too many nulls may be counter-productive. Therefore, a node doesn't send a null transaction if it has recently (within N_{delay} seconds) sent another transaction (real or null). The hope is that another real transaction will arrive at S_i soon, making the null unnecessary. Selecting a good value for N_{delay} is a challenging problem: setting it too low may cause excessive message traffic, while setting it too high makes us revert to the premature scheduling algorithm without nulls. The impact of N_{delay} is discussed further in Section 8.3.

The third condition exists to prevent the premature scheduling of a transaction that has an expiration time which is relatively far into the future. For example, consider the following situation, assuming that S_{delay} is equal to 10. Suppose node 1 submits a transaction, T_1, at time 1200 with an expiration time of 1300. That is, T_1's expiration time is much greater than its start time plus S_{delay}. If condition 3 didn't exist, node 2 may, at time 1205, submit a null transaction, N_2, with an expiration time of 1301. Furthermore, suppose a real transaction, T_2, was submitted by node 2 at time 1210. For correctness T_2 would have to be assigned an expiration time greater than 1301, say 1302.

If all nodes were functioning properly such a situation wouldn't be a problem. In particular, all nodes would submit transactions (null or real) which would allow T_2 to be prematurely scheduled, well before time 1302. (Essentially, all nodes would simply advance their clocks up to 1302.) On the other hand, suppose a node were faulty and refused to submit any further transactions, thereby preventing the premature scheduling of T_2. That is, T_2 would not be scheduled until its expiration time, at time 1302. Essentially, a faulty node, through malicious behavior, has caused a large increase in T_2's response time.

Condition three prevents such a situation by specifying a "minimum scheduling time" for every transaction. In particular, a null transaction will be generated for T_x when the local time is greater than $e(T_x) - S_{delay}$.* Any time before then, scheduler S_i can submit a transaction with an expiration time less than $e(T_x)$, which will be scheduled "before" T_x. In this way, a faulty node can only increase the response time of other transactions by at most S_{delay}, and

* Note, if the schedulers always choose $e(T_x) = s(T_x) + S_{delay}$, then the minimum scheduling time is equal to $s(T_x)$.

this is no worse than the basic algorithm.

Notice that even with these conditions it is possible that a null transaction from S_i is not helpful. In particular, if the null transaction is added to the final transaction set before time $e(T_x)$, then it may be possible to schedule T_x early (given the next transaction from the other nodes exist.) However, if the reliable broadcast of the null transaction takes too long, it may arrive after $e(T_x)$, in which case the null transaction will be wasted. (Recall that, at the latest, T_x is scheduled when its timestamp expires at time $e(T_x)$.) Hence, nulls should not be sent when they are "expected" to arrive after $e(T_x)$. In Section 8.3 we discuss how this additional rule, which depends on the run-time performance of the reliable broadcast algorithm, can be implemented.

7. Batching Transactions

The scheduling algorithms presented so far produce many messages for each transaction. In particular, in a three node system four messages are generated for each transaction (real or null) submitted by a scheduler (Figure 2.) A decrease in message processing may be achieved by scheduling transactions in "batches". That is, each scheduler collects a number of transactions before submitting them to the other schedulers, where the entire batch is scheduled at one time. Of course, this strategy has both advantages and disadvantages.

On one hand, transaction batching decreases the total number of messages generated. For example, if three transactions are batched together, then eight messages are saved during the scheduling process. Of course, this decrease in the number of messages also decreases the amount of message "processing" performed by the schedulers. (Message processing includes the transmission time and the CPU time used by the sender and receiver.) Figure 5 shows the average processing time for messages of varying sizes in our system. We can see that the system-imposed overhead for each message (i.e., the y-intercept) is about 8.6 ms. Therefore, by batching N transactions together we may be able to save $4 \times 8.6 \times (N-1)$ ms of processing time, thereby increasing system throughput.

On the other hand, the schedulers must know "when" to submit the current batch of transactions. For example, with two transactions waiting in the batch it may be 2 ms or 2 days before a third arrives. Consequently, the scheduler should only hold a batch until it has been waiting a certain period of time or until it has been filled (i.e., the largest possible message size has been reached). Furthermore, batching may increase the average scheduling time because transactions are sometimes "waiting" for a batch to be released.

8. Experimental System

In order to analyze the various tradeoffs associated with the scheduling algorithms, an experimental system has been implemented which isolates the schedulers from the rest of the TMR database system (Figure 6). In particular, multiple users are replaced by a single monitor that submits concurrent transactions to and receives transactions from the schedulers. Such isolation allows greater control over testing itself, yet examines all processing associated with transaction scheduling. In some cases, as will be discussed later, an artificial processing load is imposed on the system to better understand the impact of other processes (e.g. transaction manager) on the schedulers.

Individual processes in the experimental system are executed on separate SUN-2/120 workstations, running Unix 4.2. Communication between processes is implemented by a 10 Mbit per second Ethernet, using the standard interprocess communication (datagram) facilities in Unix.* In general, all tests were run in the following manner.

At any given time, the monitor allows a fixed number of transactions, M, within the system, and we call M the "level of multiprogramming". A given transaction is submitted to one of the schedulers, who is responsible for distributing it to the other schedulers. When that transaction is scheduled, by each particular scheduler, it is returned to the monitor. After receiving two of the three responses, the monitor records the elapse time since the transaction's submission as its "response time". When all three responses have been received, the monitor resubmits the transaction to the system. Throughout the test, the monitor submits transactions to the schedulers in such a way as to maintain a uniform load. For example, if there are fifteen transactions in the system, each scheduler will be responsible for initiating five of them.

In addition to the average transaction response time, the throughput of the system, in terms of transactions per second, is determined for every test by dividing the total number of transactions executed by the elapse time of the entire test. The throughputs of many similar tests are combined to yield an average system throughput, with a confidence interval of five percent. For clarity, the data points in all of the graphs presented in this paper represent the central values of the confidence intervals.

Since the experimental performance of the schedulers is independent of transaction processing, we represent a transaction simply as a message of a

* We realize that the ethernet is a single point of failure in our experimental system, but we are using it for a practical reason: we have no other. However, our design does not require the "broadcast" capability of the ethernet. Any direct connection between the processing nodes will suffice. Furthermore, we believe that a different network will not change our performance results significantly.

given size (64 bytes). Experiments conducted with the complete TMR database system show that an increase in transaction size affects the rest of the system much more than it affects the schedulers. Therefore, we will not address that issue in this paper.

Finally, throughout the following discussions we make reference to a variety of empirical observations. In fact, these observations are based on other experiments aimed at gaining a better understanding of the system's overall performance. Unfortunately, we can't present all of these results formally. Similarly, many of the experimental parameters have been varied beyond the values presented in the graphs. For simplicity, we have selected the results and ranges that demonstrate our observations most clearly.

8.1. Comparison of Algorithms

Figure 7 shows the average response time and throughput for the basic, premature and null scheduling algorithms. (Remember, the "null" scheduling algorithm also makes use of premature scheduling.) The results shown are for a fixed-size transaction message (64 bytes), with varying levels of multiprogramming. Also, in our system, S_{delay} is equal to 1000 ms (see Section 4), while N_{delay} has been chosen to be 200 ms.

For the most part, we see that the response of the basic algorithm is independent of the multiprogramming and that its throughput is directly related to it. In general, the basic algorithm has predictable parameters; response time slightly greater than S_{delay} and throughput slightly less than M.

Figure 7 also shows that the premature and null scheduling algorithms perform approximately the same, except during low levels of multiprogramming. In particular, both algorithms become saturated when $M > 12$, with the null algorithm having a slightly greater throughput. This is probably due to the "gaps" in transaction processing caused by queuing delays throughout the communication network. During these gaps, null transactions provide faster response time, thereby increasing throughput in the closed system. Bear in mind, however, the increase is less than the statistical confidence interval (5 percent). Finally, when $M < 3$, the premature algorithm mimics the basic algorithm because there is never a complete set of "next" transactions for each node. On the other hand, the null algorithm proceeds at a rate determined strictly by the choice of N_{delay}.

8.2. Network Load

Some important differences between the scheduling algorithms are the number of messages generated and their dependence on the system load. To investigate these differences, we make use of an artificial network load consisting of a set of processes that broadcast a message over the network every 25 ms. (Such a network load was chosen to simulate the amount of additional processing which is typically present in the complete TMR database system.) Figure 8 shows the system performance of all three algorithms while the network is loaded. Comparing it with Figure 7, we make a number of observations.

For the most part, the basic algorithm is unaffected by the load. In fact, the average response time rose only 17 percent, while the throughput dropped 3 percent. This is mainly because the basic algorithm spends most of it's time "waiting" to schedule transactions. A delay in transaction distribution is easily handled within the S_{delay} time period. Once again, the basic algorithm's simple design yields predictable performance.

On the other hand, the performance of the premature and null algorithms has been reduced significantly and both become saturated when $M > 9$, instead of when $M > 12$. In particular, the premature and null algorithms perform, respectively, 33 and 40 percent worse than the no-load situation (Figure 7). Figure 8 also shows that the null algorithm performs 10 percent worse than the premature algorithm when the network is loaded. To understand this situation, we examine the percentage of nulls generated by the schedulers with and without a network load, as shown in Figure 9. We see that the network load causes a greater number of nulls. Specifically, the network load, and its corresponding load on the node processors themselves, causes each scheduler to execute at a slower rate. Therefore, there is a greater chance for a scheduler to go N_{delay} ms without submitting a real transaction, thereby causing the generation of a null transaction. Also, the higher load makes it more likely for a null to be delayed, arriving after the expiration of the transaction it was intended to help. The extra work caused by these nulls results in a decreased throughput of real transactions. In this case, null transactions hinder the system because N_{delay} is too small.

8.3. Impact of Null Delay

As seen in the last test the performance of the null algorithm depends on the value of N_{delay}, relative to the load on the system. Figure 10 shows this relationship by plotting curves for N_{delay} equal to 100, 200, 300, and 400 ms.* The response and throughput graphs support the same conclusion; at low levels of multiprogramming ($M < 3$), a small value for N_{delay} should be used. This generates a large percentage of null transactions (Figure 10c), but this extra

* Note, values of N_{delay} greater than 400 ms yield results that are statistically equivalent to the premature algorithm, therefore they have not been shown.

processing is easily absorbed by the lightly-loaded schedulers. Conversely, at high levels of multiprogramming ($M \geq 6$) a large value for N_{delay} is most effective, thereby reducing the percentage of nulls generated. In fact, when N_{delay} equals 400, the null algorithm performs as well as the premature algorithm, yet provides better performance at low multiprogramming levels.

The results of this test suggest that a "hybrid" algorithm would be useful. That is, a small value of N_{delay} could be used while the system is lightly-loaded, and the generation of null messages can be stopped (by using a large N_{delay}) when the system is heavily-loaded. Specifically, the choice of an appropriate N_{delay} can be based on the average time between transactions submitted by the user, $T_{arrival}$, and the average time to schedule a transaction, S_{resp}. Therefore, at this time, we examine two different hybrid algorithms. One uses a fixed "cutoff" based on analysis of the overall system, while the other attempts to determine the best cutoff using run-time parameters. In both cases, S_{resp} and $T_{arrival}$ are calculated by each scheduler based on the last ten transactions processed. (Once again, the number ten was chosen based on preliminary tests which showed that anything smaller caused erratic operation and anything larger prevented the schedulers from responding quickly to changes in the work load.)

Analysis of Figure 10b clearly shows that the schedulers should stop using nulls when the multiprogramming level is greater than 4. This cutoff point corresponds to a throughput of 15 or, equivalently, an average time period of 200 ms between transactions received by a given scheduler. (Recall that there are three nodes, so a system throughput of 15 corresponds to a throughput of 5 transactions per second at each site.) Consequently, when the inter-arrival time drops below 200 ms, a scheduler knows that the system is sufficiently loaded and can stop using null transactions. The performance of the fixed-cutoff hybrid is shown in Figure 11, compared to the premature and null algorithms, with a network load. As hoped, the hybrid mimics the null algorithm at low levels of multiprogramming and the premature algorithm at high levels. Furthermore, we see a drop in the average response time between levels 3 and 5. Presumably, this is because the hybrid only uses nulls when they are most effective (i.e. during temporary slack periods caused by queuing delays). This helps to reduce the average response time, while not degrading throughput noticeably.

Given that a fixed-cutoff of 200 ms works well, we might ask if there is anything magic about that number? The answer is, no. By examining the S_{resp} versus $T_{arrival}$ (Figure 12) we see that the curves cross at 200 ms. That is, when the average time between transactions is greater than the time it takes to schedule a transaction it is beneficial to use null transactions, because that null will be processed before the next user transaction arrives. Conversely, when the time between transactions is less than S_{resp} nulls are a hindrance; the system will still be processing them when a real transaction arrives and requires processing. Our second hybrid determines the crossover point dynamically and uses null transactions accordingly.

A comparison of the fixed-cutoff and crossover hybrids is shown in Figure 13. In general, both hybrids perform the same over most multiprogramming levels. However, in the range 3 to 6 we see different behaviors. Both curves have a point at which the response time remains relatively constant, but the crossover hybrid is the higher of the two. During this transitional period, both algorithms switch between nulls and no-nulls trying to adjust to the work load. Although the values are within 7 percent, they appear to indicate that the fixed-cutoff hybrid is better able to cope with the transitional period. This is probably because the crossover hybrid makes use of two approximate measures (S_{resp} and $T_{arrival}$) instead of just one, as in the fixed-cutoff hybrid.

8.4. Message Batching

As stated earlier, the reliable broadcast algorithm generates 4 messages for every transaction (real or null) submitted. By batching 2 or more transactions together the system should achieve a greater throughput, because large messages require proportionately less work than small ones. Figure 14 shows the effects of transaction batching on the crossover hybrid algorithm ("b1" represents no batching, while "b2" and "b3" represent batches of size 2 and 3, respectively.) In this case, a network load was imposed and the "batch delay", B_{delay}, was set to $S_{delay} / 2$. That is, a batch was held until it was filled, or until the first transaction placed in the batch had waited B_{delay} ms.

As expected, when $M < 9$ batching degrades the response time and the throughput of the system. In particular, batches usually wait a full B_{delay} ms before being distributed. This delay directly increases the average response time and, consequently, decreases the throughput of the closed system. On the other hand, at high levels of multiprogramming batching provides significant advantages. First, the amount of processing saved by batched transactions increases the point at which the system becomes saturated. However, this also means that the system must have more work to execute in order to perform more efficiently. For example, a batch size of 1 is saturated when M equals 12, but it still has a greater throughput then a batch size of 3, which has no apparent saturation point.

In general, the greater the batch size the greater the need for a high multiprogramming level, but the greater the attainable throughput. In fact, the throughput for batch sizes 2 and 3 are within 11 and 6 percent, respectively, of the single batch case "without" a network load. That is, batching has effectively suppressed the degradation caused by the network load. We also see from the plots that a batch size of 2 is, in general, better than a batch size of 3. Specifically, a batch of 2 out-performs a batch of 3 over a greater range of multiprogramming levels ($M < 18$) and performs only slightly worse at the extremes ($M \geq 18$). In other words, the maximum benefit of transaction batching is achieved with small batch sizes.

Finally, Figure 14a shows an interesting effect of batching. We see that when $M \geq 12$, larger batches yield reduced response times, even though transactions are being delayed while batches are being filled. Essentially, the delay due to batching is overwhelmed by the total amount of processing which is saved. Since the schedulers are doing less work, they can provide faster response to each batch.

As with N_{delay}, a hybrid algorithm can be designed to change the batch size dynamically. Empirically, we found the following rule to be most effective for switching batch sizes. If S_{resp} is greater than $(batch\_size + 1) \times T_{arrival}$ then the batch size should be incremented, and if it is less than $(batch\_size - 0.5) \times T_{arrival}$ then the batch size should be decremented. Essentially, if the transaction arrival rate exceeds the rate at which transactions can be scheduled, then a larger batch size is warranted. Conversely, if transactions aren't arriving "fast enough" to fill up batches quickly, then a smaller batch size is needed.

Figure 15 shows a comparison of the batch hybrid described above and the crossover hybrid with fixed batch sizes (labels b1, b2 and b3). As hoped, the batch hybrid achieves a lower bound for response time and an upper bound for throughput, over the entire range of multiprogramming. This final scheduling algorithm makes use of premature scheduling, null transactions and variable-size batches to achieve high performance.

9. Conclusions

We have studied transaction scheduling in a TMR database system. We discovered that premature scheduling, null transactions, and message batching can significantly improve performance in certain cases. Thus, the best overall approach appears to be an adaptive algorithm that can change its scheduling policy as the system load varies. Also, we examined two "safeguards" which had to be used to prevent faulty nodes from adversely affecting the system's performance. In particular, each scheduler had to be limited to a maximum number of reliable broadcasts per unit of time. Similarly, null transactions were generated only for those transactions that weren't too far into the future. In general, these restrictions guaranteed that the system performed at a level no worse than that of the basic scheduling algorithm, despite a single faulty node.

The scheduler is one of the most important components of a TMR system, but there are a number of other components that must be studied in the future. In particular, failure detection and recovery can also have significant impact on performance. For example, failure recovery may require that part or all of the system state be maintained in stable storage, and this may degrade system performance during normal operation. Furthermore, our analysis of the schedulers focused on performance during no-failure periods. While these should be the most common periods, it is also important to study and improve scheduling during failures and their subsequent recoveries. Hence, a substantial amount of work remains to be done.

References

[Aghi83]

Aghili, H., et al, A Prototype for a Highly Available Database System, Research Report RJ-3755, IBM Research Laboratories, January, 1983.

[Dole82]

Dolev, D., and Strong, S., Polynomial Algorithms for Multiple Processor Agreement, *Proc. 14th ACM Symposium on Theory of Computing*, pp. 401-497, 1982.

[Garc86]

Garcia-Molina, H., Pittelli, F., and Davidson, S., Applications of Byzantine Agreement in Database Systems, to appear in *ACM Trans. on Database Systems*, 1986.

[Kent85]

Kent, J., Performance and Implementation Issues in Database Crash Recovery, Ph.D. Thesis, Princeton University, June, 1985.

[Lamp82]

Lamport, L., Shostak, R., and Pease, M., The Byzantine Generals Problem, *ACM Trans. on Prog. Lang. and Systems*, Vol. 4, No. 3, pp. 382-401, July, 1982.

[Lamp84]

Lamport, L., Using Time Instead of Timeout for Fault-Tolerant Distributed Systems, *ACM Transactions on Programming Languages and Systems*, Vol. 6, Num. 2, pp. 254-280, April, 1984.

[Lamps79]

Lampson, B., and Sturgis, H., Crash Recovery in a Distributed Data Storage System, Xerox Research Memo, April, 1979.

[Lipt84]

Lipton, R., Invariant Fingerprints, unpublished memo, Princeton University, 1984.

[Lync82]

Lynch, N., Fischer, M., and Fowler, R., A Simple and Efficient Byzantine Generals Algorithm, *Proc., 2nd Annual IEEE Symposium on Reliability in Dist Software and Database Systems*, 1982.

[Peas80]

Pease, M., Shostak, R., and Lamport, L., Reaching Agreement in the Presence of Faults, *Journal of the ACM*, Vol. 27, Num. 2, pp. 228-234, April, 1980.

[Pitt86]

Pittelli, F., Garcia-Molina, H., Database Processing with Triple Modular Redundancy, *Proc., Fifth Symposium on Reliability in Distributed Software and Database Systems*, January, 1986.

[Schl83]

Schlichting, R.D., and Schneider, F.B., Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems, *ACM Transactions on Computer Systems*, Vol. 1, Num. 3, pp. 222-238, August, 1983.

[Schn82]

Schneider, F., Synchronization in Distributed Programs, *ACM Trans. on Programming Languages and Systems*, Vol. 4, Num. 2, pp. 125-148, April, 1982.

[Siew82]

Siewiorek D. P., and Swarz, R. S., *The Theory and Practice of Reliable System Design*, Digital Press, 1982.

[Wens78]

Wensley, John, et. al., SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control, *Proc. of IEEE*, Vol. 66, pp. 1240-1254, October, 1978.

Appendix 1. Reliable Broadcast Algorithm

```
Broadcasting Node:
    request.sig   = my_unique_signature;
    request.time  = my_local_time + $S_{delay}$;
    request.trans = msg.trans;

    send_to_schedulers (request);
    add_to_final_set (request.trans);


on receipt of request message:
     if  not_in_pending_set (request)
     and legal_signature (request.sig)
     and my_local_time <= request.time - $S_{delay}$ / 2
     then
          store_in_pending_set (request);
          confirm = request;
          confirm.sig = confirm.sig + my_unique_signature;
          send_to_schedulers (confirm);


on receipt of confirmation message:
     if  not_in_pending_set (confirm)
     and legal_signature (confirm.sig)
     and my_local_time <= confirm.time
     then
          store_in_pending_set (confirm);


when corresponding request and confirmation messages
exist in pending set:
     if (request.trans == confirm.trans)
     then add_to_final_set (request.trans);
     else ignore both;

when a timestamp expires:
     add_to_final_set (msg.trans);
```

Appendix 2. Correctness of Reliable Broadcast Algorithm

In this appendix, we show that $S_{delay} = 2M_{delay} + 2C_{diff}$ is sufficient to guarantee that the reliable broadcast algorithm given in Appendix 1 satisfies property S1. Throughout our discussion we will consider a single transaction submitted to a given scheduler, which we will call the "general", G. The other schedulers will be called "lieutenants", L_1 and L_2.

To show that $S_{delay} = 2M_{delay} + 2C_{diff}$ is sufficient to guarantee that all perfect schedulers maintain the same transaction set we examine the three situations which can arise.

1) All schedulers are perfect.
2) A single lieutenant is insane.
3) The general is insane.

We now discuss each situation separately.

All Perfect Schedulers

Suppose that during the reliable broadcast algorithm no schedulers fail and that the general broadcasts a request message, with an expiration time T_e, to both lieutenants. Furthermore, suppose that the general sends the requests on or before time $T_e - 2M_{delay} - 2C_{diff}$, as determined by his clock. Without loss of generality, consider the processing performed by L_1. By assumption A1, it may take at most M_{delay} ms for the general's request to arrive. Accordingly, the request will be received by L_1 at time R_g, where $R_g \leq T_e - M_{delay} - 2C_{diff}$, as determined by the *general's* clock. Now we must determine T_{L1}, the time the request is received based on L_1's clock, to determine how L_1 will process the request. By assumption A2, L_1's clock may be C_{diff} ms faster or slower than the general's. Therefore, we know that $R_{L1} \leq R_g + C_{diff} = T_e - M_{delay} - C_{diff}$. Consequently, L_1 will accept the request message and will send a confirmation to L_2. (A similar argument holds for the request sent to L_2.) Now, consider the confirmation message, sent by L_1 to L_2. Once again, by assumption A1, it will be received by L_2 at time $C_{L1} \leq T_e - C_{diff}$, as determined by L_1's clock. Using assumption A2, C_{L1} corresponds to $C_{L2} \leq T_e$, based on L_2's clock. Consequently, L_2 will accept the confirmation and will compare it to the general's request. Since all schedulers are perfect, both messages will contain the same transaction, which will be added to L_2's transaction set. A similar result holds for L_1.

A Single Insane Lieutenant

The easiest failure to deal with is that of an insane lieutenant. Since messages contain unforgeable signatures (Assumption A3), an insane lieutenant is unable to introduce misleading messages. Specifically, the perfect lieutenant will receive the general's request before time $T_e - M_{delay} - C_{diff}$ (on its local clock) because of assumption A1 and A2. Additionally, the perfect lieutenant may or may not receive a confirmation from the insane lieutenant. If one is received, it must contain the general's unforgeable signature and, therefore, must be identical to the request message, because the general is perfect. In such a case the transaction will be added to the transaction set of the perfect lieutenant. Furthermore, if no confirmation from the insane lieutenant is received, then the transaction will be added to the transaction set when its timestamp expires.

An Insane General

An insane general may disrupt processing in many ways. First of all, the insane general may simply ignore all user requests for transaction scheduling. Such a failure trivially satisfies property S1 because no transactions are scheduled on any node.

Now, suppose the general sends a request to L_1 and nothing to L_2. Clearly, if the general's request is not received by L_1 before time $T_e - M_{delay} - C_{diff}$ (on L_1's clock) then it will be ignored and property S1 holds trivially. If, on the other hand, L_1 receives the request on time, then the confirmation will be sent before $T_e - M_{delay} - C_{diff}$ on L_1's clock. Therefore, by assumption A1, the confirmation will be received by L_2 at time $T_e - C_{diff}$ on "L_1's" clock. However, by assumption A2, L_1's and L_2's clocks differ by at most C_{diff} units. Therefore, the confirmation message is received before T_e on L_2's clock and is accepted. Consequently, the transaction will be scheduled by both L_1 and L_2 when its timestamp expires, because neither lieutenant will receive corresponding request and confirmation messages.

An insane general may also send different request messages, bearing the same expiration time, to each lieutenant. As in the previous cases, if a lieutenant accepts one of the request messages, then the other will accept a corresponding confirmation message. Consequently, both lieutenants will see conflicting transactions bearing the same expiration time and will ignore them.

Appendix 3. Basic Scheduling Algorithm

```
on receipt of a transaction request from a user:
      reliable_broadcast (transaction)


when a transaction in the final set has an expired timestamp:
      schedule (transaction)
```

Appendix 4. Premature Scheduling Algorithm

```
on receipt of a transaction request from a user:
    reliable_broadcast (transaction)


when a transaction in the final set has an expired timestamp:
    schedule (transaction)


when a transaction is added to the final set:
    count = 0
    for all transactions in final set
        if trans.seqn == next_seqn_from(trans.origin)
        then
            count = count + 1

    If count == 3
    then
        schedule (transaction_with_earliest_timestamp)
```

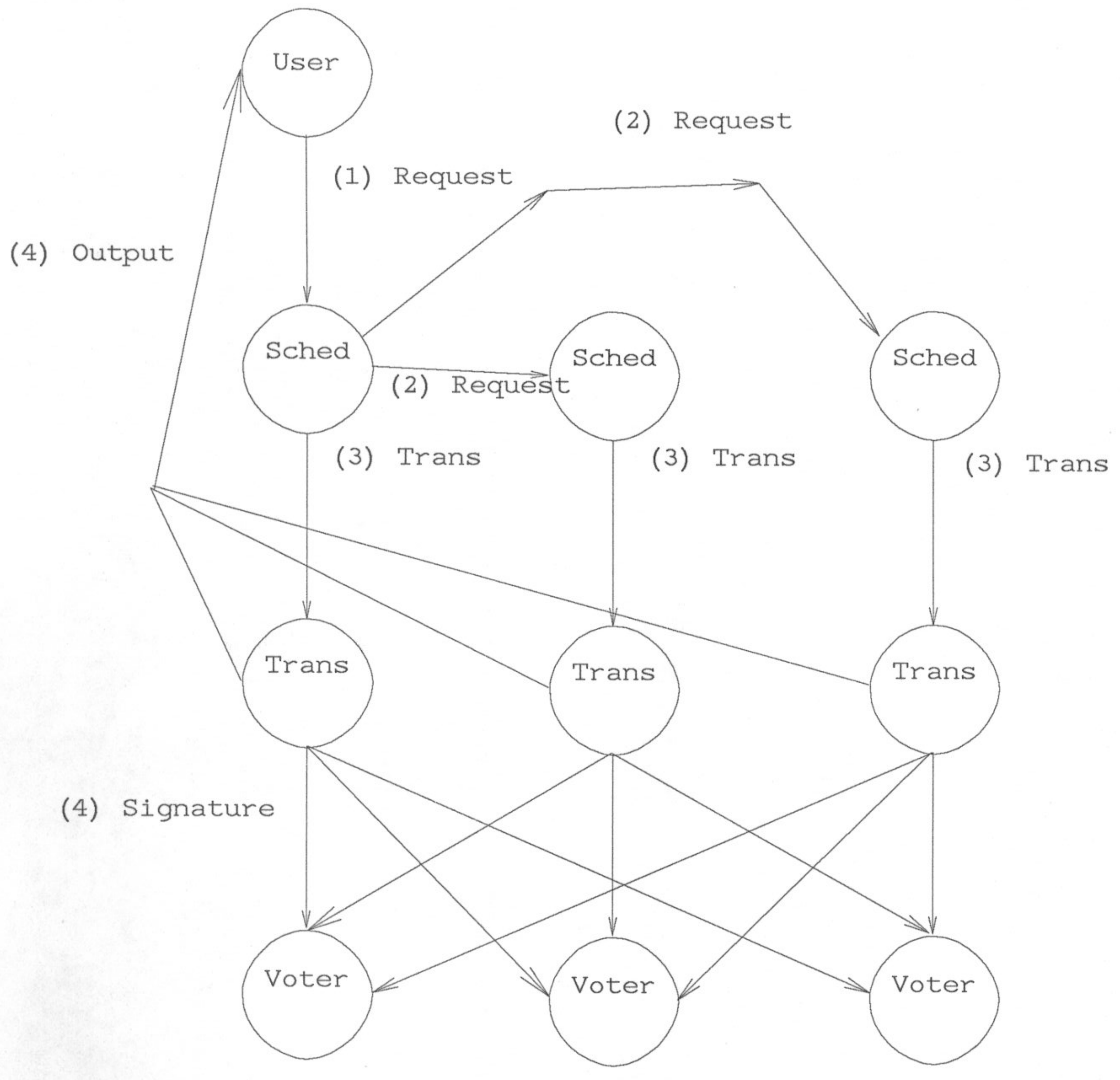

Figure 1. Normal Transaction Processing

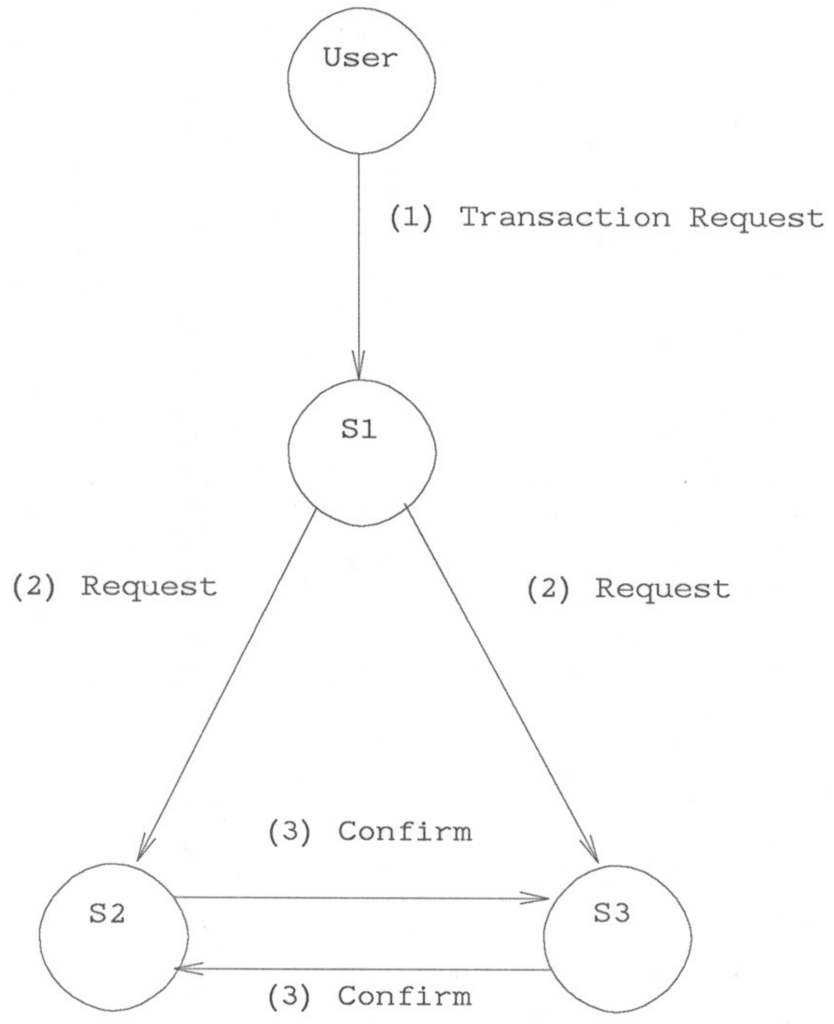

Figure 2. Reliable Broadcast

Final Transaction Set			
Tran	Node	Time	Seqn
T_1	1	1300	11
T_2	2	1385	5
T_3	3	1385	7
T_4	1	1400	12

Next Seqn Expected	
Node	Seqn
1	11
2	5
3	6

Figure 3. Premature Scheduling Example

Final Transaction Set

Tran	Node	Time	Seqn
T_1	1	1300	11

Next Seqn Expected

Node	Seqn
1	11
2	5
3	6

a) Before Nulls Are Sent

Final Transaction Set

Tran	Node	Time	Seqn
T_1	1	1300	11
N_1	2	1400	5
N_2	3	1430	6

Next Seqn Expected

Node	Seqn
1	11
2	5
3	6

b) After Nulls Are Received

Figure 4. Premature Scheduling Example

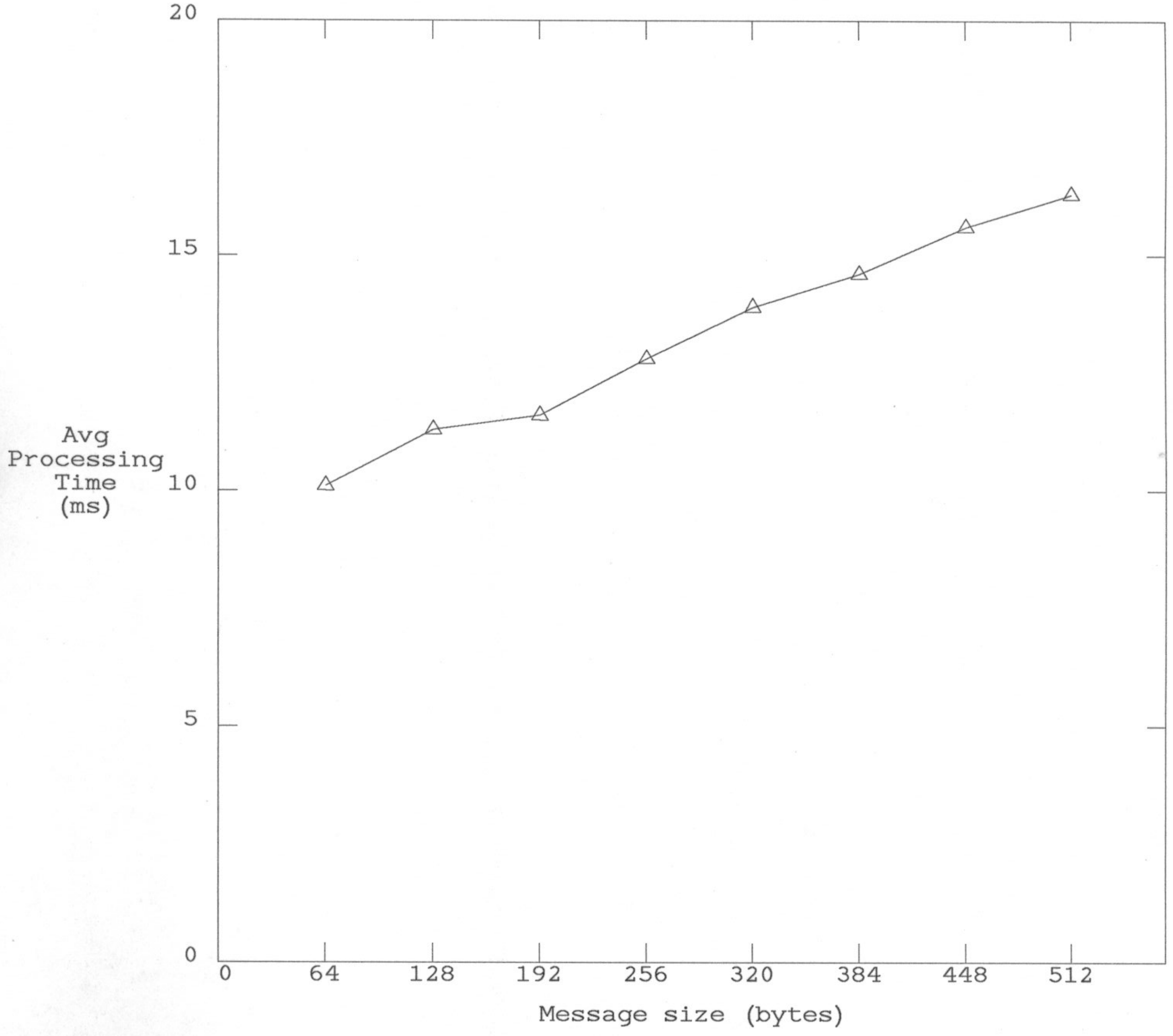

Figure 5. Message Processing Time

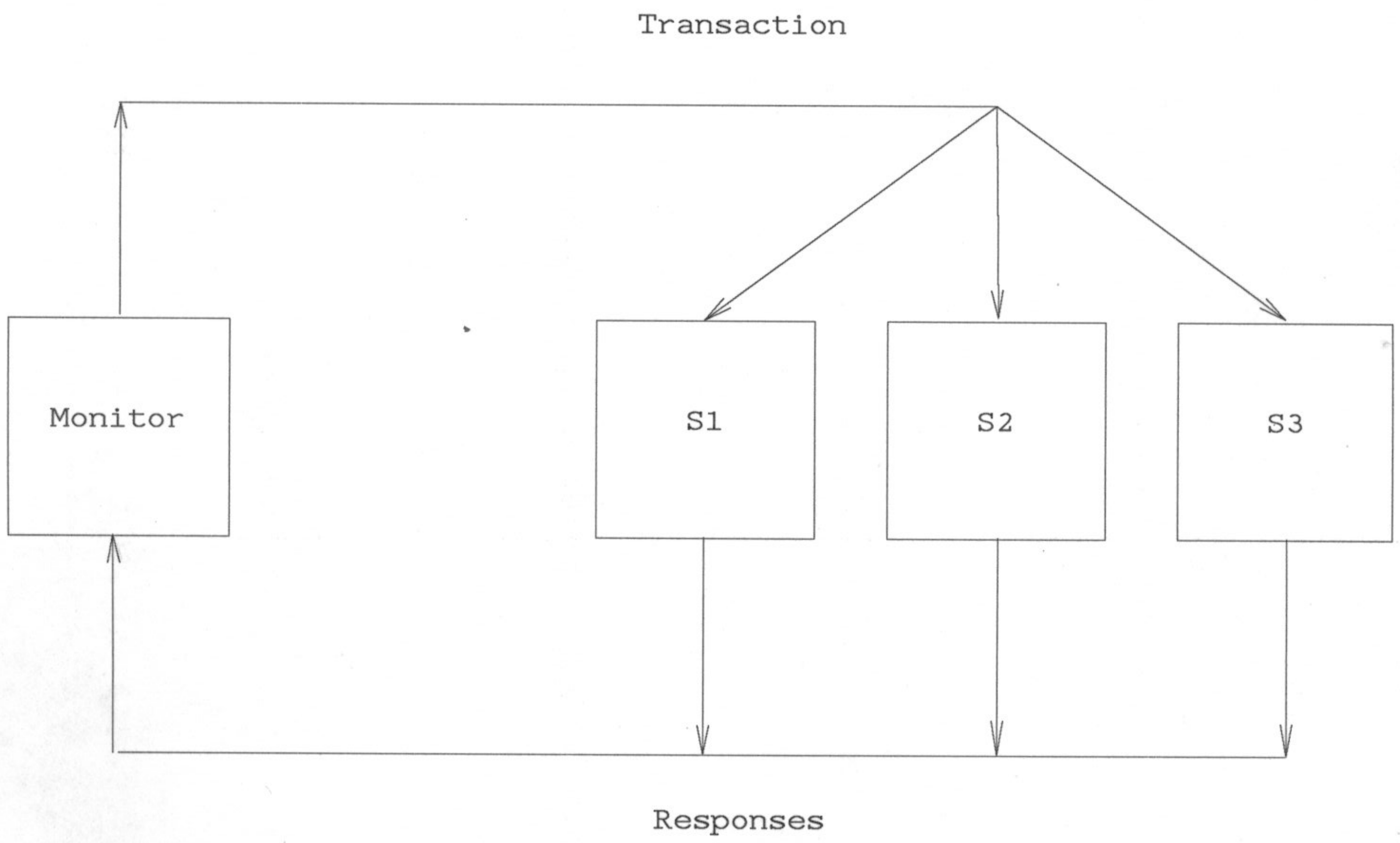

Figure 6. Experimental System

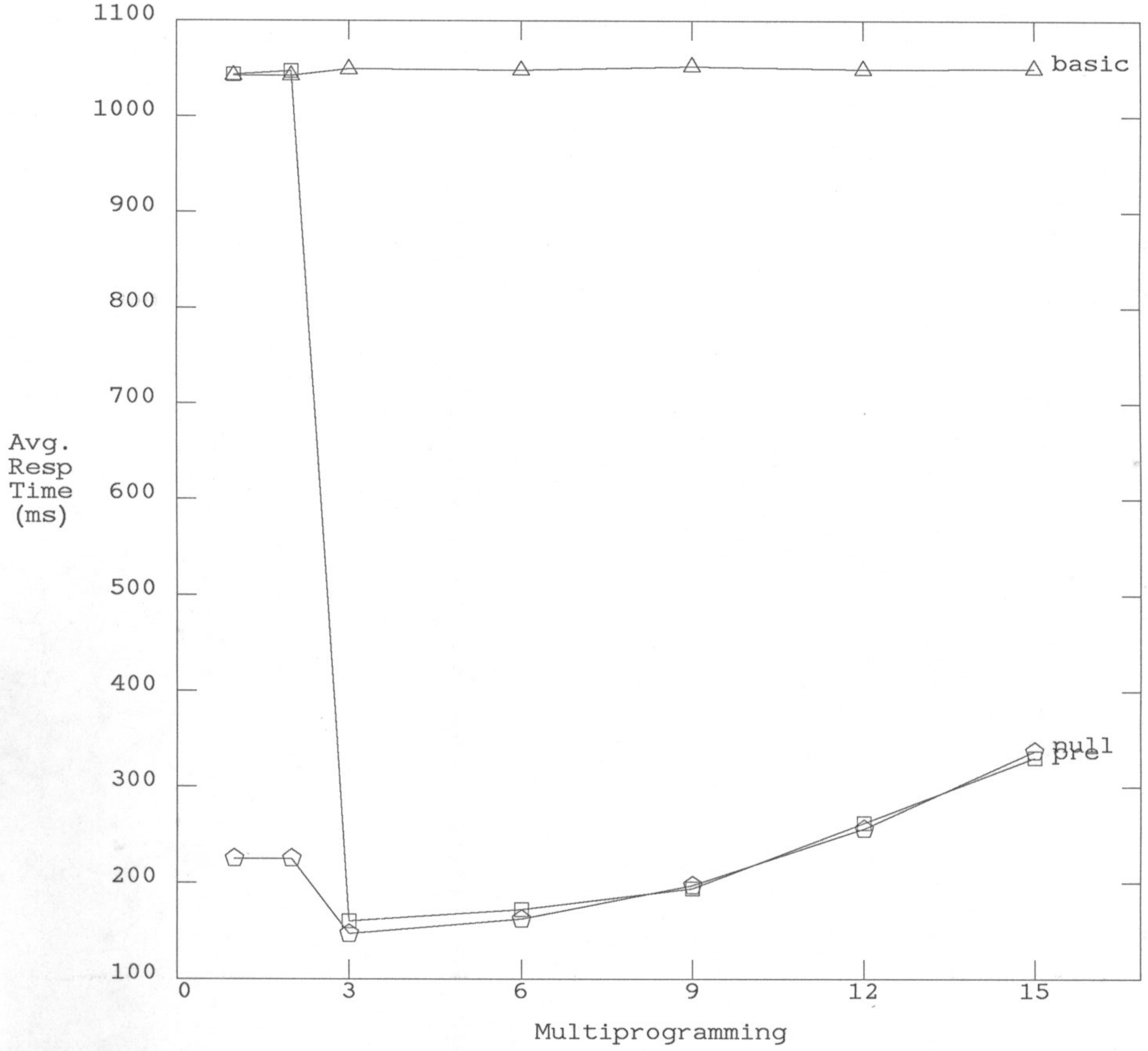

Figure 7a. Comparison of Scheduling Algorithms

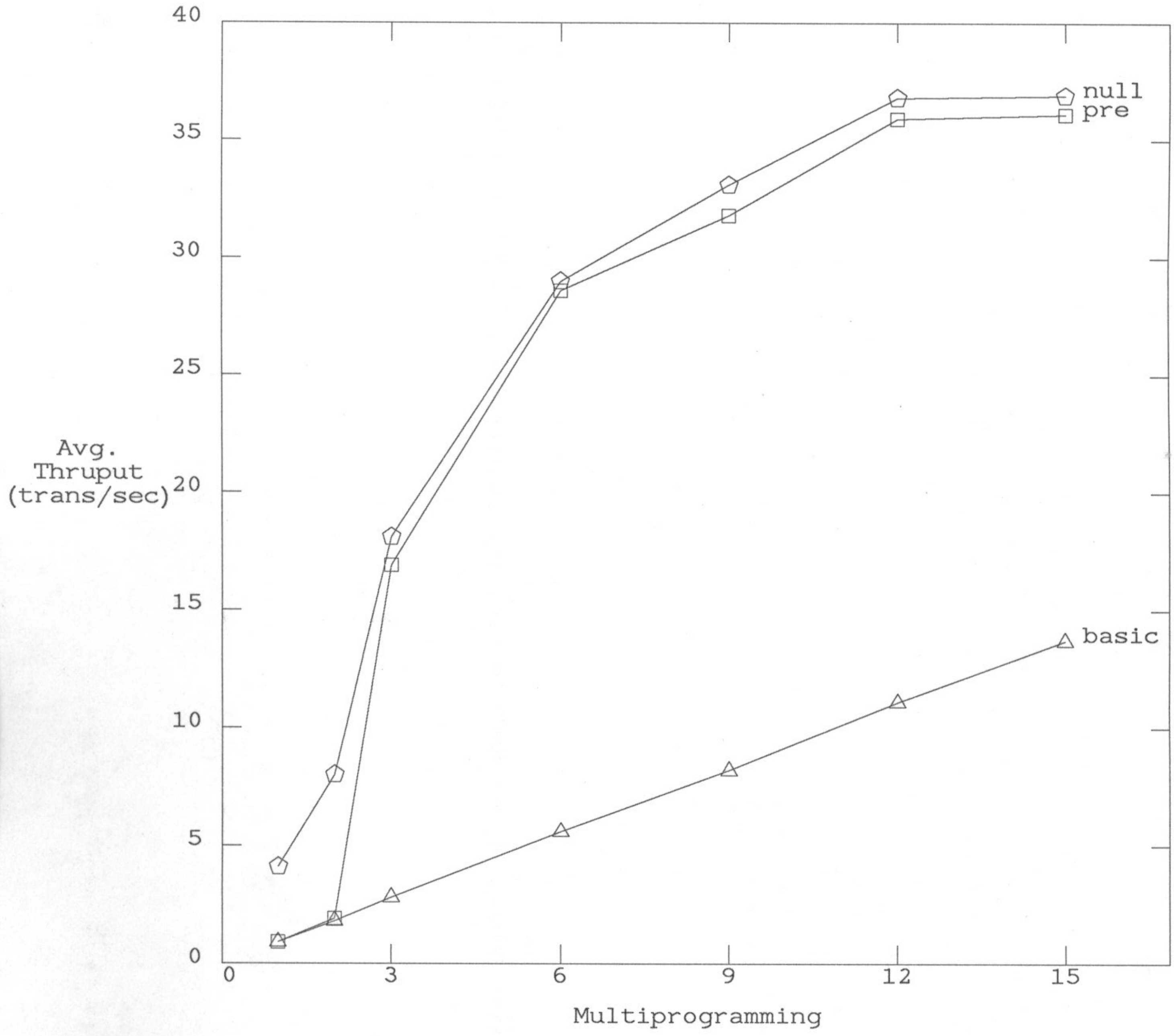

Figure 7b. Comparison of Scheduling Algorithms

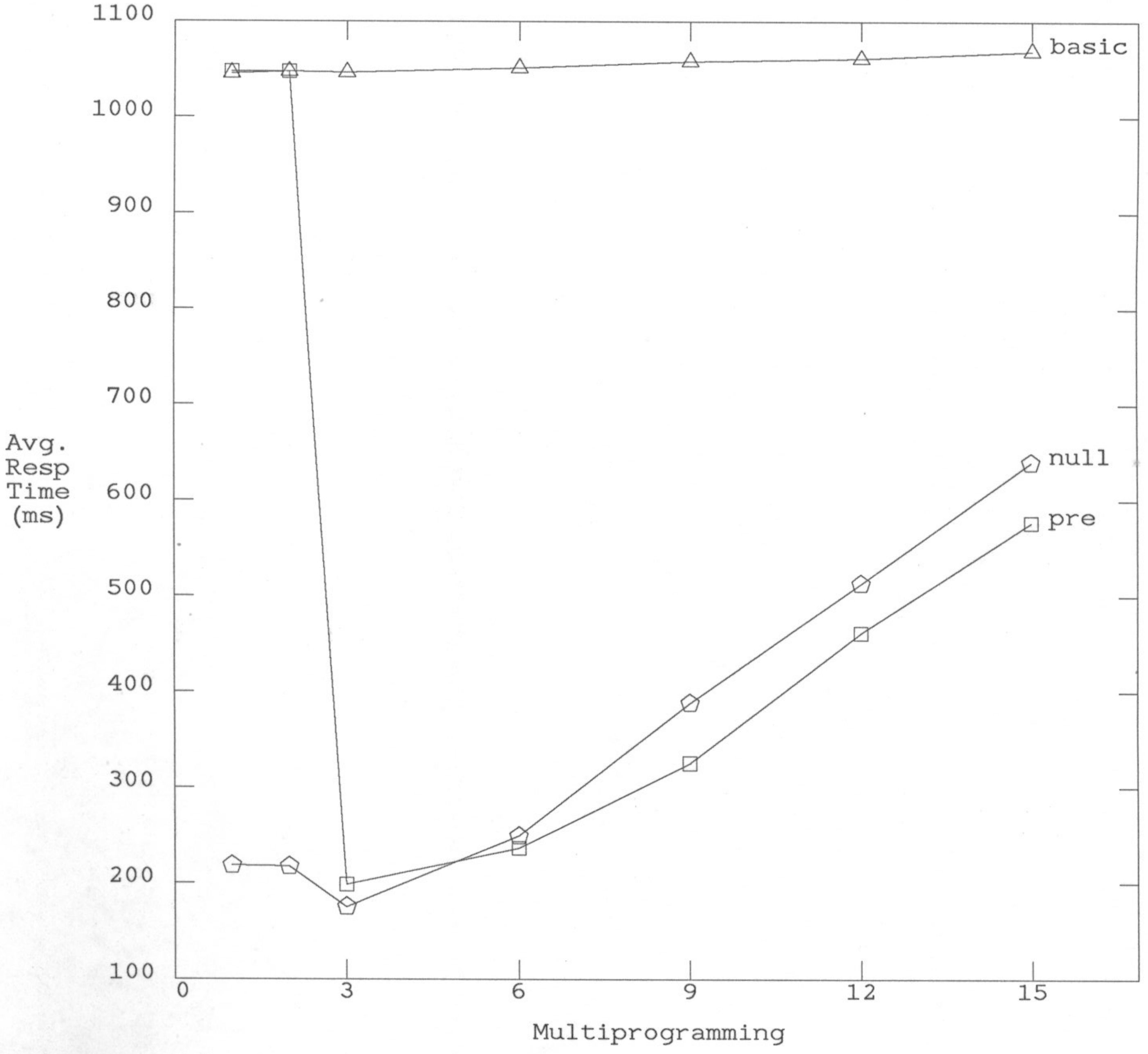

Figure 8a. Impact of Network Load

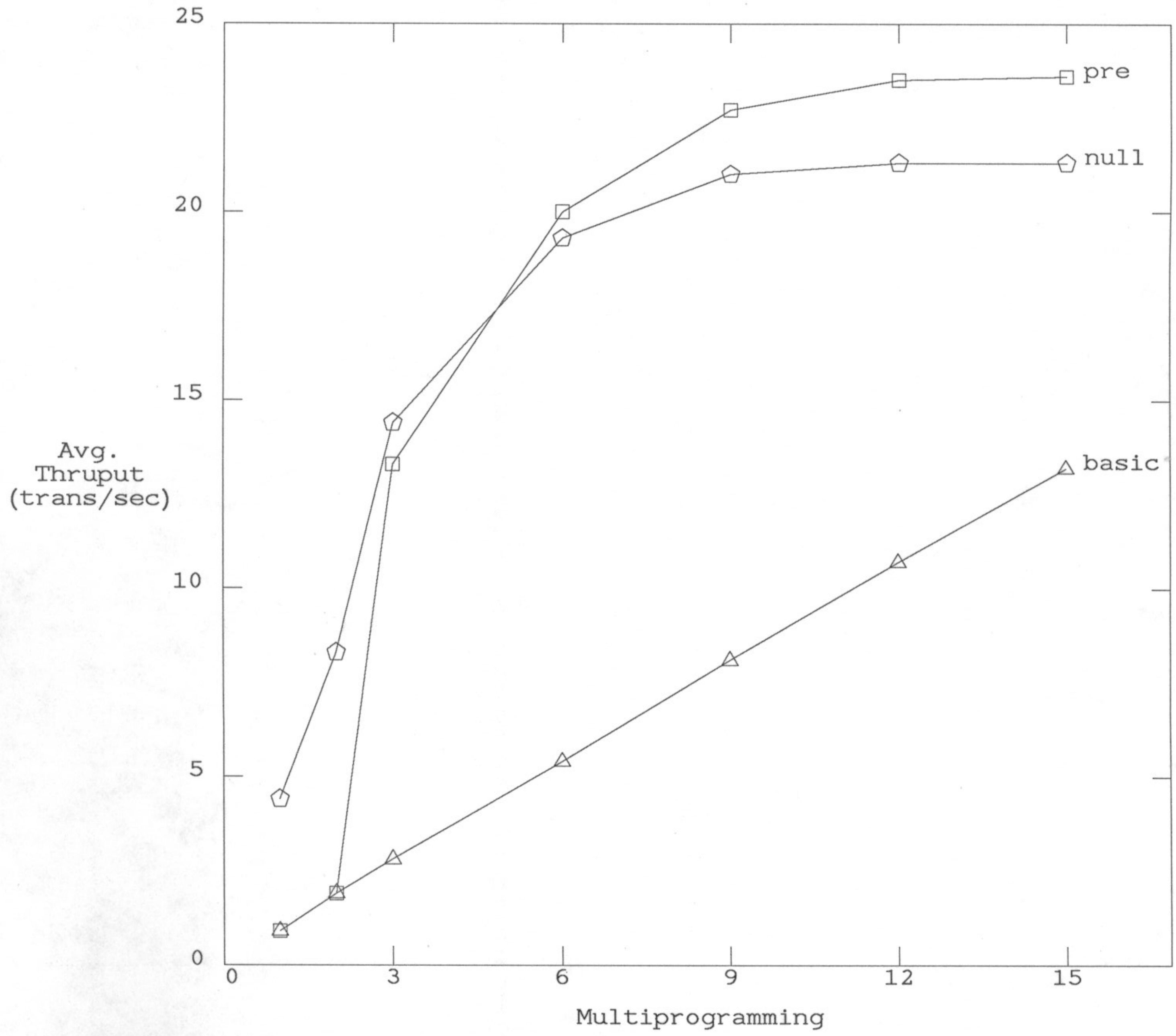

Figure 8b. Impact of Network Load

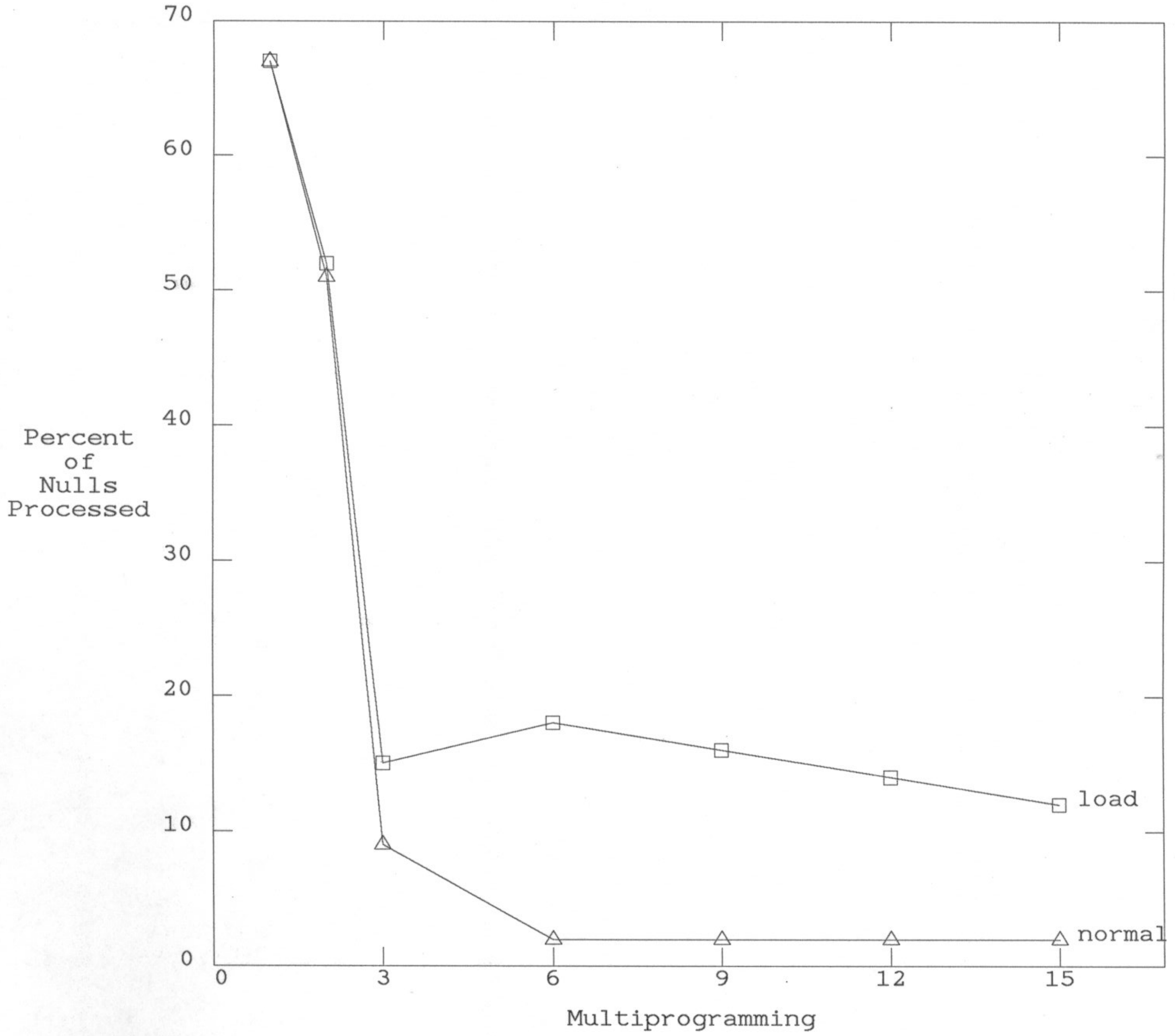

Figure 9. Impact of Network Load on Null Algorithm

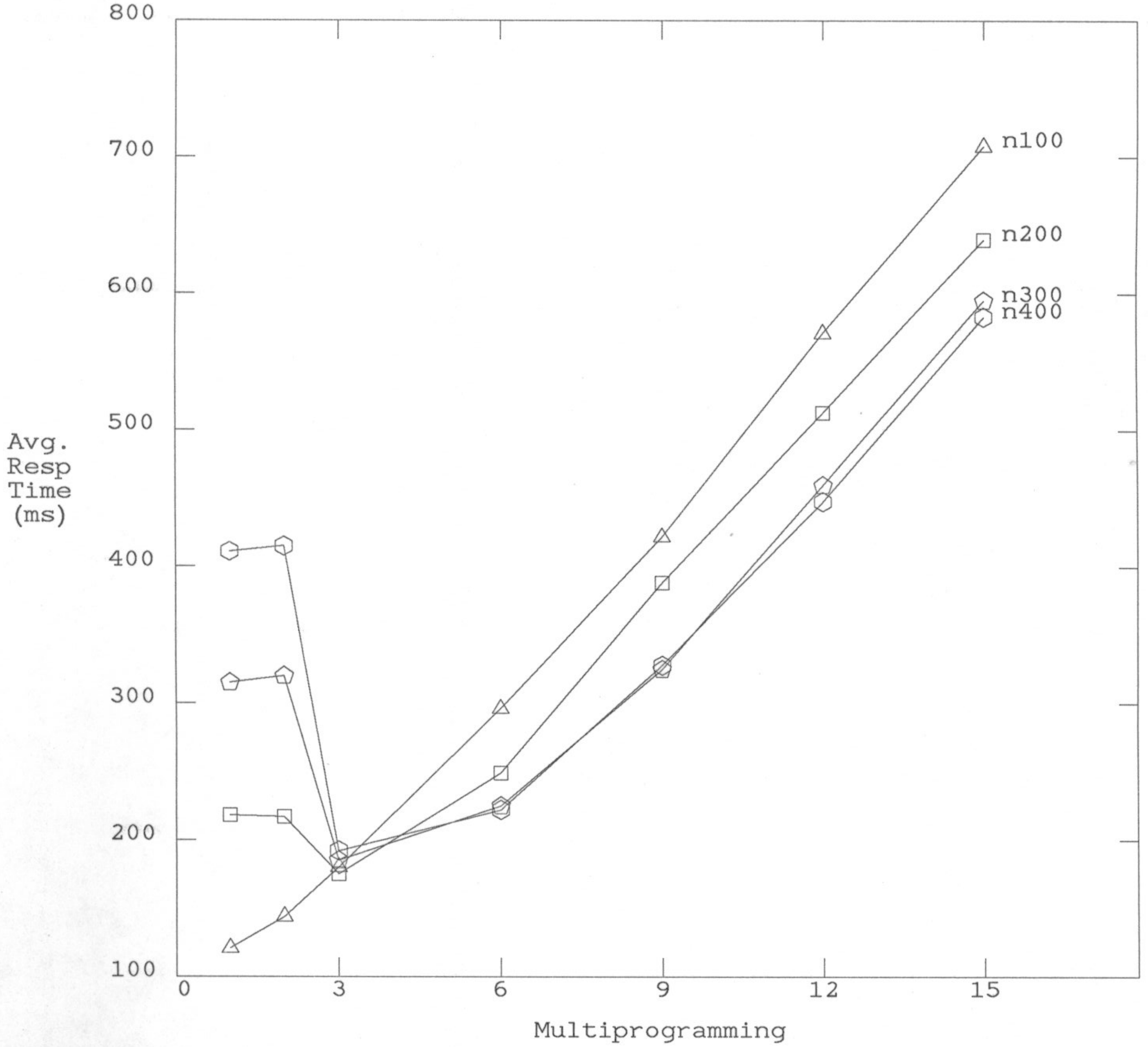

Figure 10a. Effect of Null Delay

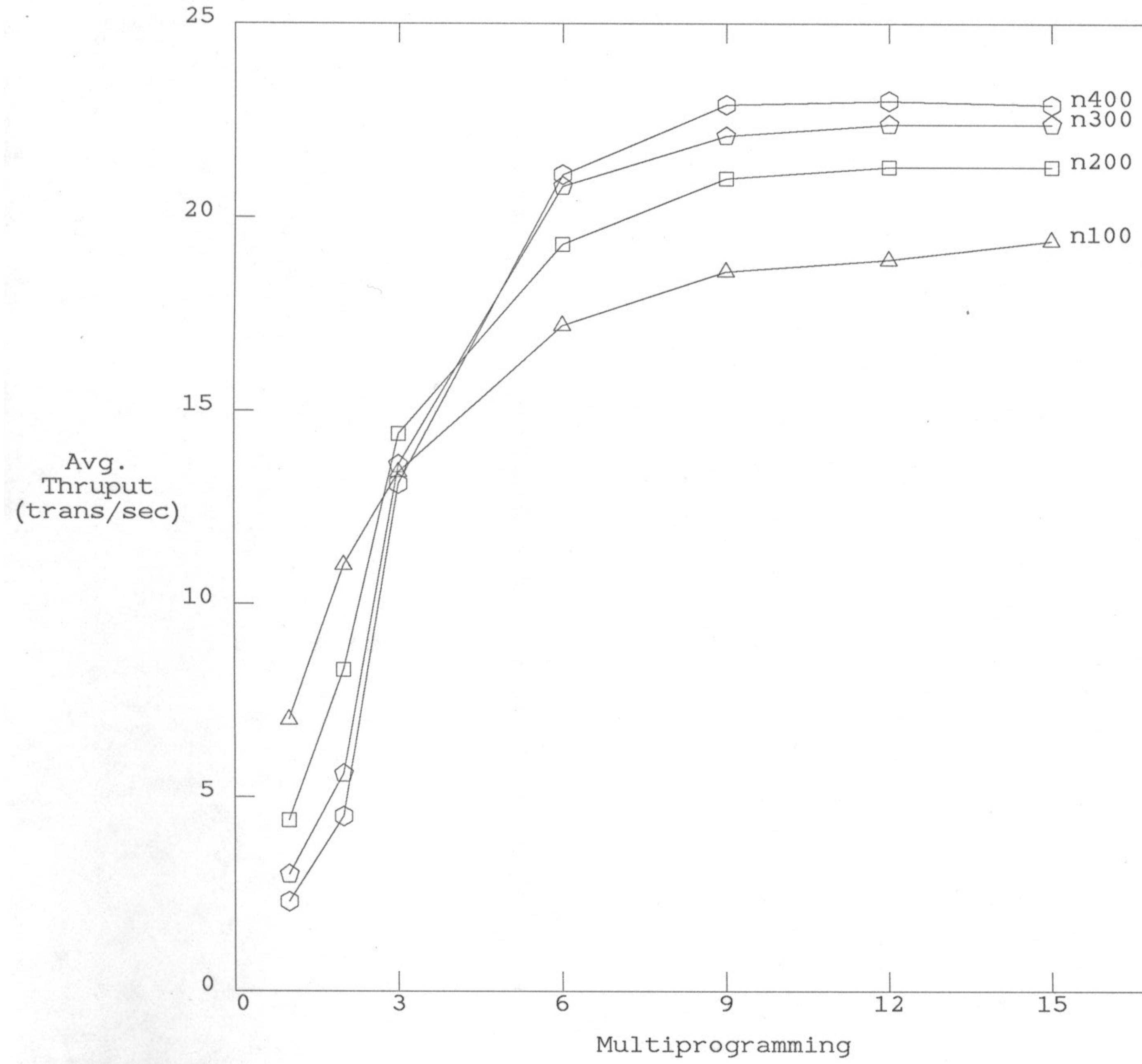

Figure 10b. Effect of Null Delay

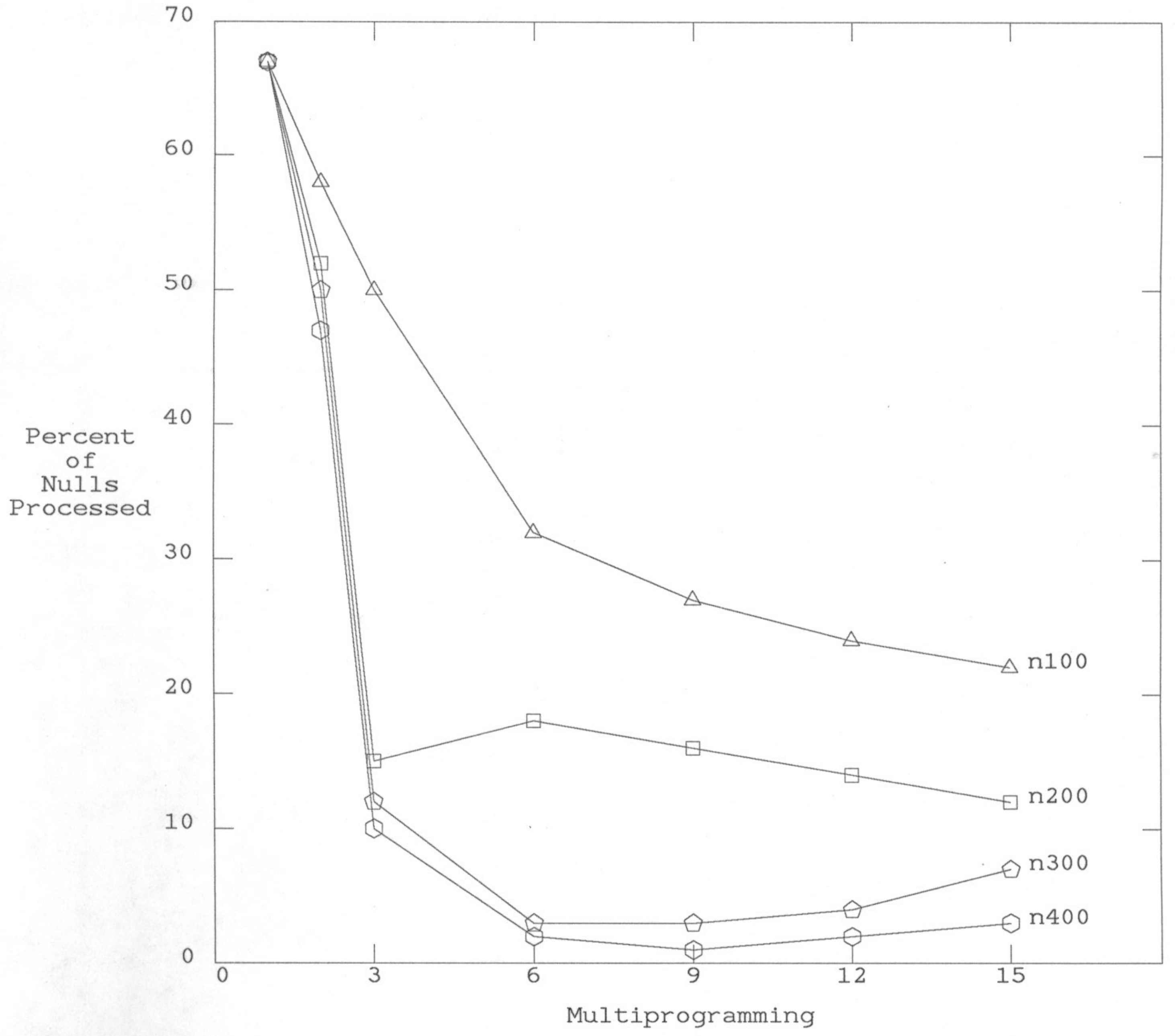

Figure 10c. Effect of Null Delay

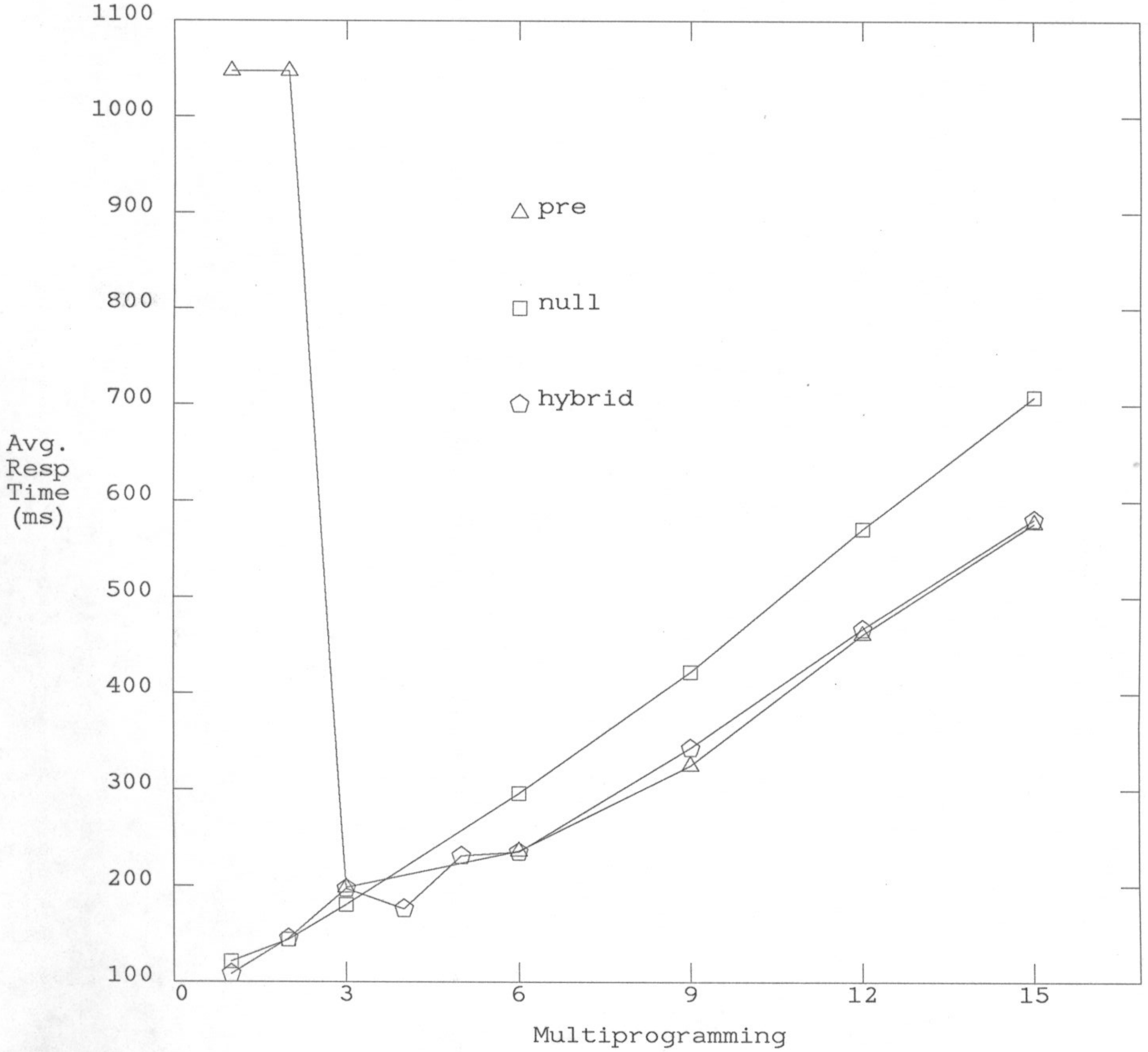

Figure 11a. Comparison of Premature, Null, and Fixed-Cutoff Hybrid

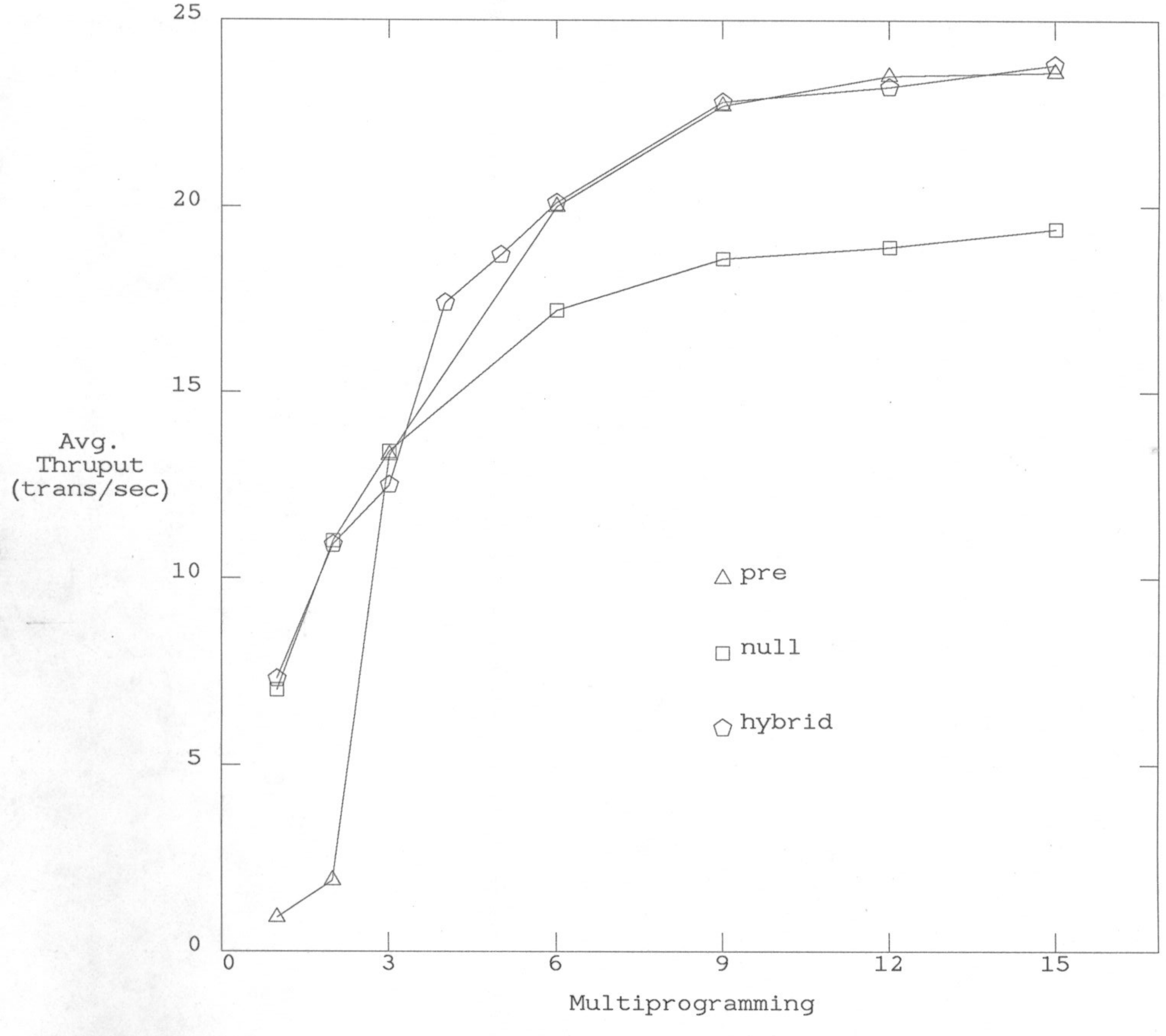

Figure 11b. Comparison of Premature, Null, and Fixed-Cutoff Hybrid

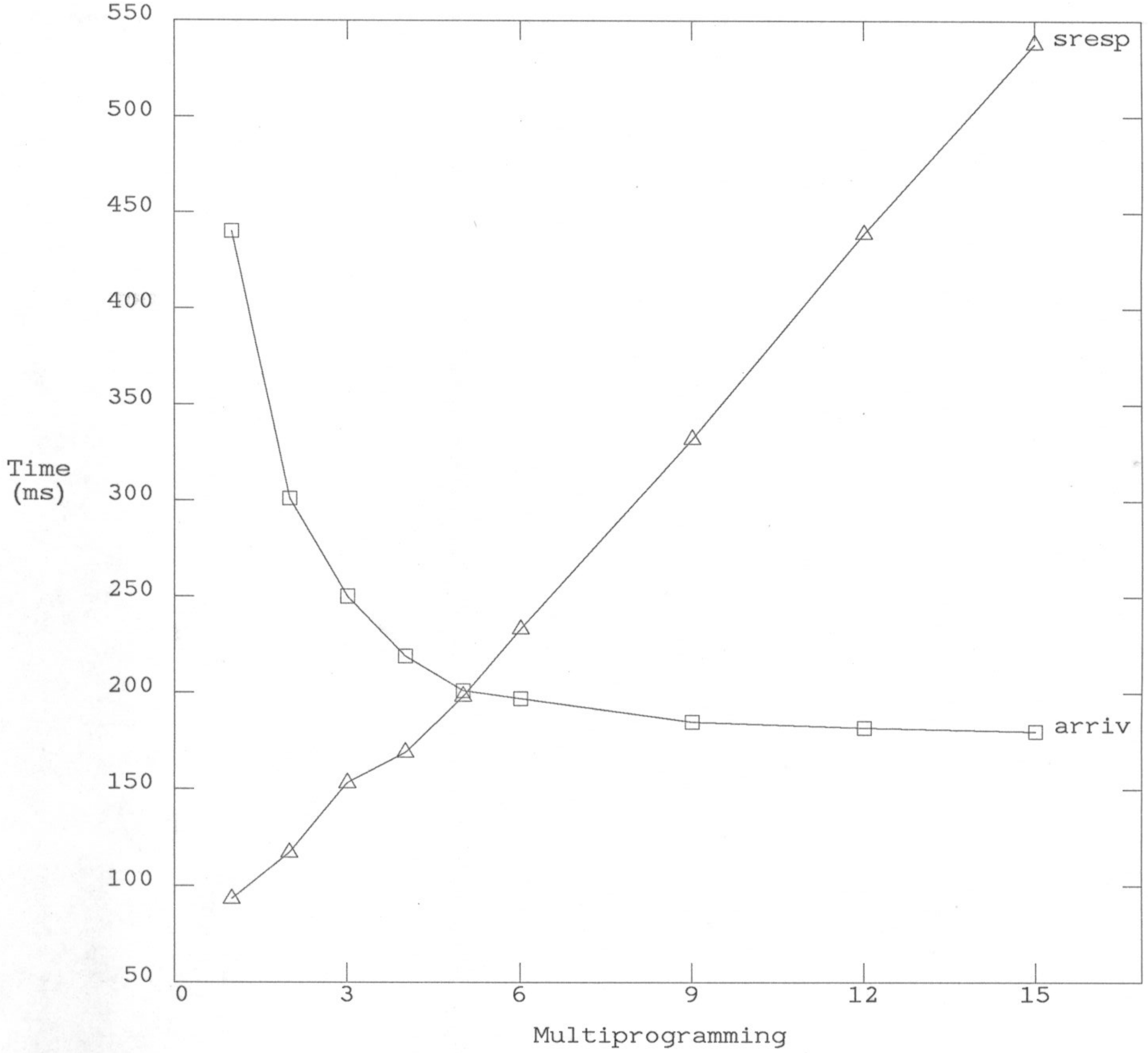

Figure 12. Sched Resp vs. Inter-arrival Time (Null Algorithm)

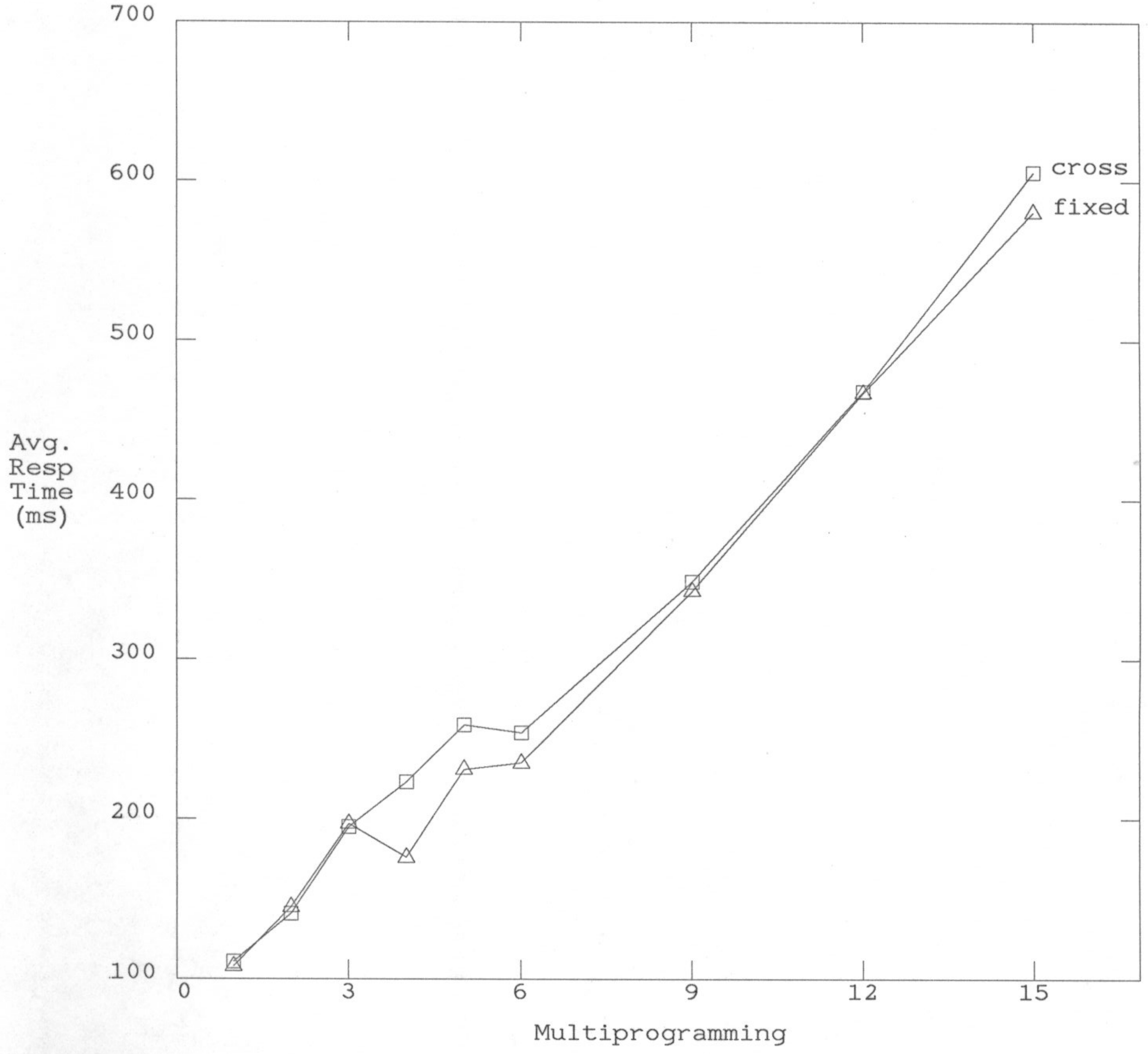

Figure 13a. Comparison of Fixed-Cutoff and Crossover Hybrids

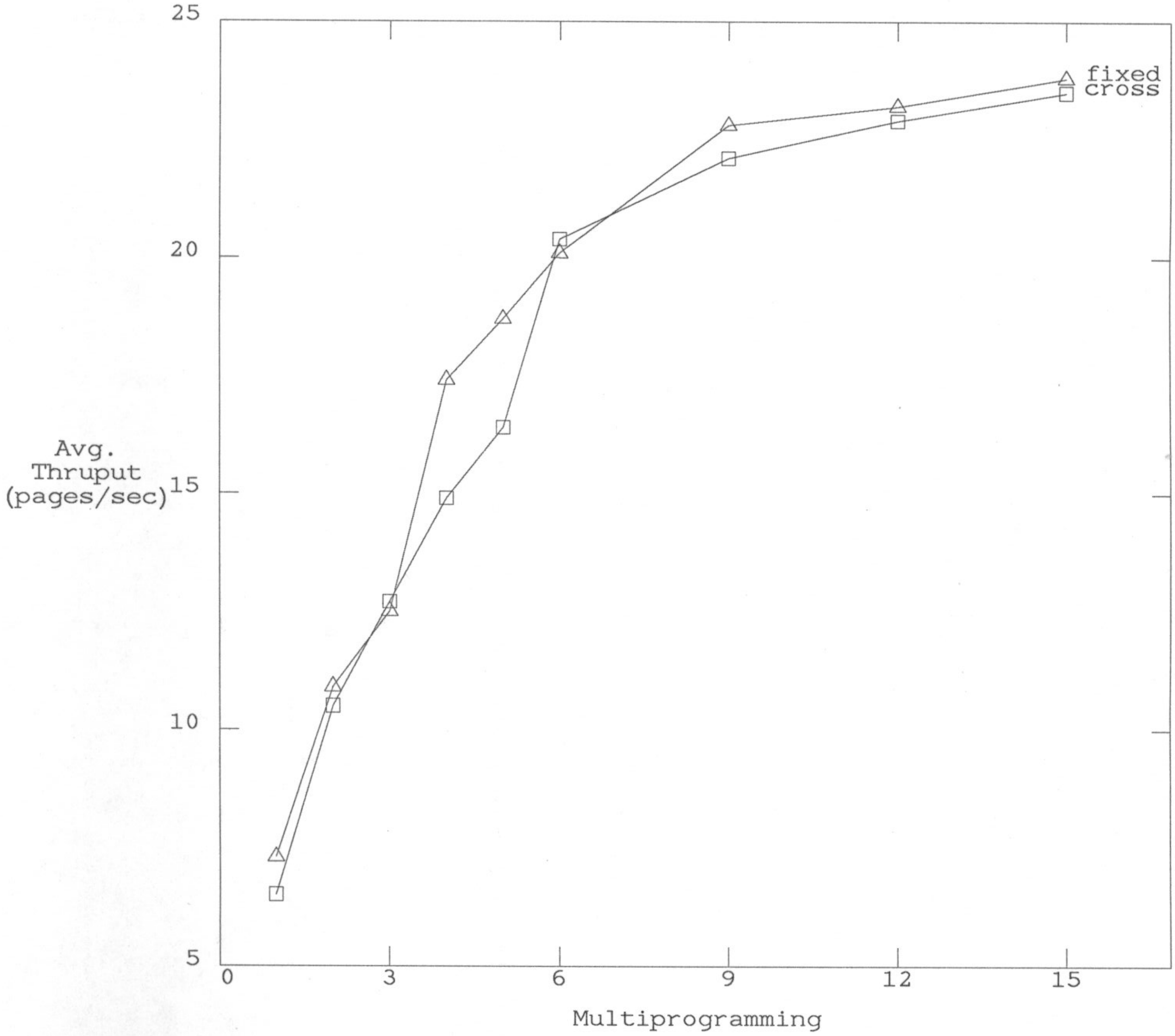

Figure 13b. Comparison of Fixed-Cutoff and Crossover Hybrids

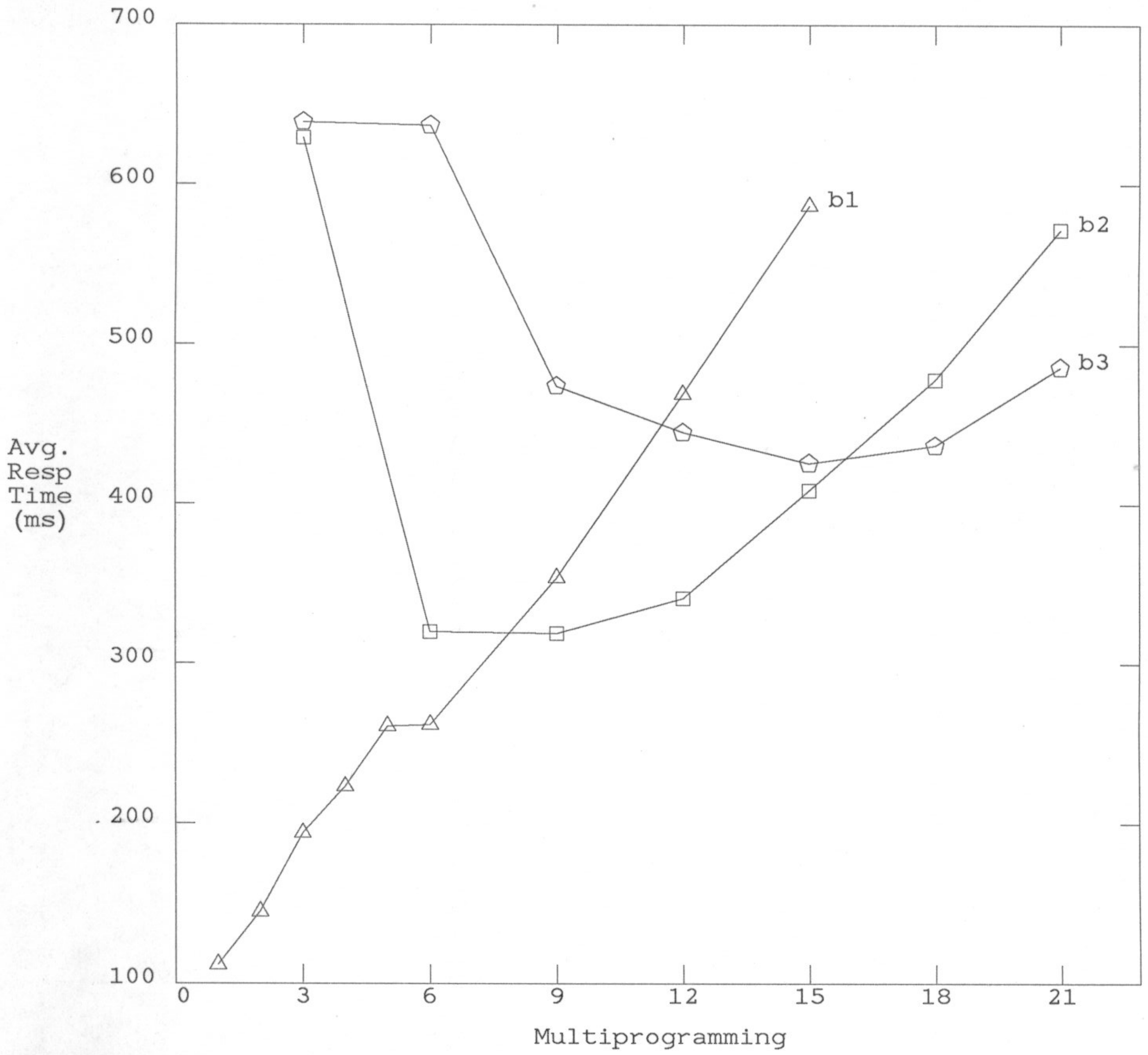

Figure 14a. Impact of Batching on Crossover Hybrid

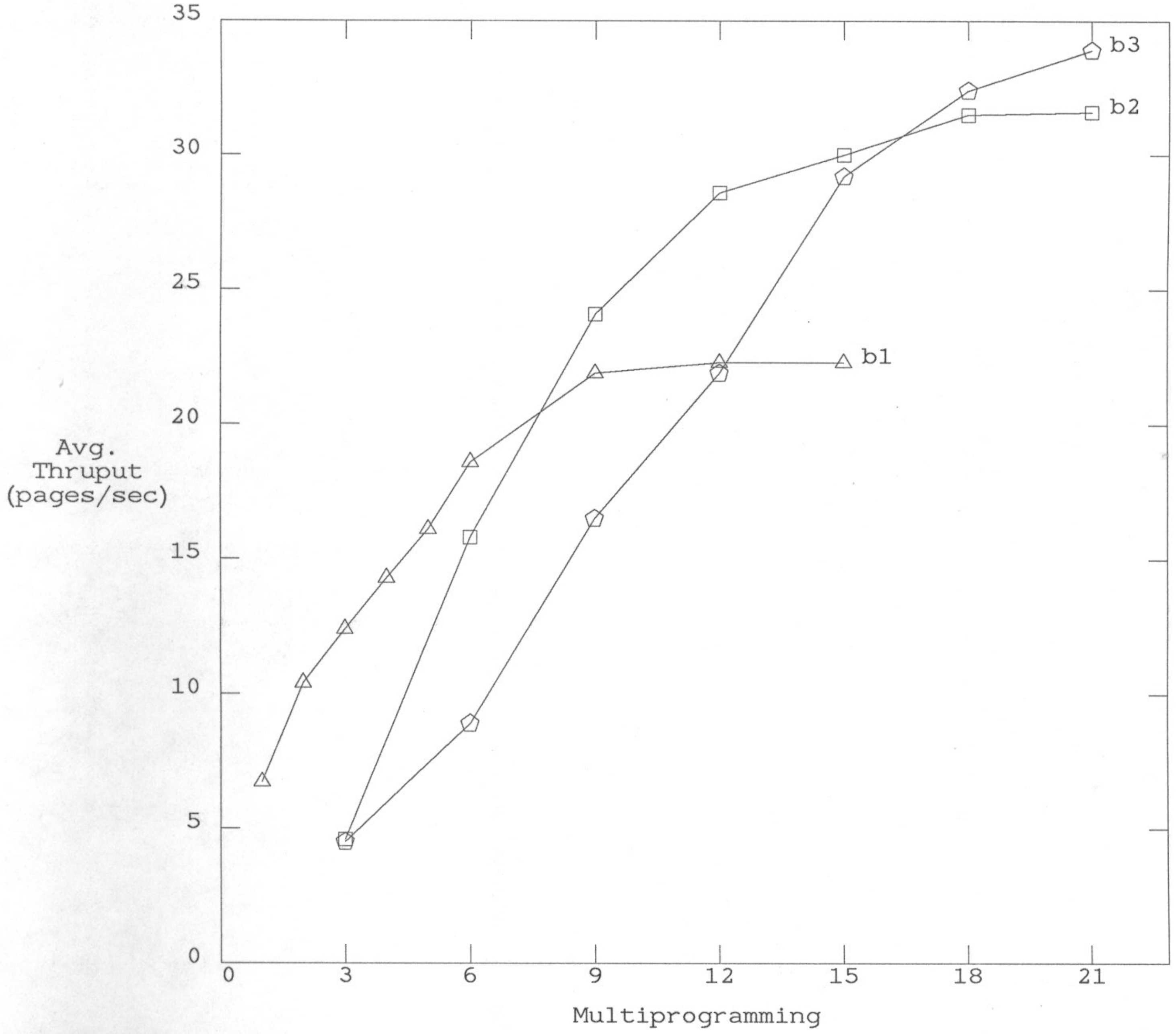

Figure 14b. Impact of Batching on Crossover Hybrid

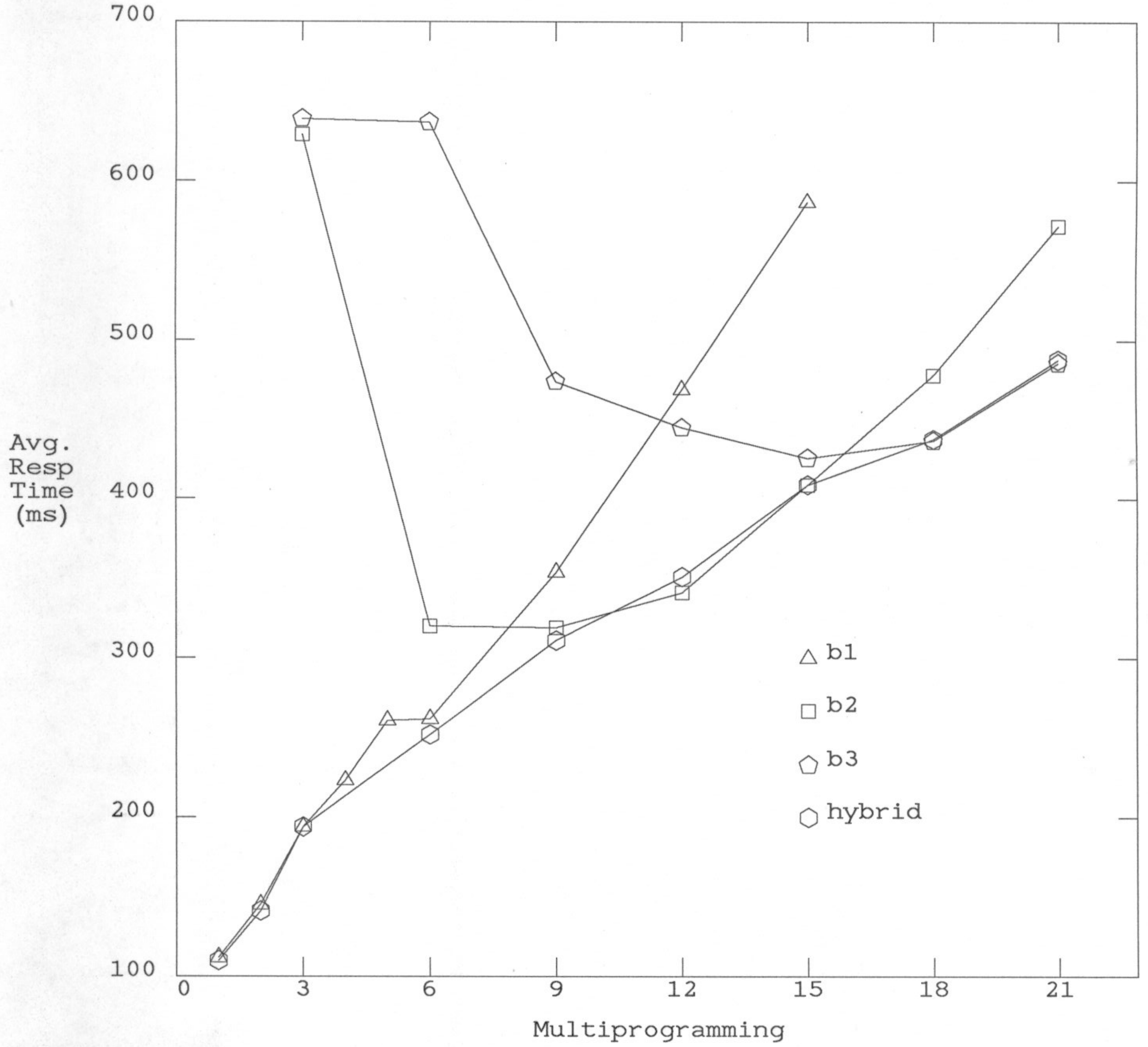

Figure 15a. Comparison of Batch and Crossover Hybrids

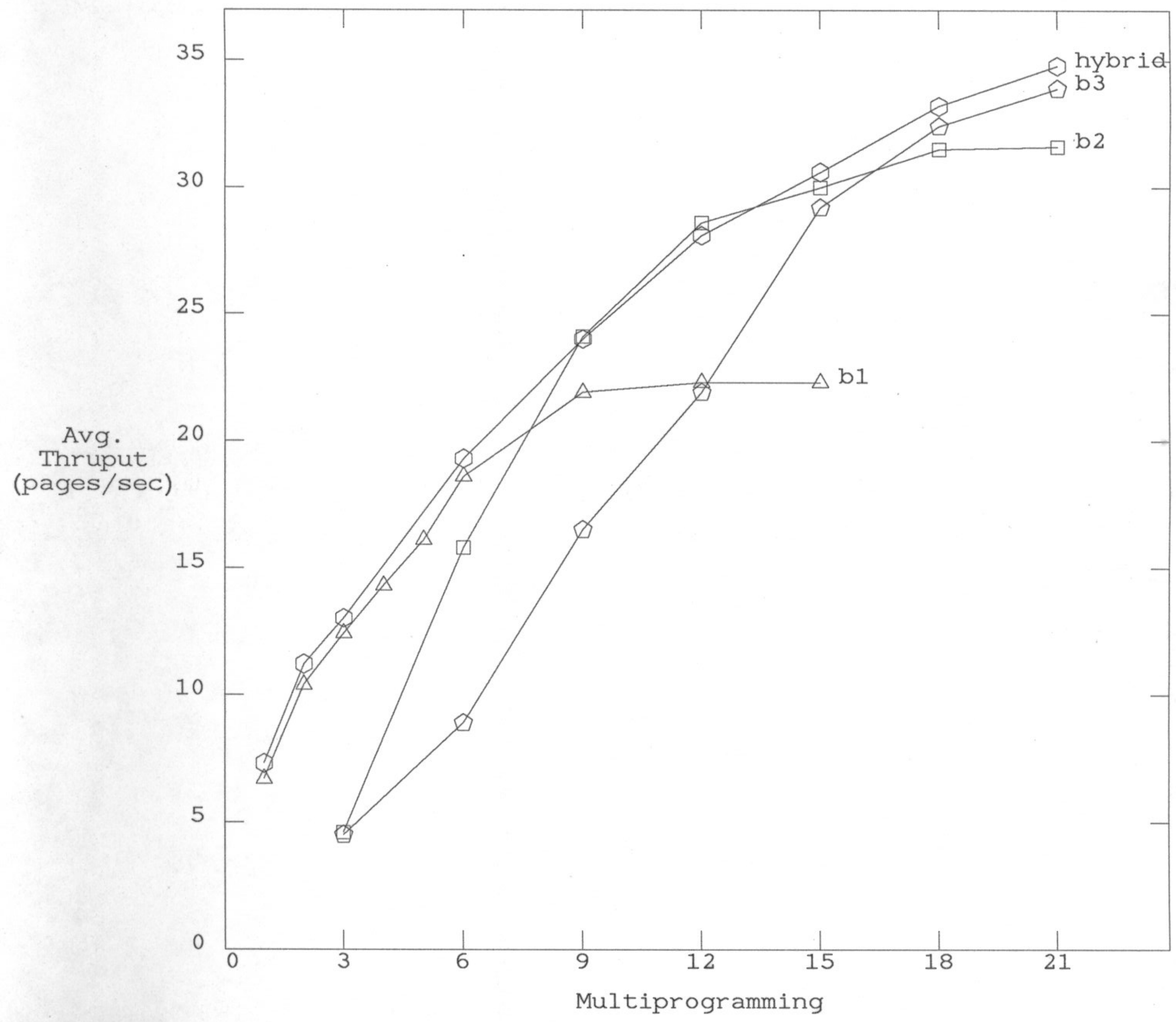

Figure 15b. Comparison of Batch and Crossover Hybrids