Combining Windows

A Performance Evaluation of Design Options

Philip Bitar

Doctoral Dissertation
Filed 92/12/18
Refinements completed 93/02/10

Technical Report UCB//CSD-93-727
Computer Science Division
University of California
Berkeley, California 94720

Abstract

A *combining window* is an interval of time in a combining node during which incoming requests are gathered in the node in order to combine them into a single outgoing request. Our thesis is that a combining window is necessary in order to realize the dual forms of concurrency — execution and storage concurrency — that a combining tree is designed to achieve. *Execution concurrency* among the nodes at each level of a combining tree is necessary for the tree to achieve the *speed up* that it is designed to give. Without sufficient execution concurrency, the tree will not achieve the desired speed up. *Storage concurrency* among the nodes at each level of a combining tree is necessary for the tree to achieve the *buffer storage* that is required in order to implement the combining of requests. Without sufficient storage concurrency, node buffers will overflow. More specifically, the combining window shows how to bound node buffer size.

For Marie

who has put up with so much

and so little

for so long

Acknowledgements

I thank my advisor Edward A. Lee for the opportunity to work in the Ptolemy research group. This provided a challenging and rewarding research experience in developing system software, as well as providing a vehicle for running simulations.

I thank my colleagues in the Ptolemy group, and the staff that supports our work, for their invaluable contribution, without which I could not have completed my research.

I thank my former advisor Al Despain (now at the University of Southern California) for the earlier opportunity to work in the Aquarius research group, where I pursued the topics of synchronization, coherence, and parallel Prolog execution.

I thank Velvel Kahan for his support during the time that I was without an advisor and needed to find a new advisor, and for his support since that time.

I thank Michel Dubois for his encouragement during the transition period, as I turned from parallel Prolog execution back to synchronization and coherence, and during the years since then as I have continued to develop the theory of synchronization and coherence and publish pieces of it.

I thank IEOR Prof. Ronald Wolff, whose classes in stochastic models and queuing theory provided the mathematical theory that underlies this dissertation. I count these two classes as the most interesting classes that I took at Berkeley.

I thank my committee members for their criticism and insight, which stimulated notable improvements in the dissertation.

I appreciate the support of my mother and (before he died) my father, who provided unflagging moral support and, when needed, financial support, for my education.

I appreciate the encouragement of many friends and relatives, including my brothers David, Roger, and Byron, my sister Marilyn, my friends Steve, Jay, and Ted, and friends from church.

I especially thank my wife Marie for her devoted love throughout a surprisingly lengthy graduate education. I am grateful to Marie, and to our children Brittany and Owen, who were born during this time. They endured the family life of a stressed and busy husband and daddy.

And now, I say, let's get on with real living! Let's get on with the family life that it's time to stop dreaming about and time to start living — a family life that allows us to look more seriously beyond our own needs, to the needs of others.

This research was partially funded by Sony Corp., Hitachi Ltd., Hitachi America, and NSF grants MIP-8657523 and CDA-8722788.

Table of Contents

1. The Thesis

1.1. The Problem

In a combining tree, the combining of requests requires their temporal proximity.
Does the proximity need to be ensured?

The answer is *yes.* In order to ensure the parallel execution, and hence speed up, that the combining tree is designed to give, it is necessary to observe a *combining window*, an interval of time in a combining node during which incoming requests are gathered in the node in order to combine them into a single outgoing request.

At first glance, one might rebut, ‘‘But this will slow the computation down.’’ In fact, it will not slow the computation down if the window has an appropriate size, but to fail to observe a combining window may slow the computation down. For without a combining window, sufficient combining may not occur at the wider levels of the tree (closer to the leaves), so the processing demands may be concentrated on the smaller number of nodes closer to the root. The result is that the request arrival rate at these nodes may exceed their service rate, so the combining tree may not be able to obtain the speed up that it is designed to achieve.

The problem of speed up that is solved by sufficient *parallel execution* has a dual problem: the problem of node buffer space, which is solved by sufficient *parallel storage.* That is, if the arrival rate at a node exceeds its service rate, not only will processing rate be too slow, but the node must also store all of those requests, so its buffer space may be exceeded. Thus, the combining window allows us to bound node buffer size.

1.2 The Thesis

Why are combining windows necessary?

A *combining window* is an interval of time in a combining node during which incoming requests are gathered in the node in order to combine them into a single outgoing request.

- **The thesis:** A combining window is necessary in order to realize the dual forms of concurrency — execution and storage concurrency — that a combining tree is designed to achieve.
  - □ *Execution concurrency* among the nodes at each level of a combining tree is necessary for the tree to achieve the *speed up* that it is designed to give. Without sufficient execution concurrency, the tree will not achieve the desired speed up.
  - □ *Storage concurrency* among the nodes at each level of a combining tree is necessary for the tree to achieve the *buffer storage* that is required in order to implement the combining of requests. Without sufficient storage concurrency, node buffers will overflow.

1.3. Overview

What is in store for the reader of this dissertation?

Chapter 2 describes the design of asynchronous MIMD combining trees — their motivation, their structure, their parameters — and illustrates these principles using fetch-and-add. We develop a queuing perspective on combining trees, and on that basis explain exactly why combining windows are necessary. We also present a bound for mean node buffer size in terms of asymptotic notation. *Chapter 3* reviews the literature on MIMD combining trees that is relevant to combining windows. *Chapter 4* presents an analytic solution to the node buffer problem, based on queuing networks. *Chapter 5* describes the simulation model, along with the simulation experiments and their results. *Chapter 6*, finally, presents our conclusions, along with avenues for future research.

2. Combining Tree Design

The concept of combining tree was invented by the Ultracomputer designers and developed by them in the context of hardware combining (Gottlieb et al. 1983a, 1983b). The concept of software combining tree was introduced by Yew et al. (1987) for polling-based busy wait, while Goodman et al. (1989) addressed the issue of generalizing software combining trees to arbitrary combinable operations.

In this chapter, we build on the earlier work by developing the notion of combining tree for asynchronous MIMD architecture in a novel way and independent of physical implementation, in order to make the essential aspects clear and to make implementation options clear. The essential aspects may be explained in terms of *synchronization concepts* and in terms of *queuing constructs.* (Note that our first presentation of these concepts was in Bitar 1990a, and our reference for stochastic modeling and queuing theory is Wolff 1989).

2.1. Basic Concepts

Where do we start in trying to understand combining trees?

Consider an associative binary operation '$\circ$' on i terms. A combining tree may be used to speed up this kind of computation through parallel execution, reducing the execution time from $\Theta(i)$ to $\Theta(\log i)$, as illustrated in Figure 2.1a. In addition, suppose that the operation is also commutative — such as addition — and that each CPU may contribute values at arbitrary times, requesting the addition of a local value x to y. Figure 2.1c illustrates the combining and decombining, while Figure 2.1b provides an abstract representation of Figure 2.1c.

Figure 2.1b represents the idea that each node of the tree combines requests flowing downward and decombines replies flowing upward. This defines *two conceptually distinct atomic operations* at each node: *a combining and possible downward sending operation*, and *a decombining and upward sending operation.* Specifically, when a node receives a request, if there is another request with which it can be combined in the combining buffer, the incoming request is combined with the other request, making a combined request that contains not only the combined value, but also identifies the original requests so that decombining will be possible. Then the combined request is either retained in the combining buffer for further combining, or else a corresponding request is sent down to the parent node, and the combined request is transferred to the decombining buffer to await the reply.

What was the original motivation for the combining tree?

The *fetch-and-add* paradigm was conceived by the Ultracomputer designers for incrementing a variable I that is used to derive index values for a FIFO queue implemented as a circular array (Gottlieb et al. 1983a, 1983b). Processes may concurrently obtain queue cells by incrementing I using the combining tree and then by taking the remainder modulo the queue size. This allows the processes to concurrently insert into

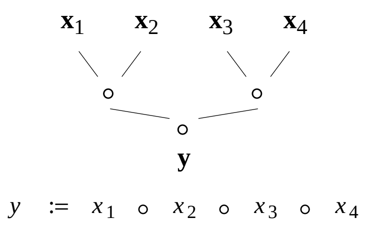
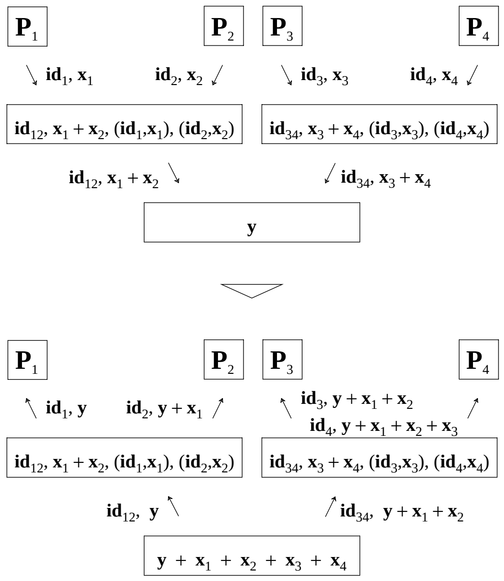


Figure 2.1a. Operation Tree.



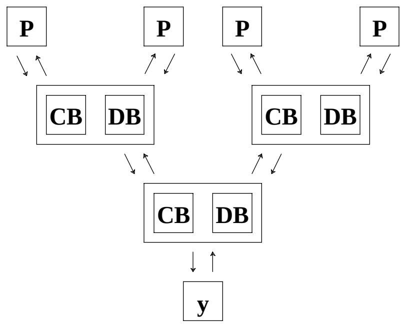
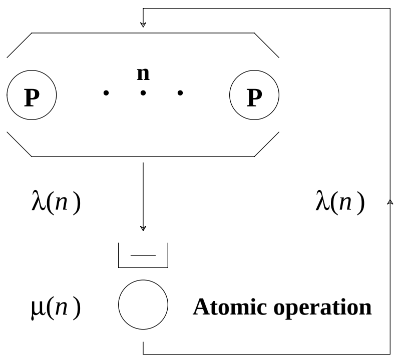


Figure 2.1b. Implementation Structure.

Figure 2.1c. Combining and Decombining.

Down arrow = request {read y then add a value to y}
Up arrow = reply {a value of y}
CB = combining buffer, DB = decombining buffer

Processor i request is fetch-and-add(x_i , y)
Fetch-and-add(x_i , y) = {read y then add x_i to y}
CB, DB are not distinguished. Root node is implicit.



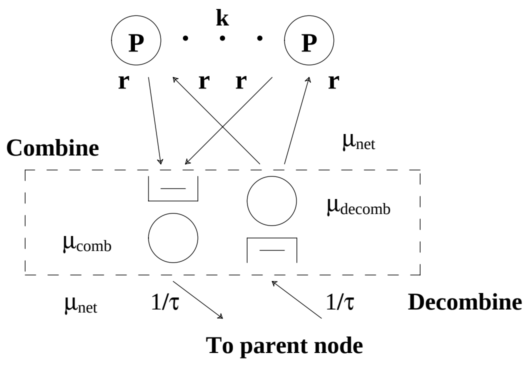


Figure 2.1d. Closed Queuing Network Model.

Figure 2.1e. Node Performance Parameters.

$\lambda(n)$ = steady-state request rate
$\mu(n)$ = operation steady-state service rate
Each processor has fixed number of contexts.

k = fan-in, r = processor steady-state request rate
μ_{comb} = max combining rate, τ = mean combining window size
$k/\mu_{comb} \leq \tau \leq 1/r$

Figure 2.1. Combining Tree.
P = processor

the respective cells. This simple paradigm must, however, be accompanied by a busy-wait queue at each cell, since the cell indices will repeat themselves as I is incremented. At this busy-wait queue, a process will poll the queue state while it waits for access. Similar handling of a delete index D allows parallel deletions, but provision must be made for alternating inserts and deletes on each cell.

In addition, suppose it is desired that a delete request first detect whether there is at least one occupied cell before continuing with the delete algorithm (and thus queuing at some cell to wait for an insert). To maintain the minimum number of occupied cells, a third variable *min* is incremented after each insert and decremented before each delete. Similarly, if it is desired that an insert request first detect whether there is no more than some maximum number of occupied cells (such as the number of cells in the queue less one), a fourth variable *max* is incremented before each insert and decremented after each delete. The details of insert and delete algorithms are shown in Figure 2.2, which provides an expression of algorithms proposed by Gottlieb et al. (1983b). The authors also present other methods of handling busy wait at the cells. Algorithms for sleep-wait queuing (P and V) are presented in the appendix.

What are hardware and software combining trees?

Hardware combining tree. The Ultracomputer designers envisioned using full-service synchronization in a tree implemented completely in hardware, yielding a hardware combining tree (Gottlieb et al. 1983a, 1983b; Gottlieb 1987; Almasi, Gottlieb 1989). Under *full-service synchronization* an atomic operation is executed by a server process, or processor, to which the requesting process sends a request for the operation (Bitar 1990b, 1992). The Ultracomputer designers used the configuration of Figure 2.3a with a multistage interconnect, and at each switch in the network they placed a server processor (SP) having sufficient capability to implement fetch-and-add combining and decombining. The designers of the IBM RP3 took a similar strategy in an architecture like that of Figure 2.3c, but with SPs at the memory banks as in Figure 2.3a (Pfister et al. 1985).

Software combining tree. Yew et al. (1987), on the other hand, proposed implementing the atomic operations using full-service synchronization in Figures 2.3a and 2.3b with no combining in the network — just the SPs at memory execute atomic operations — and the tree itself is defined by software, yielding a software combining tree. They proposed this for busy-wait operations that create hotspots at memory, and hence bottlenecks in the network. An example of such an operation is polling a barrier count or a bit, waiting for it to become zero.

Using a software combining tree to implement general operations, such as fetch-and-add, rather than just busy-wait operations, would require greater complexity in the SPs in order to enable them to implement the combining and decombining operations illustrated in Figure 2.1b. In addition, the SPs would need to be able to communicate with each other, or else the CPUs would need to mediate inter-SP communication, notified by the SPs when appropriate.

Alternatively, notice that the pair of atomic operations of a software combining node (Figure 2.1b) may be implemented by the CPUs using *self-service synchronization.* In this case, the tree is defined by the node buffers, which are not associated with unique SPs as they would be under full-service synchronization. Under self-service, then, a CPU executes an atomic operation — combine, insert, or delete — for itself by locking the appropriate node buffer and executing the operation on the buffer contents.

Also observe that since the architecture of Figure 2.3c already has a CPU at each memory bank, it would be easy to implement a software combining tree in this architecture using the CPUs as the servers. This strategy would avoid the cost of hardware combining nodes and the cost of SPs, while retaining the flexibility of a software combining tree. Finally, we point out that the software combining strategy of Goodman et al. (1989) does not allow the pipeline concurrency that is available in our abstract model (Figure 2.1b), which is a generalization of the Ultracomputer hardware combining strategy.

Figure 2.2. Fetch-and-Add Queuing Algorithms.

Figure 2.2a. FIFO Enqueue (Insert) Using Fetch-and-Add.

```
Enqueue(input: adr_entry; output: success_flag ):

global variable:  Q_size, max, min, I, Q[Q_size], Next[Q_size];
local, register variable:  my_I, cell, ticket;

begin
    if  max < Q_size  then                              /* queue not full */
    begin
        if  fetch-and-add(max, 1) < Q_size  then        /* queue still not full */
        begin
            my_I := fetch-and-add(I , 1);

            /* now (in effect) divide my_I by Q_size */
            /* then take remainder to get cell, or truncate and multiply by 2 to get ticket */

            cell := remainder(my_I /Q_size );
            ticket := 2 * floor(my_I /Q_size );         /* 2 tickets/cell: 1 enqueue, 1 dequeue */

            while  Next [cell ] ≠ ticket  do null;      /* busy-wait for turn, could also delay retry */

            Q [cell ] := adr_entry ;                    /* write cell */

            fetch-and-add(Next [cell ], 1);
            fetch-and-add(min, 1);

            success_flag := 1;
            return;                                     /* return */
        end;
        else                                            /* queue full */
            fetch-and-add(max, -1);
    end;
    success_flag := 0;                                  /* queue full */
end;
```

Notes for Figure 2.2.

1. The queue size, Q_size, is a power of 2 that is less than or equal to the capacity of I and D.
2. The variable *ticket* is used to implement FIFO busy-wait queuing at the cells. However, I and D must be large enough so that by the time they wrap around — and thus begin repeating *ticket* values — the same values of *ticket* from the previous iteration of I and D have been used, that is, the respective enqueues and dequeues have been completed.

Figure 2.2b. Variable Values for Q_size = 4.

my_I:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	· · ·
cell:	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	· · ·
ticket:	0	0	0	0	2	2	2	2	4	4	4	4	6	6	6	6	· · ·

Figure 2.2 (continued). Fetch-and-Add Queuing Algorithms.

Figure 2.2c. FIFO Dequeue (Delete) Using Fetch-and-Add.

```
Dequeue(output: adr_entry, success_flag ):

global variable:  Q_size, max, min, D, Q[Q_size], Next[Q_size];
local, register variable:  my_D, cell, ticket;

begin
    if  min > 0  then                                   /* queue not empty */
    begin
        if  fetch-and-add(min, -1) > 0  then            /* queue still not empty */
        begin
            my_D := fetch-and-add(D, 1);

            /* now (in effect) divide my_D by Q_size */
            /* then take remainder to get cell, or truncate, multiply by 2, and add 1 to get ticket */

            cell := remainder(my_D/Q_size);
            ticket := 2 * floor(my_D/Q_size) + 1;       /* 2 tickets/cell: 1 enqueue, 1 dequeue */

            while  Next[cell] ≠ ticket  do null;        /* busy-wait for turn, could also delay retry */

            adr_entry := Q[cell];                       /* read cell */

            fetch-and-add(Next[cell], 1);
            fetch-and-add(max, -1);

            success_flag := 1;
            return;                                     /* return */
        end;
        else                                            /* queue empty */
            fetch-and-add(min, 1);
    end;
    success_flag := 0;                                  /* queue empty */
end;
```

Notes for Figure 2.2.

1. The queue size, *Q_size*, is a power of 2 that is less than or equal to the capacity of *I* and *D*.
2. The variable *ticket* is used to implement FIFO busy-wait queuing at the cells. However, *I* and *D* must be large enough so that by the time they wrap around — and thus begin repeating *ticket* values — the same values of *ticket* from the previous iteration of *I* and *D* have been used, that is, the respective enqueues and dequeues have been completed.

Figure 2.2d. Variable Values for *Q_size* = 4.

my_D:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	· · ·
cell:	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	· · ·
ticket:	1	1	1	1	3	3	3	3	5	5	5	5	7	7	7	7	· · ·

Generalized combining tree. We will characterize combining trees in a general manner that applies to both hardware and software trees. In this perspective, there is a combining tree for each combinable variable y, as in Figure 2.1b, and the m combining trees in the system share the system hardware, both the combining processors and their memories. Hence, a physical combining node must have sufficient processor power and memory space to accommodate all of the trees that it must process, which will be m or fewer trees.

2.2. Purpose

When are combining trees necessary?

Given their complexity, we would like to avoid combining trees if possible, so we ask when they are necessary. A combining tree is necessary when the respective computation is a *system bottleneck*, or when it creates a system bottleneck (as a memory hotspot creates in a network), and when the bottleneck can be relieved *only by the use of high-level parallelism*, that is, parallelism at the level of the atomic operation.

In order to clarify the concept of bottleneck and how to design a combining tree that will relieve a bottleneck, let us conceptualize the execution of a program as a *closed queuing network,* and let us conceptualize an atomic operation as a *service center.* This is illustrated in Figure 2.1d for n processors. Notice that since the system is closed, the request rate $\lambda(n)$ is a strictly increasing function of program execution rate, where rate is the reciprocal of mean inter-request time. In particular, if program execution rate slows down due to congestion in the network, $\lambda(n)$ will slow down accordingly, and this dynamic balancing activity will lead to a processing state of the system that reflects the balanced, or steady, state averaged over time. In analytic terms, this corresponds to a solution to the balance equations for the queuing system. Underlying this slow down, a processor can have at most some fixed number of concurrent contexts (processes loaded in processor registers), and a context can have at most some fixed number of outstanding requests at a time, since its lookahead depth will be limited by both the processor hardware and the program dependences. Throughout our analysis, the parameters we consider will be steady-state parameters. Unless otherwise qualified, we let $\lambda(n) = \lambda_{\min}(n)$, the request rate corresponding to the *slowest acceptable program execution rate,* since this will be the value that we are normally interested in.

Now it becomes evident that the atomic operation in Figure 2.1d will be a bottleneck if $\lambda(n)$ exceeds the operation service rate $\mu(n)$ — the maximum possible throughput — for some implementation of the atomic operation. If the computation cannot be restructured to reduce $\lambda(n)$ sufficiently, then $\mu(n)$ must be sped up to match $\lambda(n)$.

The service rate $\mu(n)$ may be sped up using a faster hardware technology, by using architectural techniques other than high-level parallelism, and by using high-level parallelism, i.e., combining. If the non-combining architectural techniques are exhausted and we still have $\lambda(n) > \mu(n)$, the designer is faced with the consideration of changing hardware technologies in the relevant system components, and this introduces several problems. First, a fast enough technology may not be available. Second, if it is available, it may be too costly. Third, if it is not too costly, it may warrant changing the technology of the entire system. Yet in this case the acceptable program execution rate will probably increase, thereby increasing $\lambda(n)$ and leaving the system still unbalanced with $\lambda(n) > \mu(n)$. If non-combining architectural techniques and faster technologies are exhausted, it remains to employ high-level parallelism in implementing the atomic operation, i.e., to employ combining, and this is possible in an asynchronous architecture if the atomic operation is associative and commutative.

Another perspective on $\lambda(n)$ is its behavior when the system is scaled up. If the request rate grows arbitrarily large as the system is scaled, i.e., $\lambda(n) \to \infty$ as $n \to \infty$, then the operation service rate must also grow arbitrarily large, i.e., $\mu(n) \to \infty$ as $n \to \infty$, but the only means of achieving this is to use high-level parallelism, that is, a combining tree.

Just the same, the idea that a program must be designed such that $\lambda(n) \to \infty$, for some object, must not be

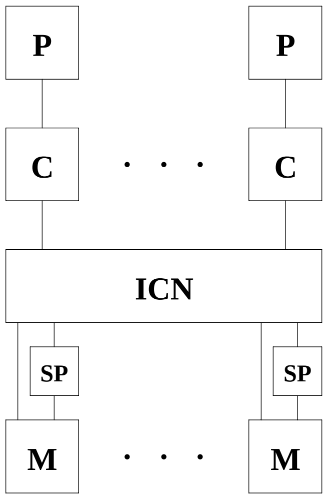


Figure 2.3a. Centralized Shared-Memory with Caches.

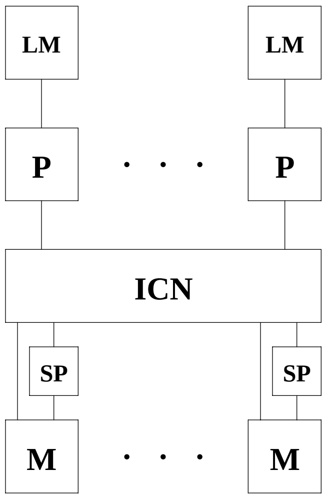


Figure 2.3b. Centralized Shared-Memory with Local Memories.

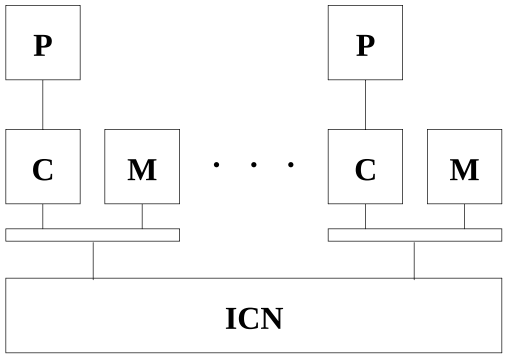


Figure 2.3c. Distributed Shared-Memory.

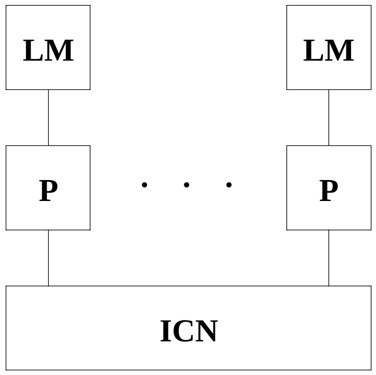


Figure 2.3d. Message Passing.

Figure 2.3. Major Architectural Configurations.

P = processor, C = cache, M = main memory, LM = local memory

SP = server processor (optional), ICN = interconnect network accepted without consideration of alternative algorithms. To illustrate, the original motivation for fetch-and-add was concurrent access to a FIFO queue containing processes to run, or work to execute. If there is only a single queue to serve the respective function for the entire system, then certainly $\lambda(n) \to \infty$ for that queue. More generally, for $\lambda(n) \to \infty$, we are talking about a relatively small number of queues compared to the number of processors. For number of queues $q(n)$, we are saying that $n/q(n) \to \infty$, that is, that the number of processors per queue is unbounded — assuming that the single-processor access rate does not increase in n. However, strict FIFO is normally not necessary, so a more reasonable design would be to have one queue for every l processors, where l yields an acceptable request rate and occupancy level per queue: $q(n) = \lceil n/l \rceil$, making $n/q(n) \leq l$. When a processor wishes to access a queue, it would randomly select a queue for insertion (or else it would randomly select a queue for deletion).

A more difficult bottleneck to eliminate is a barrier. Specifically, a common parallel execution paradigm entails dividing a computation into phases, where each phase is defined by a parallel construct such as **doall,** and the end of a phase is determined when all processes have decremented a barrier count, causing it to reach zero (Cheong, Veidenbaum 1990). During a single phase, the parallel execution threads have been arranged to maintain the integrity of writable objects, whereas execution threads occurring in different threads would interfere with each other.

2.3. Parameters

How do we design a combining tree?

The designer must address the parameters shown in Figure 2.4. Let us consider these parameters now, keeping in mind that several performance parameters are illustrated in Figure 2.1e, and that for simplicity we assume that all combining nodes are configured identically unless otherwise stated. The window parameters will be discussed last, in the next subsection.

Regarding the number of combining trees m in the system, a physical combining node must have sufficient processor power and memory space to accommodate all of the trees that it must process, which will be m or fewer trees. If the number of concurrent trees varies over time, then m is the maximum of these numbers.

Regarding processor request rate r, we assume that the program structure is independent of the number of processors n. If a program is rewritten for a larger value of n, then we have a new program for which the combining trees must be reparameterized. Consequently, the processor request rate $r = r_{\text{min}}$ — the request rate generated by the slowest acceptable program execution rate — is independent of the number of processors n; in particular, r does not increase in n. Notice that we could make the weaker assumption that $r \equiv r(n)$ is bounded; however, we prefer the simpler assumption that r is independent of n. The significance in the difference between λ and r is that λ may be unbounded above whereas r is bounded, hence we can design a combining node to handle k processor request streams, each having rate r. The maximum possible request rate r_{max} for a processor is limited by the processor architecture and the program content. Finally, if the processor request rates are not the same for all processors, we let the request rates refer to the maximum over the processors.

Regarding the maximum number of outstanding requests per processor $n_{\text{max.reqs.out}}$, as mentioned, a processor can have at most some fixed number of concurrent contexts (processes loaded in processor registers), and a context can have at most some fixed number of outstanding requests at a time, since its lookahead depth will be limited by both the processor hardware and the program dependences. The product of the two limits gives us the maximum number of outstanding requests per processor.

Regarding node combining rate μ_{comb}, we divide the sojourn time of a combine request at a node into the following three successive intervals.

Figure 2.4. Combining Tree Parameters.

Note: This is a partial list of the parameters that define a combining tree configuration.

Figure 2.4a. Processor Parameters, Other Rates, Fan-in.

- Processor parameters
  - □ Closed system limits
    - n: # processors
    - # contexts/processor: # processes (execution threads) that can be concurrently loaded into the processor.
    - max # outstanding requests/context: Both processor lookahead depth and program dependences limit this value. Once this limit is exhausted, the processor must stall until receiving a reply.
    - $n_{\text{max.reqs.out}}$: Max # outstanding requests/processor — product of the two prior variables.
  - □ Request rates
    - r: Individual processor request rate — the reciprocal of mean inter-request time for requests to a combining tree.
    - r_{max}: The value of r that would result if the delay in the network were zero.
    - r_{min}: The value of r for the slowest acceptable program execution rate.
    - λ: Total processor request rate. $\lambda \equiv \lambda(n) = nr$. $\lambda_{\text{min}} = \lambda(r_{\text{min}})$. $\lambda_{\text{max}} = \lambda(r_{\text{max}})$.
    - Unless other qualified, we let $r = r_{\text{min}}$ since r_{min} is normally relevant to the discussion.
- Other rates
  - □ μ_{comb}: Node combining rate — maximum rate at which incoming requests can be combined.
  - □ μ_{decomb}: Node decombining rate — maximum rate at which decombined requests can be produced. We assume $\mu_{\text{decomb}} = \mu_{\text{comb}}$.
  - □ μ_{net}: Network request capacity — maximum rate at which the network can service requests from a combining node.
  - □ μ_{mem}: Memory service rate — maximum rate at which memory can service requests.
- k: Fan-in — combining node fan-in, decombining node fan-out.

Figure 2.4 (continued). Combining Tree Parameters.

Figure 2.4b. Window Parameters, Buffer Sizes, # Combining Trees.

- Window parameters
  - □ τ: Mean combining window size — mean time for which a combining node combines incoming requests into a single outgoing request. τ_{root} is the mean window size for the root node, which feeds memory.
  - □ τ_{indep}: The value of τ as an independent variable. This is relevant if window size is an independent variable (as opposed to batch size).
  - □ τ_{dep}: The value of τ as a dependent variable. This is the mean inter-request time for outgoing requests of a combining node.
  - □ β: Mean batch size — mean number of requests that are combined into a single outgoing request.
  - □ Notes
    - If batch size (rather than window size) is an independent variable, then τ_{indep} is undefined and $\tau = \tau_{\text{dep}}$. On the other hand, if window size (rather than batch size) is an independent variable, then $\tau_{\text{dep}} \geq \tau_{\text{indep}}$, with equality holding iff empty batches do not occur, and $\tau_{\text{dep}} \equiv \tau_{\text{dep}}(\tau_{\text{indep}})$ is an increasing function of τ_{indep}, where ‘‘increasing’’ does not imply strictly increasing.
    - τ_{dep} is a function of a node’s window discipline and its arrival process, hence in general it will vary with distance from the processors.
    - Unless otherwise stated, we let $\tau = \tau_{\text{dep}}$ since τ_{dep} is normally relevant to the discussion.

- Buffer sizes
  - □ Buffer entry size
  - □ Combining buffer size (the buffer is associative by destination address)
  - □ Decombining buffer size (the buffer is associative by request i.d.)

- Number of combining trees
  - □ m: # combining trees in the system.
  - □ Note: All of the foregoing parameters are determined per tree.
  - □ $r_{\max *}$: Maximum of the r taken over the m trees.

- *Entry time:* Time from the request's arrival at the node's hardware queue until the node enters the request in its combining buffer, combining it with a request already present (if any).
- *Window time:* Time for which the request is held in the node's combining buffer in order to allow combining with subsequent arrivals.
- *Exit time:* Time from the end of the window time until access to the network is obtained for routing the request to the parent node.

In addition, there are two alternatives for when to move a request from the combining buffer into the decombining buffer, thereby disallowing further combining of it.

- *Pre-exit move:* Move the request into the decombining buffer at the beginning of the exit time. This would be normal for a *software combining node* since the user will manage the buffers while the system will manage access to the network.
- *Post-exit move:* Move the request into the decombining buffer at the end of the exit time. This could be done in a *hardware combining node* since the hardware will manage both the buffers and the network access.

We see, then, that the combining rate μ_{comb} is a node's processing capacity corresponding to the entry time — the rate at which a node can enter incoming requests into its combining buffer. The network request capacity μ_{net} corresponds to the exit time — the rate at which a node can gain access to the network to place outgoing requests into the network. From Figure 2.1e we see that the node combining rate μ_{comb} must accommodate the incoming request rate, and the network must allow an exit rate μ_{net} that matches the node request rate: we must have $\mu_{\mathrm{comb}} \geq kr$, $\mu_{\mathrm{comb}} \geq k/\tau$, and $\mu_{\mathrm{net}} \geq (k+1)/\tau$. Clearly the combining rate depends on the node design, so let us look more closely at node design now.

The highest-level decision in the node design is the choice between self-service synchronization (locking) and full-service synchronization. *Under self service,* μ_{comb} will normally be affected by the network traffic, since a processor will normally not have a private connection to the memory bank containing a node's buffers. *Under full service,* the primary decision is the choice between hardware and software combining nodes. Hardware combining nodes have the advantage of speed, but they have serious inflexibility disadvantages: inflexibility in the atomic operations they can perform, in the buffer resources they can use, and in the fan-in k. In contrast, implementation of a software combining tree in Figure 2.3c or 2.3d, using full service, gives a combining node the power of a CPU, the flexibility of software, and the buffer capacity of a memory bank.

Regarding fan-in k, the fan-in k must be chosen so that the combined request rate of k processors, or k combining nodes, does not exceed the combining rate μ_{comb}: as stated above, we must have $k/\tau \leq \mu_{\mathrm{comb}}$. Hence, combining rate μ_{comb} and fan-in k must be considered together in the design of the tree.

For a software combining tree, another consideration with respect to fan-in is *the tradeoff between tree depth and system balance:* greater fan-in k reduces tree depth, $(\log_k n) - 1$, but also reduces system balance with respect to that tree since it concentrates the work on fewer physical nodes, namely, $(n-1)/(k-1) \approx n/(k-1)$ nodes. (The expressions assume that n is a power of k.) The advantage of smaller tree depth is that, for a given processor utilization, smaller depth allows a smaller limit on the number of outstanding requests that a processor can issue before having to wait for a reply. In fact, if the depth is sufficiently large in relation to the limit on the number of outstanding requests, then a processor will not be able to maintain its minimum acceptable request rate r because it will be forced to wait for replies. We will characterize this precisely in Section 2.5. In a software combining tree, then, we can accommodate smaller tree depth by dispersing different trees over different physical nodes and by maintaining system balance across trees rather than within trees. For simplicity, in the ensuing analysis we will assume that fan-in k is the same for all m trees.

Combining of requests requires their temporal proximity.
Does the proximity need to be ensured?

Combining window. The answer is *yes.* In order to ensure the parallel execution, and hence speed up, that the combining tree is designed to give, it is necessary to observe a *combining window,* an interval of time in a combining node during which incoming requests are gathered in the node in order to combine them into a single outgoing request.

At first glance, one might rebut, ''But this will slow the computation down.'' In fact, it will not slow the computation down if the window has an appropriate size, but to fail to observe a combining window may slow the computation down. For without a combining window, sufficient combining may not occur at the wider levels of the tree (closer to the leaves), so the processing demands may be concentrated on the smaller number of nodes closer to the root. The result is that the request arrival rate at these nodes may exceed the node service rate, so the combining tree may not be able to obtain the speed up that it is designed to achieve. In addition, the node buffers may overflow at the nodes closer to the root.

To see this, suppose that a combining window is not observed — the window time is zero. Then if two requests at a leaf do not arrive at the same time and if no combining occurs during the exit time, then the requests will not be combined, so the leaf will become invisible for those requests: the requests will pass right through to the parent node, subject to a transit delay through the leaf node.

To make the problem more explicit, suppose that the combining nodes are designed to handle a request rate of kr, i.e., $\mu_{\text{comb}} = kr$ (Figure 2.1e). Now let us focus on a group of k sibling processors, and let us assume that during a time interval I of size τ, each processor sends a request to the group's leaf, *but the requests are mutually staggered,* each arriving in a distinct subinterval of size τ/k. Assume that during the interval I, this also occurs for the other processor sibling groups in the system. Under this scenario, if a combining window is not observed and if no combining occurs during the exit time, then the leaf nodes become invisible (except for the transit delay), and a second-level node will receive requests at the rate of $k^2/\tau = k^2 r$ instead of at the rate of $k/\tau = kr = \mu_{\text{comb}}$. This is a worst-case scenario, but it clarifies that if the combining window is not observed, the second-level nodes may receive requests at a rate that exceeds the rate μ_{comb} on which the design is based.

Continuing our scenario, due to the excessive request rate at the second-level nodes, more combining will occur there, and the third-level nodes will receive a reduced request rate, closer to the design rate kr. But then the third-level nodes will play a role like the leaf nodes, so the fourth-level nodes will receive an excessive rate. This alternation phenomenon will continue through the tree, and the tree will not be processing requests at the needed rate of $\lambda(n) = nr$, due to the loss of parallel execution. For example, the processing that should have occurred at the leaves, but occurs at the second level instead, will proceed at a rate k times slower than at the leaves.

Time/space duality. The problem of speed up that is solved by sufficient *parallel execution* has a dual problem: the problem of node buffer space, which is solved by sufficient *parallel storage.* That is, if the arrival rate at a node exceeds its service rate, not only will the processing rate be too slow, but the node must also store all of those requests, so its buffer space may be exceeded. Thus, the combining window allows us to bound node buffer size.

What is an appropriate window size?

Window size. A combining window creates a type of batch request, in this case a request that represents a batch that will be stored in a decombining buffer if its size is greater than one. But if window size (rather than batch size) is manipulated as an independent variable, when a window closes, the batch may be empty due to the random inter-arrival times of requests. Since an empty batch will not generate a request for the parent node, we wish to adjust the mean window size τ_{indep} so that, given that an empty window will be subsumed by the next window, the resulting request rate is a desired value, such as r, i.e., so that $\tau_{\text{dep}} = 1/r$. This implies that the mean independent window size τ_{indep} will be smaller than the mean dependent window size τ_{dep}, unless no empty batches occur.

Observe that τ_{dep} is the mean inter-request time for a node's outgoing requests. It will depend not only on a node's window discipline, but also on the arrival process to the node. Hence, τ_{dep} will in general vary with a node's distance from the processors. For a complete tree, a node's distance from the processors is the same for all relevant processors, so in this case we may write $\tau_{\text{dep}} \equiv \tau_{\text{dep},\,i}$, where $i = 1,\ldots,\log_k n$. For convenience, we also let $\tau_{\text{dep},\,0}$ be the processor mean inter-request time. Since we will normally be interested in the dependent window size, for convenience we let $\tau = \tau_{\text{dep}}$ and $\tau_i = \tau_{\text{dep},\,i}$ unless otherwise qualified. Unsubscripted τ refers to the node being discussed, or to all nodes if all nodes are assumed to have the same τ.

Now window size, even if manipulated as an independent variable, need not be constant. In general it is a random variable having mean τ, where constant size is a special case. Window size and batch size may be related by a time/space duality of means, for mean batch size at distance i from the processors is proportional to the ratio τ_i / τ_{i-1}: $\beta_i = (k/\tau_{i-1})\,\tau_i = k\,\tau_i/\tau_{i-1}$. This gives us *the fundamental time/space duality relation:*

$$\beta \equiv \beta_i = \beta(\tau_{i-1}, \tau_i) = k\,\tau_i / \tau_{i-1} \tag{2.1}$$

Rephrasing what we stated earlier, we can see from Figures 2.1e and 2.4 that we must have $\tau \geq k/\mu_{\text{comb}}$ for a node's outgoing request rate to avoid outpacing the parent node and possibly overflowing its decombining buffer. In addition, for the root node we need $\tau_{\text{root}} \geq 1/\mu_{\text{mem}}$ for the node to avoid outpacing its memory unit. (For simplicity, we assume that the memory unit is fed solely by the respective root node.) Finally, we need $\tau \leq 1/r$ to maintain a per-processor throughput of at least r through the tree. The two one-sided bounds together yield *the fundamental combining-window bounds*:

$$k/\mu_{\text{comb}} \leq \tau \leq 1/r \tag{2.2}$$

which, in terms of throughput, are

$$k \bullet \text{node output rate} \leq \mu_{\text{comb}}, \quad \text{node output rate} \geq r$$

where node output rate $= 1/\tau$. Notice, however, that we do not need a throughput greater than r since r yields an acceptable program execution rate. Consequently, *we will assume that $\tau = 1/r$ since this rate will avoid wasting network bandwidth and processor cycles.* Just the same, in Chapter 5 we will explore the implications of reducing τ closer to k/μ_{comb} when $k/\mu_{\text{comb}} < 1/r$.

Window discipline. For $\tau = 1/r$, the simplest window discipline is to close a window every k requests, that is, to define windows by constant batch size. However, this introduces the problem of window-size variance, since it will take a variable amount of time for k successive requests to arrive at a node. From a high-level point of view, *constant batch size* will tend to propagate transient lulls and bursts in processor request rate through the tree, thereby increasing the queue-length variance for each queue in the network. Since queue length has a lower bound of zero, if queue length reaches zero at times during steady state, increasing queue-length variance will increase queue-length mean. From a different perspective, increasing server idle time, for a given request rate and service rate, will increase queue length. The result of longer queues, in turn, will not only be longer transit times through the network, but also larger decombining buffer utilizations, as we will see below. In contrast to the discipline of constant batch size, the discipline of *constant window size* will tend to smooth out lulls and bursts, thereby decreasing the queue-length variances and means, and hence reducing the transit times and decombining buffer utilizations.

In short, we have a tradeoff between batch-size variance — a space variance — and window-size variance

— a time variance: constant batch size gives us variable window size, while constant window size gives us variable batch size. Window-size variance, in turn, affects the probability distributions of queue length, transit time, and decombining buffer utilization.

Observe that the problem of lulls under constant batch size can be compensated if the computation is defined by phases, such as barriers, and the lulls occur in transition from one phase to another. In this case, a processor can send an end-of-window request when arriving at the end of a phase. When a node receives such a request, it will close its window and also send an end-of-window request to its parent. More generally, in a group of k sibling processors, when a processor arrives at the end of a phase, it could decide whether to send an end-of-window request based on its arrival order among the k. The decision algorithm would attempt to reduce the number of such messages while maintaining the flow rate through the tree.

2.4. Bounding Node Buffer Size

How do we find an upper bound on node buffer size?

Using asymptotic notation, let us determine an upper bound on mean buffer size in terms of parameters n, m, k, r, and d_{root}, which is the distance of a node in question from the root. This is a much simpler task than obtaining the probability distribution, which we address in Chapter 4. We will consider buffer entry size, combining buffer size, and decombining buffer size. Regarding asymptotic notation, keep in mind that $O(\cdot)$ is an asymptotic upper bound, $\Omega(\cdot)$ is an asymptotic lower bound, and $\Theta(\cdot)$ is the conjunction of the two one-sided bounds.

Buffer entry size. Let the buffer cell size accommodate c requests, $c \geq k$, and let us speak of cell size as c, ignoring a constant overhead component per cell. A batch entry will comprise some number of cells. Now if batch size is constant, cell size will be $c = k$, and an entry will comprise one cell. However, if batch size is not constant, we may want $c > k$, and more than c requests may arrive during a window. The overflow may be handled either by extending the *entry* — increasing the batch-size variance — or by extending the *window* — increasing the window-size variance. Under the first strategy, of extending the entry, at the arrival of a request $ic+1$ within a window ($i = 1,2,...$), the entry will be extended, say by linking or rehashing. Under the second strategy, of extending the window, at the arrival of request $c+1$ within a window, the current entry (containing c requests) will be completed and a new batch will be started, but the window of the new batch will be lengthened by the remaining, unused portion of the prior window. The first of the two strategies, which increases batch-size rather than window-size variance, will have greater effect in smoothing out bursts, and thus in reducing transit-time variance, but at the cost of the complexity of extending an entry. We express mean entry size in terms of k as $\Theta(k)$.

Combining buffer size. The combining buffer for a tree node needs only one entry, hence its mean size is $\Theta(k)$. However, the combining buffer for a physical combining node must, in general, handle multiple trees — m or fewer trees — giving a bound of $O(mk)$ on its mean size.

Decombining buffer size. We need to bound decombining buffer size in terms of the node's level in the tree, since the level will determine the bound, as follows.

- *Absolute bound:* The distance of a node *from the processors* determines the absolute upper bound on decombining buffer size.
- *Probabilistic bound:* The distance of a node *from the root* determines an upper-bound probability distribution for buffer size, and hence an upper bound on mean buffer size.

Now let us see why this is true. As stated earlier, let d_{root} be the distance of a node in question from the root, $d_{\text{root}} = 0,...,\lceil \log_k n \rceil - 1$, and let $d_{\text{processor}} = \lceil \log_k n \rceil - d_{\text{root}}$, which is the distance of the node from the processors. Let us consider the absolute bound, followed by the probabilistic bound.

Absolute bound. The absolute upper bound on decombining buffer size of a node may be determined from

$d_{\text{processor}}$. Specifically, as mentioned earlier, a processor can have at most some fixed number of concurrent contexts, and a context can have at most some fixed number of outstanding requests at a time (Figure 2.4a). Thus, the absolute upper bound for distance $d_{\text{processor}}$ is the maximum number of requests per node $n_{\text{max.requests/node}}$.

$$n_{\text{max.requests/node}} = (n_{\text{max.reqs.out}} \text{ requests/processor})\,(k^{d_{\text{processor}}} \text{ processors/node}) \qquad (2.3)$$

The number of requests is then multiplied by the maximum number of bits per request, but we will ignore this conversion.

Now if batch size were constant, ensuring k-fold combining, then setting $d_{\text{processor}} = 1$ in (2.3) would give us an absolute bound for all nodes. However, under non-constant batch size, k-fold combining for each window closure will not be guaranteed. Consequently, we must determine a probabilistic bound. In fact, even if $d_{\text{processor}} = 1$, if (2.3) is large, buffer size based on (2.3) may not be feasible, thus requiring a probabilistic bound anyway. Here we determine a bound on the mean, while in Chapter 4 we address the probability distribution of buffer size.

Probabilistic/mean bound. To obtain an upper bound, we will assume that $n_{\text{max.reqs.out}} = \infty$; that is, we will assume that we have an open system, so that processor request rate never slows down because a processor must wait for a reply. More generally, this means that $n_{\text{max.reqs.out}}$ is sufficiently large that the number of outstanding requests for a processor is never limited by $n_{\text{max.reqs.out}}$.

We begin by observing that after a request leaves a combining node for the parent node, the mean round-trip transit time to memory, via the root, is the sum of the mean transit times at each of the intervening levels:

$$\sum_{i=1}^{d_{\text{root}}} t_{ni} + t_{\text{mem}}$$

where t_{ni} is the mean round-trip transit time from (exclusive) level i to (inclusive) level $i-1$ and where t_{mem} is the mean round-trip transit time from the root to memory. The term t_{mem} is independent of n assuming that processor requests are uniformly distributed across the memory banks, except for those handled by combining trees, and that the number of memory banks is proportional to n. The term t_{ni} is, in turn, the sum of the mean sojourn times at the links on the round-trip pathway from (exclusive) level i to (inclusive) level $i-1$:

$$t_{ni} = \sum_{j=1}^{l_{ni}} t_{nij}$$

where l_{ni} is the number of links on the round-trip pathway from (exclusive) level i to (inclusive) level $i-1$ and t_{nij} is the mean sojourn time at link (i, j), $i = 1,..., d_{\text{root}}$, $j = 1,..., l_{ni}$. (This assumes that the l_{ni} are constants, but if they were random variables, we could obtain a suitable result with appropriate assumptions.)

This gives us the following expression for mean round-trip time from level d_{root}:

$$\sum_{i=1}^{d_{\text{root}}} \sum_{j=1}^{l_{ni}} t_{nij} + t_{\text{mem}} = \Theta\Big(\sum_{i=1}^{d_{\text{root}}} \sum_{j=1}^{l_{ni}} t_{nij} \Big)$$

To replace the sums by a product, let $l_n = \max_i(l_{ni})$ and $t_n = \max_{i,j}(t_{nij})$, which yields asymptotic bound on mean round-trip time

$$O(t_n\, l_n\, d_{\text{root}})$$

Now once a request has been sent, the mean number of additional requests that will be sent from the node until the reply for the original request returns is just the mean round-trip transit time divided by $\tau = 1/r$. Once a reply arrives and the original request is decombined, the request will be deleted from the decombining buffer. For if reliable delivery of decombined requests may be a problem, we assume that it will be handled by the network, which will handle each resulting component request individually.

Since mean batch size is k for $\tau = 1/r$, this yields the following results for decombining buffer size, keeping in mind that $d_{\text{root}} \le \lceil \log_k n \rceil - 1$, and $r_{\max *}$ is the maximum request rate r over all m trees.

- Mean decombining buffer size
  - □ For a tree node for one combinable variable:

$$O(k\, r\, t_n l_n d_{\text{root}}) \quad = \quad O(t_n l_n \log_k n) \tag{2.4}$$

  - □ For a physical combining node:

$$O(mk\, r_{\max *}\, t_n l_n d_{\text{root}}) \quad = \quad O(t_n l_n \log_k n) \tag{2.5}$$

- Absolute decombining buffer size, from (2.3):

$$\Theta(n/k^{d_{\text{root}}}) \quad = \quad O(n) \tag{2.6}$$

Notice that in deriving (2.4) and (2.5), we assumed that singleton batches are stored in the decombining buffer. We will see below what the correction for eliminating them is and that this correction does not invalidate (2.4) and (2.5).

Now observe that a combine request that represents a batch of size one will not be stored in the decombining buffer, hence let $\beta^* \equiv \beta_i^* = \beta^*(\tau_{i-1}, \tau_i)$ be the mean size of batches in the decombining buffer of a node at distance i from the processors. To see the relationship between β and β^*, let $B \equiv B_i = B(\tau_{i-1}, \tau_i)$ have the distribution of the size of non-empty batches, and let $B^* \equiv B_i^* = B^*(\tau_{i-1}, \tau_i)$ have the distribution of the size of batches in the decombining buffer. Then if $P\{ B > 1 \} > 0$, $B^* \sim (B \mid B > 1)$; otherwise $B^* \equiv 0$. We will assume that $P\{ B > 1 \} > 0$, since this is the interesting case. Since $E(B) = E(B \mid B > 1)\, P\{ B > 1 \} + P\{ B = 1 \}$, we can obtain an expression for β^* in terms of β:

$$\beta^* \quad = \quad \frac{\beta - P\{ B = 1 \}}{P\{ B > 1 \}} \quad = \quad \Theta(\beta) \tag{2.7}$$

Also observe that the arrival rate to a node is equal to the arrival rate to the node's decombining buffer in terms of combined requests, plus the departure rate from the node of uncombined requests (singleton batches). More simply, the arrival rate to a node is equal to the departure rate of requests in terms of the requests in the respective batches. To illustrate, we express the latter for distance i from the processors in a complete tree — a rearrangement of (2.1):

$$k / \tau_{i-1} \quad = \quad \beta_i / \tau_i \tag{2.8}$$

Since $k = \beta(1/r, 1/r)$, if we let $k^* = \beta^*(1/r, 1/r)$ and $K = B(1/r, 1/r)$, we have (2.7) for the case which we are considering of $\tau_i \equiv 1/r$:

$$k^* \quad = \quad \frac{k - P\{ K = 1 \}}{P\{ K > 1 \}} \quad = \quad \Theta(k) \tag{2.9}$$

If we let r^* be the rate of batches entering the decombining buffer for $\tau = 1/r$, then $r^* = P\{ K > 1 \}\, r$, yielding

$$k^* r^* \quad = \quad (k - P\{ K = 1 \})\, r \quad = \quad \Theta(kr) \tag{2.10}$$

Thus, to exclude singleton batches from (2.4) and (2.5), we would replace kr by $k^* r^*$, but due to (2.10), (2.4) and (2.5) are valid as stated.

Let us examine more closely the derivation of the term kr in (2.4) by defining and relating the underlying random variables. Let $R_{d_{\text{root}}}$ be the round-trip transit time from a node at level d_{root} to memory, let N be the number of outgoing requests from the node that occur during such a round trip (before the trip is done), let T_i be the inter-request time preceding outgoing request i, $i = 1,\ldots, N+1$, and let $S_j = \sum_{i=1}^{j} T_i$. Then for any execution ω, and for any respective round trip in that execution, we have

$$S_N(\omega) \quad < \quad R_{d_{\text{root}}}(\omega), \qquad S_{N+1}(\omega) \quad \geq \quad R_{d_{\text{root}}}(\omega)$$

Since we are assuming an open system, the T_i are mutually independent, and if we assume a complete tree, then the T_i are identically distributed since the distance from the processors determines their distribution. In addition, N is not a stopping time (Wolff 1989) for the T_i because until the round trip is finished, request N cannot be identified as the last request that will occur during the round trip. However, $N+1$ is a stopping time for the T_i, and since the T_i are i.i.d., we can use Wald's Equation (Wolff 1989) to obtain

$$E(R_{d_{\text{root}}}) \quad \leq \quad E(S_{N+1}) \quad = \quad E(N+1)\,\tau_{\text{dep}}$$

noting that $E(T_i) \equiv \tau_{\text{dep}}$. Since $S_N = S_{N+1} - T_{N+1}$, we have

$$E(R_{d_{\text{root}}}) \quad > \quad E(S_N) \quad = \quad E(N+1)\,\tau_{\text{dep}} \;-\; E(T_{N+1}) \quad \approx \quad E(N)\,\tau_{\text{dep}}$$

Hence, we can conclude that

$$E(N) \quad = \quad \Theta(\,E(R_{d_{\text{root}}})/\tau_{\text{dep}}\,)$$

Now the number of requests that enter the decombining buffer during the round trip may be defined as follows:

$$D_N \quad = \quad \sum_{i=1}^{N} B_i$$

which includes singleton batches, or

$$D_N^* \quad = \quad \sum_{i=1}^{N} B_i\,I_i$$

which excludes singleton batches, where indicator $I \equiv I_i = 1$ if and only if $B_i > 1$. N is independent of the batch sizes, since N is determined by the T_i and $R_{d_{\text{root}}}$ independent of the batch sizes, giving us

$$E(D_N) \quad = \quad E(N)\,E(B) \quad = \quad \Theta(\,E(R_{d_{\text{root}}})\,\beta/\tau_{\text{dep}}\,) \tag{2.11}$$

$$E(D_N^*) \quad = \quad E(N)\,E(BI) \quad = \quad \Theta(\,E(R_{d_{\text{root}}})\,\beta^*\,P\{\,B > 1\,\}/\tau_{\text{dep}}\,) \tag{2.12}$$

If we let $\tau_{\text{dep},\,i} \equiv \tau_{\text{dep}} = 1/r$, and $E(R_{d_{\text{root}}}) = O(\,t_n\,l_n\,d_{\text{root}}\,)$, then (2.11) gives us (2.4), and (2.12) gives us (2.4) with k^*r^* replacing kr in (2.4).

In conclusion, the derivation of (2.11) and (2.12) clarified that we converted the mean of one sum that has a random limit into a product of means using Wald's Equation, and that we converted the mean of another sum that has a random limit into the product of means using independence of the sum limit from the sum terms. These manipulations underlie the derivation of (2.4).

Scalability. Observe that the bound on *mean* size *increases* in d_{root} since it increases in the round-trip transit time to memory, while the bound on *absolute* size *decreases* in d_{root} since it increases in distance from the processors. Thus, for a node at a fixed distance $d_{\text{processor}}$ from the processors, as $n \to \infty$, the bound on mean size goes to infinity, while the bound on absolute size remains constant.

On the other hand, for a node at a fixed distance d_{root} from the root, as $n \to \infty$, the bound on absolute size goes to infinity, while the bound on mean size depends on the round-trip transit time to memory, which is $O(\,t_n\,l_n\,d_{\text{root}}\,)$. If t_n and l_n are independent of n, then mean buffer size for the node is $O(\,d_{\text{root}}\,)$, independent of n, and the system becomes scalable. That is, if the system is scaled up by adding nodes at the leaves, leaving all prior nodes in place, *the buffers for the prior nodes may remain unchanged.* The buffers of the new leaves, however, must be sufficiently larger than the buffers of the prior leaves, in order to accommodate the longer round trip to memory. Notice that the absolute bound is irrelevant to scaling because when a new level is added to the tree, the absolute bound for all prior nodes increases by a factor of k.

Multistage interconnect. Let us illustrate the bound on mean size using a multistage interconnect having

$k \times k$ switches, in which case the number of links between child and parent is independent of level in the tree, i.e., $l_{ni} \equiv l_n$. We have assumed that the number of memories is proportional to n, that the individual processor request rate to the memory system is independent of n, and that processor requests are uniformly distributed across the memory banks, except for those handled by combining trees. This makes the link (mean) sojourn time $t_{nij} \equiv t_n \equiv t$ independent of n, giving us the following bounds from (2.5).

- For a software combining tree: $l_n = \Theta(\log_k n)$, yielding mean size

$$O((\log_k n)\, d_{\text{root}}) \quad = \quad O(\log^2 n) \tag{2.13}$$

- For a hardware combining tree: $l_n = 1$, yielding mean size

$$O(d_{\text{root}}) \quad = \quad O(\log n) \tag{2.14}$$

Eliminating the decombining buffer. In a system with caches, if coherence is maintained for a block by linking the cached copies into a list, as in the IEEE Scalable Coherent Interface (James et al. 1990), it is possible to eliminate the decombining buffer of a physical combining node by distributing the decombining information to the respective cache nodes. However, for software combining nodes, the distribution of information serves no useful purpose since the combining trees should be distributed across the processor nodes in a balanced manner. For hardware combining nodes, one would expect the distribution strategy to incur a substantial performance cost due to the extra traffic and due to the decombining latency, thereby possibly eliminating the speed advantage of hardware combining trees over software combining trees.

2.5. Determining the Parameter Values

How are the parameter values determined and implemented?

Software combining node. A software combining node is under software control, so processor request rate can easily be evaluated during the computation, and window size may be incrementally modified, based on an initial estimate. These evaluations could be made for each combining node independently, hence τ_{indep} could differ with level in the tree to obtain equal τ_{dep} among the tree levels. However, this is unnecessary. All that is necessary is to select τ_{indep} so that τ_{dep} at all levels satisfies the fundamental window bounds in (2.2). With regard to implementing a time-based window discipline, a processor can set a timer trap, and if needed, it can keep additional timing information in a data structure that it consults when a timer trap goes off.

Fan-in k and tree depth may be tuned over multiple program executions. More specifically, let us consider the sequence of requests from a processor to a combining tree. For a given request i, let us consider the subsequence that starts with i and that continues to the next request (if any) whose issuance will cause the processor to stall if a reply to i has not yet been received. Let us call the latter request the *stall point* for i. Let random variable $L_{\text{stall},\,i}$ be the length of this sequence if the stall point for i exists; otherwise let $L_{\text{stall},\,i}$ be ∞. Let random variable $T_{\text{stall},\,i}$ be the sum of the inter-request times contained in this sequence if the stall point for i exists; otherwise let $T_{\text{stall},\,i}$ be ∞. And let random variable $T_{\text{trip},\,i}$ be the round trip time for i. Then a processor will never stall in order to wait for a reply from the combining tree if and only if for all executions ω and for all requests i in the execution

$$T_{\text{trip},\,i}(\omega) \quad \leq \quad T_{\text{stall},\,i}(\omega) \tag{2.15}$$

that is, if and only if the reply for every request i will be received by the time that the stall point for i is reached. Keep in mind that in an asynchronous execution, the stall point for a request i may not be determined until after i has been issued.

Tree depth matters, then, because $T_{\text{trip},\,i}$ is an increasing function of tree depth. Programs having small values of $T_{\text{stall},\,i}$ must have correspondingly small values of $T_{\text{trip},\,i}$ if the processors are not to stall waiting for replies, which may reduce their request rate below r_{min}. Observe that the model presented in Figure 2.4

oversimplifies by assuming that for all requests i, $L_{\text{stall},\,i}$ is a constant, viz., $n_{\text{max.reqs.out}}$.

In addition, it may be helpful to notice that the problem of avoiding stalling may roughly be characterized in terms of filling a pipeline. Specifically, in a certain sense of the word, a combining tree may be thought of as creating a pipeline for each request i, where the length of the pipeline is the round-trip time of the request $T_{\text{trip},\,i}$. In order to avoid stalling, then, a processor must fill the pipeline with inter-request times that fall between the issuing of request i and the arrival at the stall point for request i. If the processor can fill this pipeline, then the reply for request i will emerge from the pipeline by the time that the stall point has been reached.

Finally, notice that under lock-synchronized nodes, each node is identified by its pair of buffers, and a processor must be designated as a supervisor for each node, in order to implement the combining window for the node. In order to achieve this, for each tree there would be a mapping from the set of memory banks to the set of processors, and each tree node would be assigned to the processor corresponding the node's memory bank.

Hardware combining node. For a hardware combining node, the situation is much more difficult due to the cost and the inflexibility of hardware implementation. It is necessary to design for an acceptable number of combining trees m, and then estimate the combining buffer size, which is $\Theta(mk)$, and the decombining buffer size having bound on its mean $O(mk\ r_{\max *}\ t_n l_n d_{\text{root}})$. Hence, the parameters must be determined by modeling and simulation.

In a hardware combining node, the exit time provides an *implicit combining window* having mean length, say, τ_{exit}. But this length is independent of the processor request rate r to a particular combining tree; consequently, only by coincidence would τ_{exit} be an appropriate size for a given combining tree. For example, assuming that $\tau_{\text{exit}} < k/\mu_{\text{comb}}$, as we would expect, if $r \geq 1/\tau_{\text{exit}}$, then insufficient combining will occur: the parent node will not be able to accommodate the resulting request rate and possibly the resulting decombining buffer utilization. Consequently, aside from special cases, we would expect it to be necessary for explicit combining windows to be implemented in a hardware combining node.

How may an explicit combining window be implemented in a hardware combining node? Constant batch size is the most attractive discipline from the point of view of hardware simplicity. However, it will tend to propagate lulls and bursts in the processor request streams and thereby increase network queue lengths, as discussed earlier. Thus, it may be desirable to use a time-based window discipline. A time-based window may be implemented by augmenting a combining buffer entry with a counter that maintains the age of the respective entry, decrementing the value from τ_{indep} to zero. Since τ_{indep} will, in general, differ from tree to tree, each request should carry a value of τ_{indep} with it, where τ_{indep} has been selected so that τ_{dep} at all levels will satisfy the fundamental bounds in (2.2). A node would then send a request to a parent when the combining buffer of the node has an entry that is old enough — an entry with a counter equal to zero. If a hardware combining buffer entry has a fixed size, then a time-based window must be accompanied by a method for dealing with entry overflow — either extending the entry or extending the window, as discussed earlier. Also note that in a time-based scheme a processor must dynamically estimate r for each tree, as in a software combining scheme, and store the information in an associative software table.

Finally, in a hardware combining network, tree depth cannot be altered, and one depth applies to all combining trees. Hence, the task of satisfying (2.15) depends on minimizing $T_{\text{trip},\,i}$ through efficient network nodes and network management, and it depends on maximizing $T_{\text{stall},\,i}$ through a sufficient number of processor contexts, a sufficient processor lookahead depth, and a sufficient program lookahead depth.

2.6. Conclusion

What have we accomplished?

We have developed the notion of combining tree for asynchronous MIMD architecture in a novel way and independent of physical implementation, in order to make the essential aspects clear and to make implementation options clear. We have represented accesses to atomic operation in terms of a closed steady-state queuing model (Figure 2.1d). This has enabled us to identify when a combining tree is necessary, and how to parameterize the tree nodes (Figure 2.1e).

In particular, the model led us to the concept of *combining window*, an interval of time in a combining node during which incoming requests are gathered in the node in order to combine them into a single outgoing request. We then showed that the combining window is necessary in order to realize the dual forms of concurrency — execution and storage concurrency — that a combining tree is designed to achieve:

- *Execution concurrency* among the nodes at each level of a combining tree is necessary for the tree to achieve the *speed up* that it is designed to give. Without sufficient execution concurrency, the tree will not achieve the desired speed up.
- *Storage concurrency* among the nodes at each level of a combining tree is necessary for the tree to achieve the *buffer storage* that is required in order to implement the combining of requests. Without sufficient storage concurrency, node buffers will overflow.

More specifically, the combining window allows us to bound decombining buffer size of a combining node as a function of the distance of the node from the root. For a multistage interconnect, the bound for a software combining node is $O((\log_k n) d_{\text{root}}) = O(\log^2 n)$, while the bound for a hardware combining node is $O(d_{\text{root}}) = O(\log n)$, where d_{root} is the distance of the node from the root.

3. Literature Review

What concepts relevant to windows have appeared in the literature?

In his dissertation, Ranade (1989) developed a synchronous combining scheme that is a realization of a concurrent-read-concurrent-write parallel random access machine (CRCW PRAM). The resulting node buffer size is $O(\log n)$, as shown in Chapter 5 of his dissertation. In Chapter 5, Ranade also briefly generalized his synchronous scheme to an asynchronous system by introducing a time parameter τ, such that if a queue in level 0 of the interconnect is empty for time τ, then the respective node will send an end-of-stream message to the next level. The end-of-stream messages propagate through the interconnect.

The effect of the end-of-stream message would be the same as that of a combining window in our scheme if the message were sent whether or not queues were empty. Even so, without the queuing model, it is impossible to determine what the value of τ should be, for it is impossible to quantify its role with respect to overall system performance. Ranade suggests that τ might be as small as the network message system can handle. Our model clarifies that the appropriate mean window size for a combinable variable depends not only on the network processing rate, but also on the decombining buffer capacity — which could be overrun by too small a τ — and on the processor request rate that results from an acceptable program execution rate. Thus, our model clarifies that each combinable variable has its own τ_{indep}, so ideally each request will carry the τ_{indep} of its variable and be managed accordingly.

There has been little published on the topic of combining tree performance evaluation since the original study by Pfister and Norton (1985), in which they identified the problem of *tree saturation* in a multistage interconnect. In a non-combining network, tree saturation occurs when the buffers of nodes along a pathway to a memory hotspot overflow, forcing the nodes to stop accepting requests. This problem, along with the additional problem of decombining buffer overflow, can occur in a combining network under insufficient combining. Their model is a discrete-time open-system model in which processor request rate to a hardware combining network is manipulated, along with fraction of the requests that are combine requests. The combine requests all go to the same memory module, whereas non-combine requests are uniformly distributed across the memory modules. The network has fan-in two.

Several papers (Thomas 1986; Dias, Kumar 1989; Ho, Eager 1989) proposed discarding, rather than combining, requests in order to reduce the request rate to a hotspot. While this can alleviate tree saturation, it will slow down program execution since the request rate $\lambda(n)$ to the atomic operation will be reduced (Figure 2.1d). If the negative feedback strategy of Scott and Sohi (1990) were used instead of combining, the effect would be similar.

The first authors that we are aware of to identify and explore the issue of degree of combining are Lee and his colleagues (Lee et al. 1986; Lee 1989), who pointed out that if the degree of combining is not large enough, then the request rate to parent nodes will exceed their service rate, as we explained in Section 2.3. However, their model does not provide the general perspective that we have provided in terms of a queuing network; hence, they did not identify the need for combining windows. Their model is similar to that of Pfister and Norton, using what we call *implicit combining windows.* In this context, they explained that if buffer entry size is fixed — to keep the hardware simple and fast — entry size must be larger than fan-in k in order to compensate for the fact that k-fold combining will not occur for some batches due to lack of temporal proximity of the requests. They also pointed out that decombining buffer size at the leaves is unbounded as $n \to \infty$. Kang et al. (1991) and Merchant (1992) each presented a more detailed analytic model of the same type of system. Merchant allowed an unlimited batch size and showed that for a finite queue size in a node, the loss rate for non-combine requests is reduced by giving combine requests lower priority in the queue.

In contrast to the foregoing approach, our model contains the parameter $n_{\text{max.reqs.out}}$, which creates a closed system when set to a finite value. Furthermore, we have gone beyond the foregoing approach to identify the duality of execution and storage concurrency among the nodes at each level of a combining tree, and to develop the concept of combining window, which solves the dual concurrency problems. In particular, the combining window solves the problem of node buffer overflow by ensuring sufficient combining, and thereby showing how to bound decombining buffer size as a function of distance d_{root} of a node from the root. In Chapter 2 we presented a bound on the mean in terms of asymptotic notation. In Chapter 4 we will find a bounding probability distribution for buffer size using an analytic continuous-time open-system queuing model. In Chapter 5 we will present the results of simulation experiments of a closed system in order to obtain the distribution of node buffer size under a more realistic model, and in order to determine the effects on execution speed and node buffer size of different window disciplines.

Finally, in a recent Ultracomputer paper, Dickey and Percus (1992) explored the degree of combining by using an analytic discrete-time model that represents a single combining node having fan-in two. Their model is a low-level model, dealing with details of hardware implementation of a combining node, and in this context, they explored two limits on batch size: two and infinity. As with Lee and his associates, they concluded that a limit of two allows insufficient combining for fan-in two.

In contrast to the Dickey and Percus model, our model is a high-level model of an entire combining network, not just a single node. Our model is designed to accommodate arbitrary specialized models and to allow the exploration of high-level issues related to combining tree design, as discussed above and in Chapter 2.

4. An Analytic Queuing Model

How can we model the probability distribution of node buffer size?

As discussed in Chapter 2, we need a probability distribution for the size of the *combining buffer*, and we need a distribution for the size of the *decombining buffer* for a particular distance d_{root} of a node from the root. In this chapter, we will present a simple model, from which more realistic models may be obtained by relaxing constraints of choice, which we discuss at the end of this chapter. The simple model we present here will illustrate the analytic solution strategy without the distraction of complex formula manipulation, since we may appeal to well known properties and relations (Wolff 1989). (Note that the model presented in this chapter was originally formulated in December 1990, in Bitar 1990b, under the assumption that singleton batches are stored in the decombining buffer.)

More specifically, our model will make each standard service center an M/M/1 queue, while combining and decombining service centers will be handled in a manner that is compatible with this. That is, each service center will have i.i.d. exponential (Markovian) inter-arrival times and i.i.d. exponential service times. The respective arrival process is a Poisson process. If the service rate at an M/M/1 queue exceeds the arrival rate (so that the queue length will be stable, not becoming arbitrarily large), the departure process from the queue is a Poisson process having the same rate as the arrival process. Thus, suppose that the arrival sources for an assembly line of standard independent exponential queues are independent Poisson processes, then the arrival processes for all queues will be Poisson processes, making all queues M/M/1.

4.1. Memoryless Distributions

The M/M/1 arrangement makes many derivative distributions simple and makes many relevant random variables independent. The underlying reason is that the exponential distribution is *memoryless:* given that some time has passed in an interval having exponentially distributed length, the distribution of the time remaining in the interval is the same as that of the original interval. The discrete-time version of this is a geometric distribution, which represents the tossing of a biased coin until a head is obtained: given that some number of trials have passed in waiting for the first head, the distribution of the number of trials remaining until the first head is the same as at the first trial. In terms of random variables, X has a memoryless distribution if

$$(X \mid X > t) \sim X + t \tag{4.1}$$

That is, X has a memoryless distribution if, for all $t \geq 0$, conditioning X on its being greater than t yields the same distribution as that obtained by adding t to X. If we express (4.1) in terms of the tail distribution function $P\{ X > x \}$, $x \geq 0$, and if we solve (via several clever but simple steps), we obtain an exponential function for the tail.

$$P\{ X > x \} = [P\{ X > 1 \}]^x = q^x, \text{ for } x \geq 0,\ 0 < q < 1 \tag{4.2}$$

If X is discrete, (4.2) implies that $X \sim \textit{geometric/inclusive}\,(p)$, where $p = 1 - q$, and X is the count of the number of *tosses* (inclusive) until the first head; $E(X) = 1/p$. On the other hand, if X is continuous, it is convenient to express q as a function of e, letting $q = e^{-\lambda}$, $\lambda > 0$, making $X \sim \textit{exponential}\,(\lambda)$, $E(X) = 1/\lambda$. In addition, if X is geometric/inclusive(p), the count of the number of *tails* until the first head is

$X - 1 \sim geometric\,(p)$. While (4.1) is the memoryless property for the geometric/inclusive distribution, (4.3) is the memoryless property for the geometric distribution.

$$(X \mid X \geq t) \sim X + t \tag{4.3}$$

This is evident by letting $t = 0$ in (4.1) and (4.3).

As further perspective on the relation between the exponential and geometric distributions, we find that the exponential is the limit of the geometric concept as the number of trials per time unit goes to infinity. That is, if $X_n \sim$ geometric/inclusive(p/n), then X_n/n is the time until the first success for geometric trials lasting $1/n$ time units. Letting $n \to \infty$, we find $X_n/n \to$ exponential(p), as we would expect, and $\lambda = p$ is the rate parameter.

Geometric (i.i.d. Bernoulli) trials are typically used to model the occurrence of events in a computer system at the level of clock ticks. We see that the continuous-time version of this discrete model is that of i.i.d. exponential inter-event times. For an arrival process, this continuous process is a *Poisson process.* We will use the continuous-time approach here since queuing theory has been expressed in terms of continuous-time models. In this context, when a request arrives at a queue, if there is a request in service with exponentially distributed service time, the amount of time until the request is done has the same distribution as when the service started. Similarly, when the request in service is done, if inter-arrival times are exponentially distributed, as in a Poisson process, the amount of time until the next request arrives has the same distribution as at the moment of the last arrival.

The $negative\text{-}binomial\,(i, p)$ distribution represents the sum of i i.i.d. geometric(p) random variables. This sum is the count of the number of tails that occurs in order to get a specified number, i, of heads in the tossing of a biased coin. Analogously with the geometric distribution, $negative\text{-}binomial/inclusive\,(i, p)$ represents the sum of i i.i.d. geometric/inclusive(p) random variables.

The M/M/1 model must be modified to handle *combining* and *decombining,* for combining creates a batch request, while decombining splits a batch into its component requests. In modeling combining, we must separate the delay due to the work of combining from the delay due to the combining window. Thus, we will represent the delay due to the work of combining as occurring in the service time of the combining server, while we will represent the actual combining as occurring in the combining window. Further, in defining the service disciplines for the combining window and the decombining server, we must take care to ensure that the departures are Poisson, although this turns out to require no special arrangements. We also treat the service of a decombine request as a single job, from the point of view of the sojourn time of a subsequent arrival, in order to avoid the non-trivial complexity of modeling partial completion of a batch. This increases our upper bound somewhat.

4.2. Basic Setup

The system is open (in contrast to Figure 2.1d), and has two networks, one carrying requests to memory and one carrying requests from memory, as shown in Figure 4.1a. Processor requests to the combining trees are independent Poisson processes having rate r to each tree. Normal processor requests to memory (uniformly distributed across the memory banks) are independent Poisson processes at rate s. Each normal request is acknowledged by memory, making the return rate identical. We assume that requests all have the same size with regard to processing at a link node, although accommodating random size is not difficult as long as the size is distributed geometrically. We will see how to handle geometrically distributed sizes in dealing with decombine requests.

Each tree has fan-in k, and each physical combining node handles requests for all m combining trees, but not normal memory requests. More specifically, each physical combining node has two processors, a combining server for requests to memory and a decombining server for requests from memory; the processors access the decombining buffer with no conflict. We assume two processors per node, no buffer conflicts,

Figure 4.1. Components of Round Trip to Memory from Distance d_{root}.

The round-trip transit time to memory is used to determine the distribution of the decombining buffer size.

Figure 4.1a. Network Configuration.

An arc label refers to the request rate on that arc. All request streams are Poisson processes.

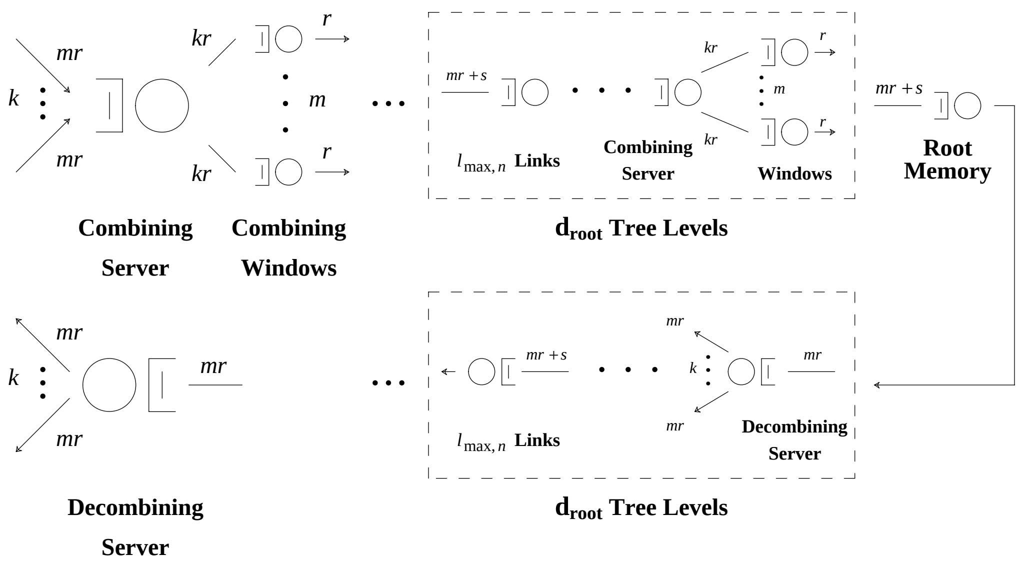


Figure 4.1b. Round Trip Components.

Decombining buffer size is based on the sum of five component sums, each representing a service center. Each tree contributes five respective sums to the total sum. The last column of this table lists the sum limits for these component sums for one tree. NB(c, p) = negative binomial(c, p).

Service Center	**Arrival Rate**	**Service Rate**	**Ratio**	**Sojourn Rate**	**Count**	**Sum Limit**
	λ	μ	$\rho = \lambda/\mu$	$\mu\bar{\rho}$	c	NB(c, p)
			$\bar{\rho} = 1 - \rho$	$\delta = \mu\bar{\rho} = \mu - \lambda$		$p = \delta/(\delta + r^*)$
Combining	mkr	$m\mu_{\text{comb}}$	kr/μ_{comb}	$m\mu_{\text{comb}}\bar{\rho}_{\text{comb}}$	d_{root}	N_{comb}
Window	kr	$w = kr/(k-1)$	n/a	w (let $\delta = w$)	d_{root}	N_{window}
Decombining	mr	$m\mu_{\text{comb}}/k$	kr/μ_{comb}	$(m\mu_{\text{comb}}/k)\,\bar{\rho}_{\text{decomb}}$	$1 + d_{\text{root}}$	N_{decomb}
Link	$mr + s$	μ_{norm}	$(mr+s)/\mu_{\text{norm}}$	$\mu_{\text{norm}}\bar{\rho}_{\text{link}}$	$2\,l_{\max,n}\,d_{\text{root}}$	N_{link}
Root Memory	$mr + s$	μ_{mem}	$(mr+s)/\mu_{\text{mem}}$	$\mu_{\text{mem}}\bar{\rho}_{\text{mem}}$	1	N_{mem}

and two networks, in order to gain independence of the combining and decombining services and in order to allow all servers to have Poisson arrival rates. We also exclude normal requests from the combining node since they would have a shorter service time than combine and decombine requests, so they could not be handled by the same M/M/1 queue.

It follows that combine requests arrive at a combining server at rate mkr, while decombine requests arrive at a decombining server at rate mr. The arrival rates are listed in Figure 4.1b. We further assume that combine and normal arrival rates to a forward communication link and to a root memory are mr and s, respectively, while the decombine and normal arrival rates to a backward link are the same. Due to our assumptions about source arrivals and service times, the arrival streams to a service center are independent Poisson processes, hence their composite stream is a Poisson process.

4.3. Combining Buffer Size

The distribution of combining buffer size depends on the combining window service discipline and its arrival process. So let us discuss this first, then determine the distribution of the buffer size S_{comb}.

A combining window is a service center in which service times are i.i.d. exponential(w) and in which a service completion clears the queue and creates a batch departure. We would like w sufficiently greater than r so that in discarding empty batches, the resulting request rate is r, or equivalently, the mean batch size of non-empty batches is k. That is, $w = 1/\tau_{\text{indep}}$ (Figure 2.4b).

In order to determine the distribution of buffer size S_{comb}, let us assume that a combining entry has size equal to the number of requests it contains. As a first cut, let us allow empty entries. An entry, then, has size equal to the number of i.i.d. exponential arrivals at rate kr that arrive during an independent exponential window that closes at rate w. Due to the memoryless property of the exponential and the geometric distributions, a race between two independent i.i.d. exponential processes produces a geometric count of the number of one type of event (a ‘‘failure’’) that occurs before an event of the other type (a ‘‘success’’) occurs. Because we discard counts of zero, we condition the geometric distribution on the property that the count is at least one. Due to the memoryless property shown in (4.3), it follows that entry size has the distribution of K, where K is geometric/inclusive(p) and probability of success $p = w/(kr + w)$. Mean entry size is $E(K) = 1/p$, and setting this equal to k, we find $w = kr/(k-1)$. In discarding batches having size zero, we are left with a departure stream having rate $P\{ K - 1 > 0 \} w = (1-p) w$, which we find is r, as it should be.

Discarding batches having size zero generalizes to partitioning the window departures into streams according to batch size, in which case the stream with batch size i would have rate $P\{ K - 1 = i \} w$, $i = 0,1,2,...$. Are these streams Poisson processes? To answer this question, notice that there are two facets to a batch departure: the inter-departure interval and the batch size. The first facet is created by a Poisson process having rate w, while the second is created by a Poisson process that has rate kr and that interacts with the first Poisson process.

Now in order to be a Poisson process, an arrival stream must have *three properties.* One property is this: the probability that two or more events occur in an interval of size h is $o(h)$, that is, it goes to zero faster than h does. This property obviously holds for the individual streams because the departures are created by a Poisson process. The other two properties are that the streams must have *independent* and *stationary* increments: the number of events in an interval must be independent of the number of events in a disjoint interval, and the distribution of the number of events in an interval must be independent of the location of the interval in time. But each of the two processes that create the batch departures has independent and stationary increments, so the resulting stream for any particular batch size will have independent and stationary increments. Therefore, each stream having a particular batch size is a Poisson process.

The combining buffer size S_{comb} is, in turn, the sum of m i.i.d. random variables $K_i \sim$ geometric/inclusive($1/k$), making the size negative-binomial/inclusive(m, $1/k$). This is an upper bound in terms of number of requests. It may be converted to the number of bits by multiplying by the number of bits per request, which we assume is constant (or nearly so).

We could, in addition, add to each K_i an overhead term that depends on K_i, such as $\lceil K_i / k \rceil$. The dependence complicates the sum, so a reasonable simplification would be to add on $\lceil K_i' / k \rceil$, where K_i' and K_i are i.i.d.

4.4. Decombining Buffer Size

The distribution of decombining buffer size is determined by marking the departure of a request from a combining window, and then counting the number of requests having batch size greater than one that the window produces during the round trip of the marked request to memory via the root of its tree. Since a batch size of one entails no combining, it will not be stored in the decombining buffer. Just the same, to maintain the simplicity of the model, we will assume that a combine request representing a batch of size one returns through the respective decombining server. Although this will increase our upper bound, without this assumption a round-trip time would be more complicated to represent since it may entail levels in which the decombining server is skipped.

More specifically, for tree i, let N_{ij} be the number of batch requests produced during the marked request sojourn at service center j. If we sum the N_{ij} over j and then add one for the contribution of the marked request to the buffer, we obtain $N_{i+} + 1$. If we sum these terms over i, we obtain the total number of batch requests $N_{\text{total}} = N_{++} + m$ in the buffer. N_{total} is the number of $K_i^* = (K_i \mid K_i > 1)$ that are summed to get the unconditioned decombining buffer size S_{decomb}, where $E(K_i^*) = k^*$, from (2.9). Since we are modeling an open system, the range of S_{decomb} extends to infinity, so S_{decomb} should be conditioned on the maximum number of requests at the node $n_{\text{max.requests/node}}$, shown in (2.3): $(S_{\text{decomb}} \mid S_{\text{decomb}} \le n_{\text{max.requests/node}})$.

To get an upper bound case, we will assume that a parent and child in a tree are separated by the maximum number of links $l_{\max,n}$. A more complex model could give a distribution to this value. A round trip to the root, in our scenario, will contain $2\,d_{\text{root}}$ stages, each stage containing either a combining or a decombining node, along with $l_{\max,n}$ links. A visit to a combining node contains a visit to a combining server and a visit to a combining window. Also, after visiting its root, the marked request visits memory.

For convenience, from this point on, when we speak of combining windows closing at the origin node, we will implicitly mean *windows producing batches of size greater than one.* These will close at rate $r^* = P\{K > 1\}\, r = (1-p)^2 w$, giving us i.i.d. window sizes ~ exponential(r^*). To determine the number of windows that close at the origin during the round trip of the marked request, we determine the number of windows that close during each sojourn of the request at a service center along the way. But, as we will see, a visit to a service center will entail an exponential sojourn time, thus the number of windows that close at the origin during each sojourn time is geometric, so the sum over these sojourns for a particular kind of node, such as a link node, is negative binomial. The last column in Figure 4.1b shows the negative binomial distribution for each service center. The random variables in that column are summed to get the number of closed windows for one tree.

Let us now cover details of note for each service center. We will take the service centers in order of increasing complexity. The reader is invited to refer to the appropriate row of Figure 4.1b when discussing each center. Keep in mind that for an open system model, we need the service rate μ to exceed the arrival rate λ. This is assumed to hold in the figure.

Combining window. In this case, the exponential(r^*) window process at the origin races against the exponential(w) that the marked request is visiting. The number of windows that close at the origin, then, is geometric(p), where $p = w/(w + r^*)$, as shown in the table. This row is different from all others in the table because this is not a standard queue; in particular, there is no value for ρ.

Combining server, root memory, and communication link. Here the race is against the sojourn time at an M/M/1 queue. For an M/M/1 queue, the number of requests found in the service center on arrival is $N \sim$ geometric($\bar{\rho}$), where $\bar{\rho} = 1 - \rho$, and $\rho = \lambda/\mu$ = (arrival rate)/(service rate). Thus, the sojourn time is the sum of $N + 1$ i.i.d. exponential(μ) service times, which, due to the memoryless property of both distributions, is simply exponential($\mu\,\bar{\rho}$). That is, due to the equivalence of geometric/inclusive and exponential, as discussed with regard to (4.1) and (4.2), the geometric/inclusive sum of i.i.d. exponentials is exponential.

Decombining server. As mentioned earlier, we treat the service of a decombine request as a single job, from the point of view of the sojourn time of a subsequent arrival, in order to avoid the non-trivial complexity of modeling partial completion of a batch; that is, in order to avoid dealing with the probability distribution of the amount of a batch job in service that has been completed at the time of an arrival of another batch. This distribution is specified through a generating function, rather than a closed form for the distribution. This simplification increases the sojourn time of the marked request, at worst, by the service time of one decombine batch for each of the $1 + d_{\text{root}}$ decombining servers in the trip. Since the batch size is geometric/inclusive($1/k$) and since the service times for the component requests are i.i.d. exponential($m\,\mu_{\text{comb}}$), the service times of the batches are i.i.d. exponential($m\,\mu_{\text{comb}}/k$).

The sojourn time of a component request of a batch will also depend on its service position within its own batch. However, since batch size is geometric/inclusive, the position of a random sample from a batch surprisingly turns out to have the same distribution. Consequently, we can treat the marked request as being the last job of the batch without increasing its sojourn time, hence without increasing our upper bound. This simplifies our representation of the sojourn time, because this completes the notion of treating batch service as the service of a single job. Hence, the sojourn time of the marked request at a decombining server is completely analogous to the sojourn time at a combining server.

Notice that a decombining server splits its batch into component requests, resulting in k independent streams of requests. Are these streams Poisson processes? We must check to see that the properties discussed earlier hold for each of the k component streams. Only the properties of independent and stationary increments are in doubt. But we can see that both of these properties hold for a component stream because the batch arrival stream has these properties and, in turn, because the batch sizes are geometric/inclusive and geometric trials have these properties, and finally because the k streams that created each batch were independent.

Summing the Terms. For any tree $i = 1,...,m$, the last column of the table lists the terms N_{ij}, which are independent negative binomial(c_j, p_j), where j is the service center (row). The sum of the N_{ij}, plus m, yields $N_{\text{total}} = N_{++} + m$, which is the sum limit for determining decombining buffer size, as in (4.4).

$$S_{\text{decomb}} = \sum_{l=1}^{N_{\text{total}}} K_l^* \tag{4.4}$$

However, we cannot calculate S_{decomb} using N_{total} since the N_{ij} have different success probabilities for distinct j.

Rather, we must first sum the N_{ij} over $i = 1,...,m$, yielding the component sum limits N_{+j}, which are independent negative binomial(mc_j, p_j). Then we must approximate the respective component sums using independent random variables $X_j \sim$ normal(μ_j, σ_j^2), for appropriate μ_j, σ_j^2, which we determine below. This yields estimate $\hat{S}_{\text{decomb}}$ in (4.6).

$$S_{\text{decomb}} = \sum_{j=1}^{\#\text{ service centers}} \sum_{l=1}^{N_{+j}+1} K_l^* \tag{4.5}$$

$$\hat{S}_{\text{decomb}} = \sum_{j=1}^{\#\text{ service centers}} X_j \tag{4.6}$$

This is justified by a generalization of the Central Limit Theorem since the K_l^* are i.i.d. For a sum limit $M_i \sim$ negative binomial(i, p), M_i converges in distribution to infinity as $i \to \infty$. That is, $P\{ M_i \le j \} \to 0$ as $i \to \infty$ for $j = 0,1,...$. This means that the M_i will go to infinity, although not necessarily monotonically.

So using the Central Limit Theorem for small i in this case has weaker justification than if the M_i were constants going to infinity monotonically.

The mean and variance of a component sum S, having sum limit N, are determined using conditional expectation: $E(S) = E[\,E(S\,|\,N)\,]$, $V(S) = E[\,V(S\,|\,N)\,] + V[\,E(S\,|\,N)\,]$. This is straightforward in our cases since S and N are independent. Letting $q_j = 1 - p_j$, this yields $\mu_j = mk\,c_j\,q_j/p_j + k$, and $\sigma_j^2 = (mk\,c_j\,q_j/p_j^2)[\,(k-1)p_j + k\,] + k(k-1)$. Since the X_j are independent normal(μ_j, σ_j^2), their sum $\hat{S}_{\text{decomb}}$ is normal($\sum_j \mu_j, \sum_j \sigma_j^2$).

Finally, a normal distribution has infinite negative and positive range, so we obtain our final estimate by truncating at 0 and at $n_{\text{max.requests/node}}$ from (2.3), and then normalizing with respect to $P\{\,0 \le \hat{S}_{\text{decomb}} \le n_{\text{max.requests/node}}\,\}$, yielding random variable $(\,\hat{S}_{\text{decomb}} \mid 0 \le \hat{S}_{\text{decomb}} \le n_{\text{max.requests/node}}\,)$ as our final estimate. Actual values of this distribution are obtained by converting probability expressions to expressions in terms of the normal(0,1) distribution.

Mean of the Sum. Let us relate $E(S_{\text{decomb}})$ to the upper bound we obtained in (2.5) for the mean decombining buffer size for a physical combining node: $O(mk\ r_{\max*}\,t_n\,l_n\,d_{\text{root}})$, where $r_{\max*}$ is the maximum request rate over all m trees and t_n is the mean sojourn time at a link.

In our network, the relevant term of S_{decomb} is the N_{link} term since it contains the number of links factor in the bound expression. The corresponding sum limit over all trees is negative binomial(mc, p), so due to independence the respective sum of the K_i^* over all trees has mean $mk\ cq/p\ = O(mk\ c/p)$, for $q = 1 - p$. But $c = 2\,l_{\max,n}\,d_{\text{root}}$, and $1/p = 1 + r^*/(\mu_{\text{norm}} - (mr + s))$, where $r^* = \Theta(r)$, $r = r_{\max*}$, and $1/(\mu_{\text{norm}} - (mr + s))$ is the mean sojourn time at a link. Thus, we see that the expression $O(mk\ c/p)$ contains all of the terms in $O(mk\ r_{\max*}\,t_n\,l_n\,d_{\text{root}})$, with the replacement of l_n by $l_{\max,n}$.

4.5. Conclusion

Weakening the assumptions. The M/M/1-based model that we have presented illustrates the analytic solution strategy; however it is simple and not highly realistic. How can we make it more realistic?

The simplest feature to add is to accommodate a different *request rate* r_i for each of the m trees. This would simply require splitting each composite arrival stream at rate mr, into m distinct arrival streams, each having its own rate r_i; mr would then be replaced by the sum of the r_i in Figure 4.1b. Notice that none of the service times would be affected by this change. In particular, the decombining service time would not be affected because batch size depends only on k, hence is independent of r_i.

Accommodating a different *fan-in* k_i for each of the trees implies that the model is a software combining model. This would require replacing k by k_i, and for the combining server replacing mkr by the sum of the $k_i\,r_i$.

We would also like to represent different *request classes* at the same server, in order to allow combining and decombining servers to handle both combine or decombine requests and normal memory requests, as in hardware combining nodes. The service times for distinct request classes will differ, so this takes us to the M/H/1 model, where H refers to hyper-exponential service time, a probabilistic mixture of several exponential service-time distributions.

Another important feature is to allow *feedback* in the network, in order to represent a combining node that has only one processor for servicing both combine and decombine requests. This takes us to modified Jackson networks that accommodate request classes. This approach would also give us a more realistic model of two-processor combining nodes, as in Figure 4.1a, for we could now represent the decombining buffer as a service center, in order to model contention for it between the two processors.

In addition, we would like to consider the effects of a *closed system*, since a real computer system, in our context, will be closed. A closed system would give us a smaller bound on buffer size because a processor's time-average request rate, given i outstanding requests, would be strictly decreasing in i. It would also be useful to represent service time as being constant, or nearly so, as it would be in a real system.

Finally, to push curiosity to the limit, we could represent different-sized requests, we could represent batch service at the decombining centers, and we could study transient behavior.

As we relax our assumptions in order to develop a more and more realistic analytic model, the tractability of the analytic model will decrease, and it will eventually (maybe quite soon) become necessary to switch to a simulation model in order to obtain the information that we wish to see, in particular, the effect of window discipline on execution speed and node buffer size.

Simulation. We leave to future research the effort of pushing the analytic endeavor to its limit, and we turn, now, to the medium of simulation, in which we can easily formulate a more realistic model. We use a simulation model in order to run experiments by which we investigate the effect of window discipline on execution speed and node buffer size.

5. Simulation Experiments

We turn, now, to investigating the effect of combining window discipline on execution time and node buffer size using a generic simulation model, which we specialize as appropriate. Our generic simulation model is that of a queuing system that allows weaker, more realistic assumptions than those that an analytic model can allow.

5.1. Model

Simulator. The simulations have been done using the Ptolemy simulation system, a schematic-based simulation system written in C++ and developed by the Ptolemy research group in the Electrical Engineering and Computer Sciences department at the University of California, Berkeley. I have augmented Ptolemy with 15,000 lines of C++ code, 7000 lines of Ptolemy preprocessor code, and 50 Ptolemy schematics, called galaxies. Most of this code and these schematics are general-purpose tools.

Network. As shown in Figure 5.1, our network represents a hardware combining network. The network is closed and contains four processors and four memories connected by a 4×4 multistage combining network for handling requests going to memory and by a 4×4 multistage decombining network for handling requests coming from memory. Each combining buffer in the combining network has a private zero-delay connection to the respective decombining buffer.

Parameters. The parameters of the simulation model are shown in Figure 5.2. The simulator has many parameters, and we manipulated only a few of these parameters in the set of experiments reported here — our initial set of experiments — so we have only begun to tap the capabilities of the simulator.

Regarding the master control parameters (Figure 5.2a), these parameters specify the number of processors, the random-number generator seeds, and the first and last iteration numbers in a simulation, which determine the number of i.i.d. iterations in the simulation.

Multiple i.i.d. iterations allow statistical estimates and tests to be computed across iterations of a simulation. Once the simulation reaches steady state during an iteration, the mean of a dependent variable fluctuates very little about its own mean as the variance of the dependent variable gradually approaches zero, so the mean and variance should become independent of the initial conditions. Thus, the purpose of multiple iterations under this simulation paradigm is just to verify that the independence of the initial conditions has been reached. Having no reason to doubt this independence, we ran multiple iterations for only a couple simulations. Regarding the variable, the first iteration number, this is a parameter so that a terminated simulation may be continued at the beginning of an iteration that it did not complete.

The random-number generator is the Gnu multiplicative linear congruential generator (MLCG), which is an implementation of the MLCG described in Park and Miller (1988). This generator takes two seeds, which are output at the end of each iteration.

Regarding the processor parameters (Figure 5.2b), these parameters configure the processors, all of which are configured identically and, with respect to probability distributions, are configured independently since

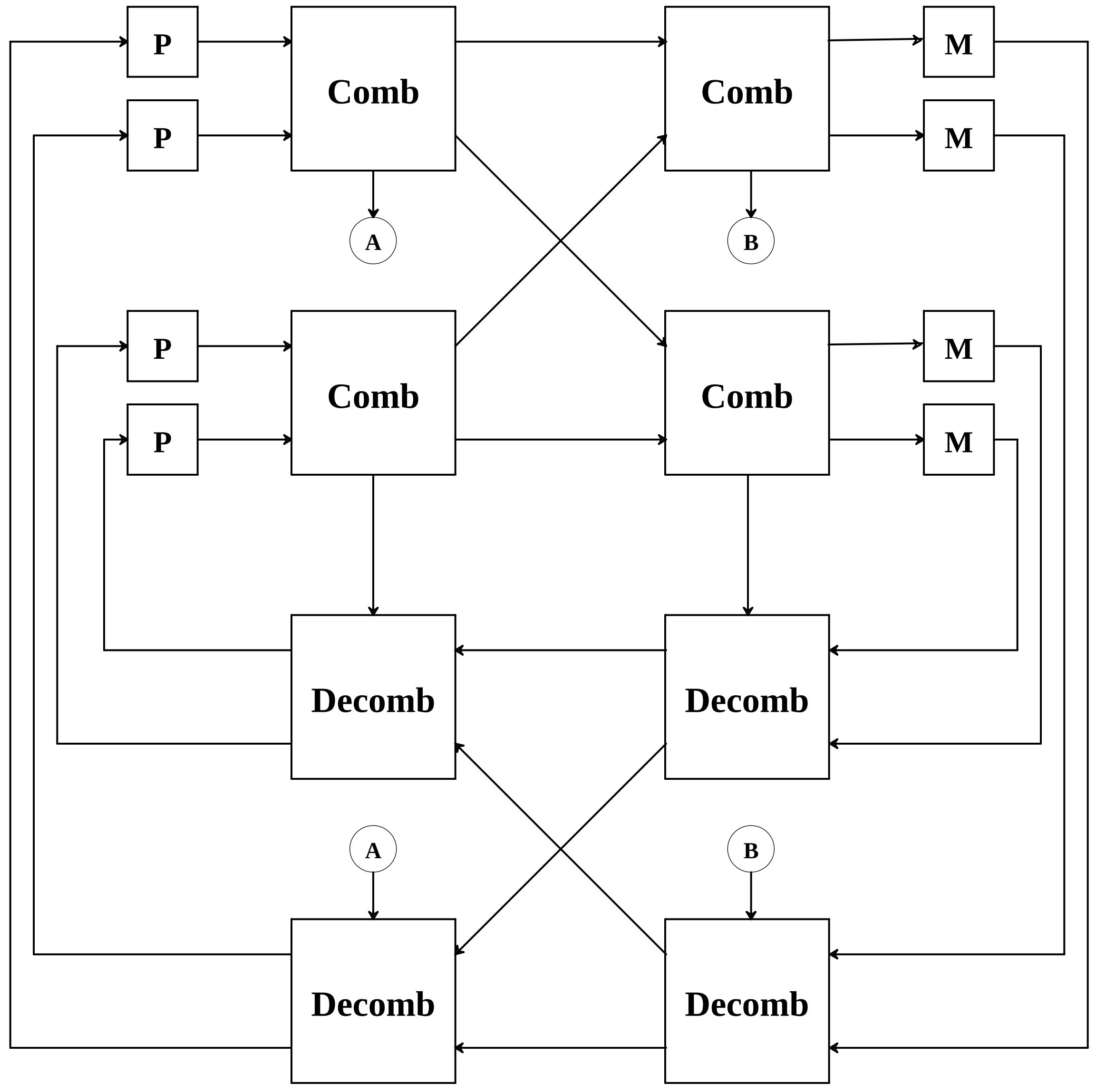


Figure 5.1. The Simulation Network.

P = processor, M = main memory

Comb = combining buffer and 2×2 switch

Decomb = decombining buffer and 2×2 switch

Figure 5.2. Simulation Model Parameters.

Figure 5.2a. Master Control Parameters.

- # processors n: May be either two or four. Set to four in our experiments.
- RNG seeds: Random-number generator (multiplicative linear congruential generator) seeds — two seeds.
- First and last iteration numbers: These determine the number of i.i.d. iterations in a simulation. Set to one in our experiments.

all random draws, both within and between processors, are independent. We treat the number of replies till done as a constant, although a more general model would treat it as a random variable. In our experiments, the stop-when-done flag is false so that the steady-state pattern of requests to the network will continue unchanged until all processors are done, at which point the iteration will terminate and final statistics will be computed. This strategy also makes it unnecessary to implement an end-of-window request under the discipline of constant batch size, which would otherwise be necessary to cause incomplete combine entries to be treated as complete, as discussed in Section 2.3. Also, the compute-time distribution is either constant or uniform, thereby allowing the variance to be manipulated without changing the mean.

Regarding the combining-buffer, decombining-buffer, memory, and network parameters (Figure 5.2c), these parameters specify the time that respective operations take, along with the initial value of the memory. All time values except for memory update time and network inter-unit delay are set to zero in our experiments because non-zero values would not contribute to the purpose of these experiments. The network inter-unit delay is a parameter that allows us to lengthen round-trip time without imposing further serialization on requests, so if it were to be conceptualized realistically, it would consist of an inter-unit pipeline whose stages had the length of the memory update time.

Statistics. The statistics gathered by the simulator are shown in Figure 5.3, along with qualifying notes. The actual simulated execution time of an iteration is recorded but is not of interest because it is largely determined by the segment of the computation prior to reaching steady state. The hypothetical mean execution time is of interest because it integrates the steady-state means of the relevant parameters — processor inter-request time and round-trip time. The formula for hypothetical mean execution time assumes that, in the execution of a hypothetical program, the number of replies per processor and the processor inter-request time are uncorrelated, allowing the respective means to be multiplied to obtain the mean of the product. In our experiments, the mean number of replies is assumed to be 10,000, which determines the relative weight of inter-request time and round-trip time in hypothetical execution time. Since we were not concerned with the location of the cross-over points, the effect of hypothetical execution time for us was simply that the mean round-trip time determines the asymptote of a curve, while the mean inter-request time largely determines the rest of the curve.

Figure 5.2 (continued). Simulation Model Parameters.

Figure 5.2b. Processor Parameters.

- # replies till done: # replies that a processor must receive until it is done executing its hypothetical program. Set to a sufficiently large value, for each experiment, to bring all iterations in the experiment to steady state — 17,000 or more replies for the simulations reported here.
- Stop-when-done flag: Indicates if a processor should stop issuing requests once it is done executing its hypothetical program. Set to false in our experiments.
- Max # outstanding requests: Maximum number of outstanding requests $n_{\text{max.reqs.out}}$ that a processor may issue. May vary from 1 to infinity. Once a processor has exhausted this limit, it must wait for a reply before issuing another request.
- Compute-time probability distribution: Constant or uniform. The mean of this distribution is $1/r_{\text{max}}$. After a processor terminates a computation period (in its execution of the hypothetical program), the processor receives waiting replies, issues a request to the network — either a combine or non-combine request — and, if its maximum number of outstanding requests is not exhausted, initiates another computation period.
- Combine request probability: The probability that a processor's request will be a combine request, as opposed to a non-combine request. Set to one in our experiments.
- # non-combine destinations: May vary from one to four memory modules. When a non-combine request is issued, the destination is chosen by a uniformly distributed draw among the non-combine destinations.
- Vector of combine destinations: May contain any combination of memory module ids 0,1,2,3. Set to the scalar zero in our experiments since here we are exploring the effects of system configuration on a single combining tree.
- Vector of window disciplines (corresponds to vector of combine destinations): A discipline may be either constant batch size or constant window size.
- Vector of window parameters (corresponds to vector of window disciplines): For the discipline of constant batch size, a window parameter is β, whereas for the discipline of constant window size, a window parameter is τ_{indep}.
- Vector of combine operations (corresponds to vector of combine destinations): A combine operation may be add, multiply, minimum, or maximum. Irrelevant to our experiments since differences in operation duration are not represented in the simulation model.
- Combine request data value: The data value that is sent with a combine request. Irrelevant to our experiments.
- Request-receive time, request-create time: Set to zero in our experiments.

Figure 5.2 (continued). Simulation Model Parameters.

Figure 5.2c. Combining Buffer, Decombining Buffer, Memory, Network Parameters.

- Combining buffer parameters: Lookup time, combine time, batch-create time, request-create time, route time. Set to zero in our experiments.

- Decombining buffer parameters: Lookup time, combine time, request-create time, route time. Set to zero in our experiments.

- Memory parameters: Initial data value, update time, request-create time. In our experiments, request-create time is set to zero, and the initial data value is irrelevant.

- Network parameters: Inter-unit delay. This is a constant value that delays a request prior to its arrival at a processor, a combining buffer, a decombining buffer, or a memory module; thus, in our network it increases round-trip time by $6 \bullet$ inter-unit delay.

5.2. Strategy

We present three experiments. The first experiment compares the effects of the two basic window disciplines, constant batch size and constant window size. The second experiment compares the effect of combining, under constant window size, with non-combining when total processor request rate λ is the same as memory service rate μ_{mem}. The third experiment explores the effect of reducing constant window size τ below processor mean inter-request time $1/r_{min}$. Figure 5.4a presents the dependent variables of the experiments.

Regarding the independent variable processor request rate r, in our experiments we will define $r_{min} = r_{max}$, and unless otherwise qualified, we will let $r = r_{min}$, $\lambda = \lambda_{min}$.

Regarding the independent variable constant window size τ, since τ_{indep} is the focus of attention here, in this chapter we define $\tau = \tau_{indep}$, rather than $\tau = \tau_{dep}$ as in Chapter 2. Furthermore, we set τ_{indep} as if it were τ_{dep}, say to $1/r$, because the probability of an empty batch will vary according to system configuration and level in the tree. This does not create a problem for interpreting our experimental results, and, where relevant, we point out the effect on our data of the fact that actually $\tau_{dep} > \tau_{indep}$.

Window discipline. In the first experiment, we investigate the two basic combining window disciplines — constant batch size and constant window size — and observe their effect on the dependent variables shown in Figure 5.4a, but in particular on hypothetical execution time, decombining buffer utilization, and memory queue utilization, where utilization is the amount of an infinite buffer/queue that is utilized, averaged over time. *Hypothetical execution time* allows us to compare the two window disciplines in terms of the time cost incurred in executing the hypothetical program, while *buffer+queue utilization* allows us to compare the two disciplines in terms of the space cost incurred in executing the hypothetical program. As explained in Section 2.3, constant window size should reduce the inter-request time variance, and hence the memory queue utilization. Figure 5.4b shows the values of the independent variables whose values are not specified in Figure 5.2.

Regarding the maximum number of outstanding requests $n_{max.reqs.out}$, the results of the experiment are evaluated by plotting, for a given window discipline, the steady-state value of a dependent variable as a

Figure 5.3. Simulation Model Statistics.

- Notes
  - □ Statistics: For each dependent variable other than execution time, the following statistics are calculated as a function of simulated time (for appropriately chosen time points) for each iteration of an experiment: sum of weights, minimum, mean, maximum, variance, standard deviation, and frequency distribution, where the distribution is defined on a set of intervals that is chosen to provide a useful collapsing of the variable's values.
  - □ Time-based *vs.* event-based statistics: A time-based statistic is one in which the weight of a data value is the length of time for which the data value is descriptive of the relevant network component, such as the length of a queue. An event-based statistic is one in which the data weights are all equal, hence may be set to one without loss of generality.
  - □ Individual *vs.* collapsed statistics: Processor statistics are collapsed across processors (that are in use); combining buffer statistics are collapsed across combining buffers (that are in use) for each network level; decombining buffer statistics are collapsed across decombining buffers (that are in use) for each network level; memory statistics are collapsed across memories (that are in use), but statistics for the individual memory queues are gathered, as well.
- Processor variables
  - □ Time-based statistics: Busy time, # requests outstanding, queue length
  - □ Event-based statistics: Inter-request time, round-trip time
- Combining buffer variables
  - □ Time-based statistics: Utilization in terms of # requests, utilization in terms of # batches (at most one batch at a time in our experiments), queue length
  - □ Event-based statistics: Inter-request time, batch size of request being output, batch size of window just ended (for discipline of constant window size), age of request being output
- Decombining buffer variables
  - □ Time-based statistics: Utilization in terms of # requests, utilization in terms of # batches, queue length
  - □ Event-based statistics: Round-trip time
- Memory variables
  - □ Time-based statistics: Queue length (includes request in service)
- Execution time
  - □ Actual execution time: Amount of simulated time till all processors receive the per-processor number of replies in the hypothetical program.
  - □ Hypothetical mean execution time: Mean number of replies per processor $\bullet$ mean processor inter-request time + mean round-trip time from processor. This formula assumes that the number of replies per processor and the processor inter-request time are uncorrelated. In our experiments, the mean number of replies is assumed to be 10,000.

function of the maximum number of outstanding requests allowed. Hence, the values of the maximum number of outstanding requests were chosen to fill in the respective curves with the amount of detail needed at a given region of the curve and, in addition, so that the last finite value yields the same results as the infinite value. The last such finite value varies from configuration to configuration, and is determined by first running the infinite case and noting the maximum value of the dependent variable '# requests outstanding' (Figure 5.3). The significance of the maximum number of outstanding requests is that the smaller this number, the more often a processor will exhaust its allotted number of outstanding requests, hence the more often a processor will have to wait for a reply before it will be able to continue its computation and issue further requests that result from that computation. Thus, as the maximum number of outstanding requests decreases, the effect of transit time will grow. We will see the results of this effect in our data.

Regarding the compute-time distribution, the two distributions were chosen to have the same mean (one) but a zero and a non-zero variance, respectively, so that the interaction of inter-request time variance and window discipline could be studied, in particular, the effect of constant window size in smoothing out lulls and bursts in the processor request streams. The compute-time mean of one was chosen to match the memory service time, giving us a total processor request rate that is four times the memory service rate. This creates a need for combining and allows a two-level binary combining tree to reduce the processor request rate to the memory service rate using a mean batch size $\beta = 2$. In terms of the notation of Figure 2.4, we have $\lambda(4) = 4r = 4\mu_{mem}$, with the level-0 request rate to memory equaling the memory service rate μ_{mem}. This match is appropriate, for under a closed system the desired request rate to a service center need not be less than the service rate to obtain a stable queue, as it generally must be in an open system.

Regarding window discipline, the batch size of two was chosen to match the fan-in of two, and the window size of one was chosen to match the mean processor inter-request time $1/r = 1$.

Combining *vs.* non-combining. In the second experiment, we compare the effect of combining, under constant window size, with non-combining when total processor request rate λ is the same as the memory service rate μ_{mem}. In Section 2.2 we motivated the use of combining when λ exceeds the service rate of the

Figure 5.4. Simulation Experiment Variables.

Figure 5.4a. Dependent Variables.

Note: These variables are defined in Figure 5.3.

- Processor variables: Utilization (busy time), # outstanding requests, inter-request time, round-trip time
- Combining buffer variables: Utilization in terms of # requests, inter-request time, batch size
- Decombining buffer variables: Utilization in terms of # requests, round-trip time
- Memory variables: Queue length
- Hypothetical mean execution time: Mean number of replies per processor • mean processor inter-request time + mean round-trip time from processor, where 10,000 is the mean number of replies per processor in the hypothetical program.

Figure 5.4 (continued). Simulation Experiment Variables.

Figure 5.4b. Independent Variables for Experiment on Window Discipline.

Note: These variables are defined in Figure 5.2.

- Max # outstanding requests $n_{\text{max.reqs.out}}$: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, infinite
- Compute-time distribution:

  Constant one ($r = 1$, variance = 0), uniform(0,2) ($r = 1$, variance > 0)
- Window discipline: Constant batch size of two ($\beta = 2$), constant window size of one ($\tau = 1$)
- Memory update time: one ($\mu_{\text{mem}} = 1$)
- Network inter-unit delay: zero

atomic operation of interest, whereas here we consider the case where λ does not exceed this service rate.

Why is this case interesting? This is interesting because of the effect that constant window size has on reducing the inter-request time variance and hence on reducing the memory queue length, which will be explored in the window discipline experiment, as discussed above. Hence, even with $\lambda = \mu_{\text{mem}}$, combining under constant window size should give us a reduction in memory queue length, and hence in round-trip time, which is important in satisfying (2.15). In addition, although non-combining incurs no decombining buffer utilization, it will incur memory queue utilization. As in the first experiment, we plot the value of a dependent variable as a function of the maximum number of outstanding requests allowed $n_{\text{max.reqs.out}}$.

Figure 5.4c shows the values of the independent variables whose values are not specified in Figure 5.2. The reason for the large values of τ is to be able to compare the results of this experiment with those of the third experiment on window size.

Window size. In the third experiment, we investigate the effect of reducing window size τ below mean processor inter-request time $1/r$. We are interested in observing the effect on hypothetical execution time and decombining buffer utilization, since memory queue utilization should not vary among the experimental conditions. As in the other two experiments, we plot the value of a dependent variable as a function of the maximum number of outstanding requests allowed $n_{\text{max.reqs.out}}$.

Figure 5.4d shows the values of the independent variables whose values are not specified in Figure 5.2. The only difference between the values in this experiment and those in the prior experiment is the value of the network inter-unit time. This is increased to a positive value, in this experiment, in order reduce the fraction of the round-trip time that consists of window sojourn time. This will enable us to see the effect of reducing window size when it does not appreciably reduce round-trip time. Notice that we include cases of $\tau > 1/r$ in order to obtain k-fold combining for $k > 2$. If fan-in k were greater than two in our network, we could obtain k-fold greater than two-fold combining with $\tau = 1/r$.

More specifically, with respect to reducing window size, there are three factors that interact to determine whether decombining buffer utilization will increase or decrease:

Figure 5.4 (continued). Simulation Experiment Variables.

Figure 5.4c. Independent Variables for Experiment on Combining vs. Non-Combining.

Note: These variables are defined in Figure 5.2.

- Max # outstanding requests $n_{\text{max.reqs.out}}$: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, infinite
- Compute-time distribution: Uniform(0,2) ($r = 1$, variance > 0)
- Window discipline: Constant window size of $\tau = 4, 2, 1, .75, .5, .25, 0$
- Memory update time: .25 ($\mu_{\text{mem}} = 4$)
- Network inter-unit delay: zero

- Increase in arrival rate: Works to increase buffer utilization
- Decrease in round-trip time: Works to decrease buffer utilization
- Increase in fraction of singleton batches: Works to decrease buffer utilization

We can see, then, that as window size decreases, arrival rate to a node works to increase buffer utilization at the node, whereas round-trip time from the node works to decrease buffer utilization at the node, and if the minimum batch size is one, the fraction of singleton batches at the node works to decrease buffer utilization at the node. This implies that for an appropriate configuration of the parameters, decombining buffer utilization may decrease non-monotonically as window size is reduced.

We may illustrate these relationships by representing them algebraically, as follows, for a complete tree. First note that in a complete tree, a node at distance i from the processors, where $i = 1,\ldots,\log_k n$, is at level $\bar{i} = (\log_k n) - i$ in the tree. Now let $t_{\text{interlevel},\,i}$ be the mean round-trip transit time from (exclusive) a node at distance i from the processors to (inclusive) a node at distance $i+1$ from the processors. Let $t_{\text{nonwindow},\,i}$ be the part of $t_{\text{interlevel},\,i}$ that excludes combining window sojourn time. Then $t_{\text{interlevel},\,i} \le \tau_{\text{indep}} + t_{\text{nonwindow},\,i}$ since τ_{indep} is an upper bound on mean window sojourn time. Also let t_{mem} be the mean round-trip transit time from the root to memory.

Finally, let us take β_i^* from (2.7) and let $p_i = P\{ B_i > 1 \}$. Then for a complete tree in which window size is an independent variable and is the same for all nodes, we have (5.1) as an upper bound on mean decombining buffer utilization of a node at distance i from the processors:

$$u_i \quad \equiv \quad u_i(\tau_{\text{indep}}, \tau_{\text{dep},\,i-1}, \tau_{\text{dep},\,i}) \quad =$$

$$\beta_i^* \;+\; [\, p_i\, \beta_i^* / \tau_{\text{dep},\,i}\,]\,[\, \sum_{j=1}^{\bar{i}} (\tau_{\text{indep}} + t_{\text{nonwindow},\,j}) + t_{\text{mem}}\,] \tag{5.1}$$

Now the left-hand term of u_i is the mean batch size of a request whose batch enters the decombining buffer, and the right-hand term is (an upper bound on) the mean number of requests that enter the decombining buffer while the first request makes a round trip. Thus, the right-hand term is the product of the arrival rate of requests to the decombining buffer times (an upper bound on) the mean round-trip time.

Since there are $k^{\bar{i}}$ nodes at level $\bar{i}$ of a complete tree, the mean decombining buffer utilization of the entire tree is the sum of the u_i over these nodes. These observations give us (5.2) as an upper bound on the mean decombining buffer utilization of a complete tree in which window size is an independent variable and is the same for all nodes:

$$u_{\text{total}} \equiv u_{\text{total}}(\tau_{\text{indep}}) = \sum_{i=1}^{\log_k n} k^{\bar{i}}\, u_i(\tau_{\text{indep}}, \tau_{\text{dep},\,i-1}, \tau_{\text{dep},\,i}) \tag{5.2}$$

Notice that the non-monotonicity of (5.2), as a function of τ_{indep}, is due to the decombining buffer arrival rate from (5.1):

$$p_i\ \beta_i^* / \tau_{\text{dep},\,i} \tag{5.3}$$

Let us analyze the behavior of this term. Using (2.7), we can replace β_i^* by β_i in (5.3), then using (2.8), we can eliminate β_i, giving us the following reexpression of (5.3):

$$k / \tau_{\text{dep},\,i-1} \ - \ P\{\,B_i = 1\,\} / \tau_{\text{dep},\,i} \tag{5.4}$$

The left-hand term of (5.4) decreases in τ_{indep} since $\tau_{\text{dep}} \equiv \tau_{\text{dep}}(\tau_{\text{indep}})$ increases in τ_{indep} (Figure 2.4). Consequently, as long as the minimum batch size is greater than one, the right-hand term of (5.4) will be zero, making (5.4) a decreasing function of τ_{indep}. Thus, as τ_{indep} decreases and the minimum batch size remains greater than one, the decombining buffer arrival rate will increase. However, as τ_{indep} decreases sufficiently that the minimum batch size reaches one, the right-hand term of (5.4) will become non-zero and will begin working to decrease the buffer arrival rate. We know that as $\tau_{\text{indep}} \to 0$, the decombining buffer arrival rate becomes zero as the fraction of singleton batches becomes one: as $\tau_{\text{indep}} \to 0$, $P\{\,B_i = 1\,\} \to 1$, $\tau_{\text{dep},\,i}(\tau_{\text{indep}}) \to \tau_{\text{dep},\,i}(0) = \tau_{\text{dep},\,i-1}(0)/k$.

In conclusion, we have shown that decombining buffer arrival rate will decrease non-monotonically as τ_{indep} decreases if $P\{\,B_i = 1\,\} = 0$ for the initial values of τ_{indep}. In fact, even with $P\{\,B_i = 1\,\} > 0$, (5.4) will still increase if, and while, the increase in the left-hand term exceeds the increase in the right-hand term.

Finally, the non-monotonicity of the terms (5.3), for $i = 1,\ldots,\log_k n$, may translate into non-monotonicity of u_{total} for an appropriate configuration of parameters, giving us a non-monotonic decrease in decombining buffer utilization as window size is reduced. In particular, we can see that increasing the network inter-unit delay (Figure 5.4d) will reduce the fraction of round-trip time that consists of window sojourn time by increasing the size of $t_{\text{nonwindow},\,j}$ and t_{mem} relative to τ_{indep} in (5.1). This will mitigate the effect that a reduction in window size will have in reducing round-trip time, thereby making the non-monotonic effect of (5.3) more pronounced.

5.3. Data

Steady state. Our first concern is to show that an iteration was run long enough to reach steady state, that is, to reach a state where the distribution of a dependent variable changes very little with time. For convenience and practicality, we focus on checking the stability of the mean of the distribution. To demonstrate steady state, Figure 5.5 shows mean memory queue length as a function of simulated time for four combinations of window discipline and compute-time distribution under the case where $n_{\text{max.reqs.out}} = \infty$. The maximum number of outstanding requests $n_{\text{max.reqs.out}}$ was chosen to be infinite for this purpose because under this condition the iteration will take the longest to reach steady state. Memory queue length is chosen because when $n_{\text{max.reqs.out}} = \infty$, memory queue length is the primary determinant of differences in transit time among the window disciplines, hence memory queue length is the primary indicator of differences in performance among the window disciplines.

In addition, we know what the population distributions are for processor inter-request time when $n_{\text{max.reqs.out}} = \infty$, so we took an iteration under this configuration to a point where, for the distribution uniform(0,2), the sample mean came to within ±.001 of the population mean of one. Based on running the random number generator independently for a million draws, we could see that ±.001 was as close as we

Figure 5.4 (continued). Simulation Experiment Variables.

Figure 5.4d. Independent Variables for Experiment on Window Size.

Note: These variables are defined in Figure 5.2.

- Max # outstanding requests $n_{\text{max.reqs.out}}$: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, infinite
- Compute-time distribution: Uniform(0,2) ($r = 1$, variance > 0)
- Window discipline: Constant window size of $\tau = 4, 2, 1, .75, .5, .25$
- Memory update time: .25 ($\mu_{\text{mem}} = 4$)
- Network inter-unit delay: four (increases round-trip time by 24)

could practically ask to come to honing in on the population mean of one. Then we took all other configurations to the same point.

Window discipline. We now wish to compare the four combinations of window discipline and compute-time distribution in terms of the means and standard deviations of the dependent variables as a function of the maximum number of requests allowed. Recall that the dependent variables are given in Figure 5.4a, and the independent variables for the window-discipline experiment are given in Figure 5.4b. The results are shown in Figures 5.6-5.8, where the legends mean the following:

- Window discipline
  - Batch 2: Constant batch size of two ($\beta = 2$)
  - Window 1: Constant window size of one ($\tau = 1$)
- Compute-time distribution
  - Constant 1: Constant one ($r = 1$, variance = 0)
  - Uniform 02: Uniform(0,2) ($r = 1$, variance > 0)

As mentioned, processor inter-request time has the compute-time distribution mean and standard deviation where the maximum number of outstanding requests is sufficiently large that it does not limit the processor request rate. Figure 5.6 shows the effect that constant window size has at level 1 in smoothing out the lulls and bursts generated by the uniform(0,2) distribution: constant window size inter-request time has half the standard deviation of constant batch size inter-request time. The analogous effect at level 0, which feeds memory directly, is shown in Figure 5.7, where the effect is even greater, reducing the standard deviation by a factor of six. This difference in variance translates into a dramatic difference in memory queue length, as seen in Figure 5.8. Memory queue utilization is added to decombining buffer utilization in the figure because the sum of the two contribute to the total system storage requirements. However, a separate graph of memory queue utilization (not shown) indicates that memory queue utilization for the lower three conditions in Figure 5.8 is one and that the dramatic effect shown is due to memory queue utilization. Keep in mind that memory queue utilization affects round-trip time and thus decombining buffer utilization, hence a larger memory queue utilization is compounded by a larger decombining buffer utilization.

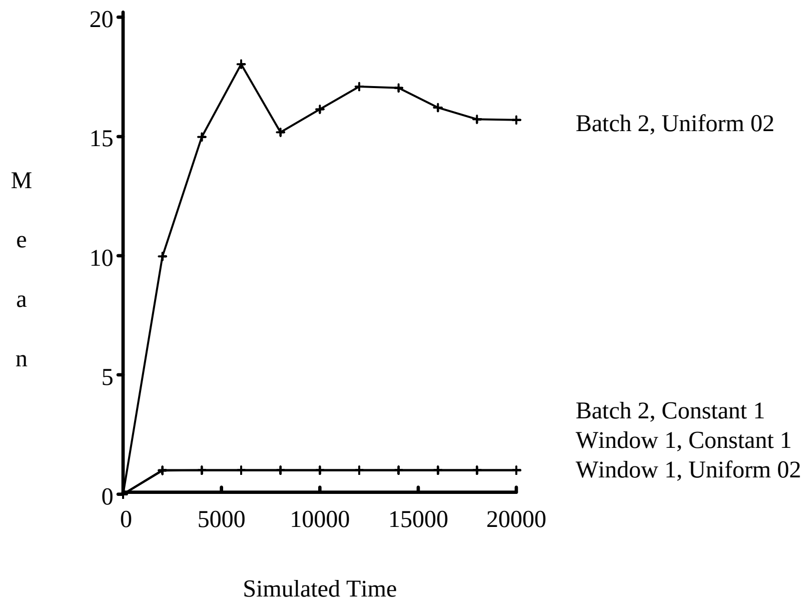


Figure 5.5. Steady State Indicator: Memory Queue Length.

Note: $n_{\text{max.reqs.out}} = \infty$.

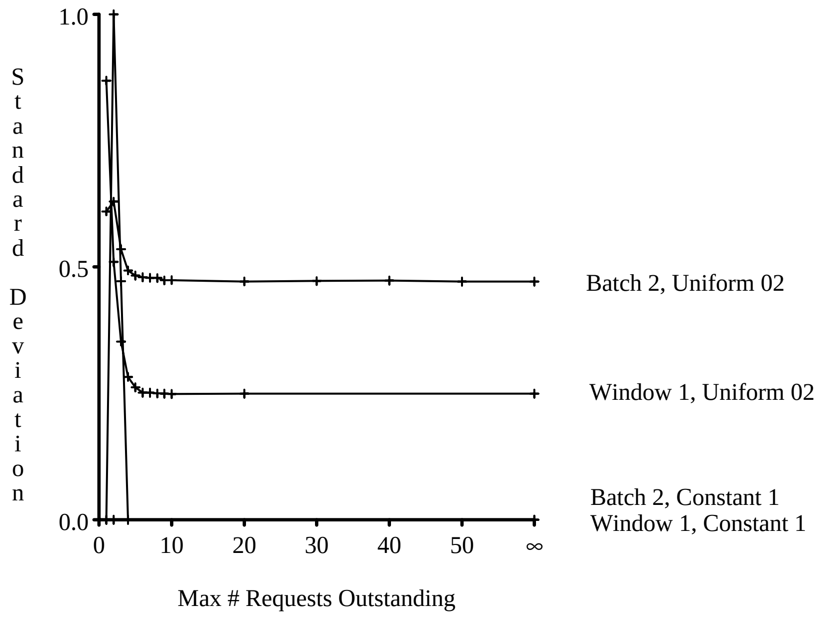


Figure 5.6. Level 1 Combining Buffer Inter-Request Time.

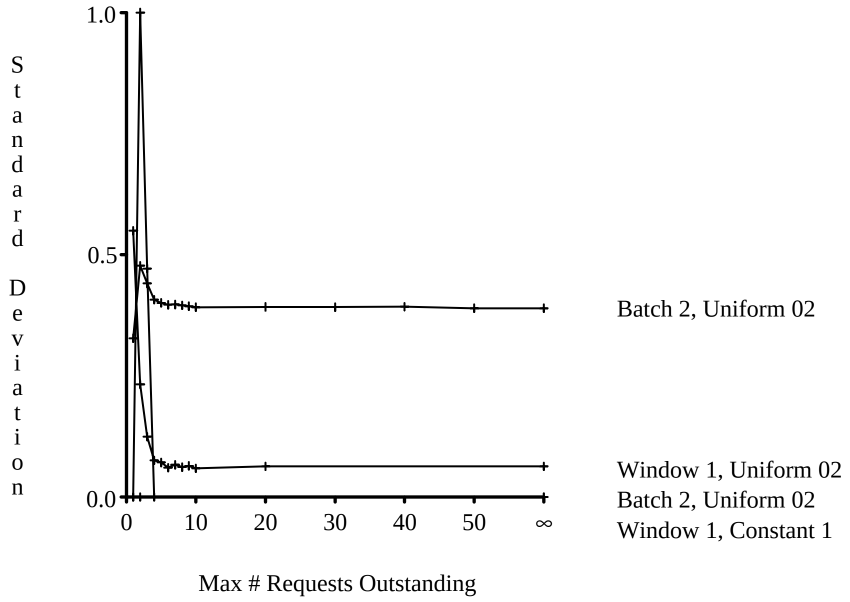


Figure 5.7. Level 0 Combining Buffer Inter-Request Time.

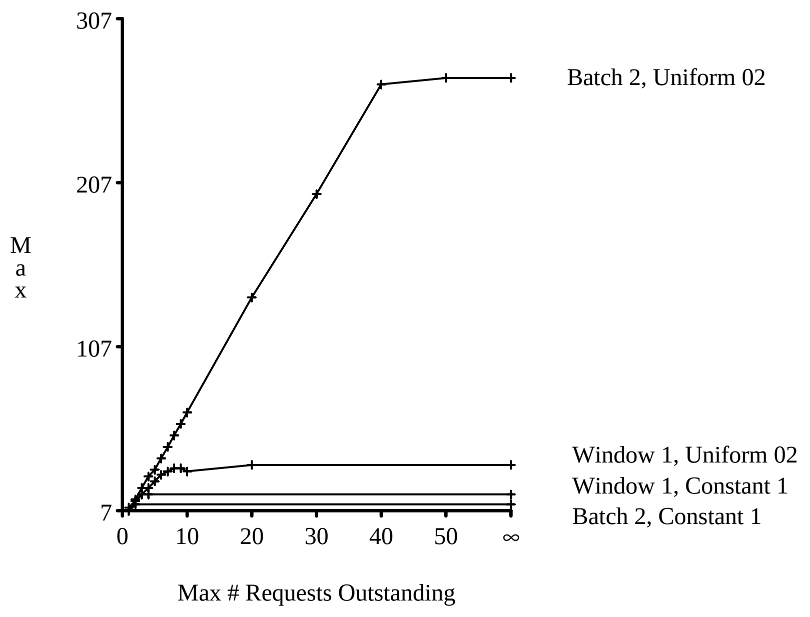


Figure 5.8. Memory Queue Plus Decombining Buffer Utilization.

Turning to Figure 5.9, let us analyze the cross overs that appear in hypothetical mean execution time. To obtain perspective for interpreting these results, let us reason through what would occur if there were *no combining*, i.e., if window size were zero here. If there were no combining, with no limit on the number of outstanding requests, the arrival rate at memory would be four times the memory service rate, hence execution time for 10,000 replies per processor would be the time that it takes 40,000 requests to be serviced by memory, viz., 40,000 time units, plus a little extra for the time it takes to saturate the memory queue and make a round trip. Furthermore, if the limit on the number of outstanding requests were decreased, memory would remain the bottleneck, so its processing rate would still determine performance within a small additive term per condition. Keep in mind that memory would serialize the processor requests and hence stagger the replies, so once the processors exhausted their maximum number of outstanding requests, their compute periods after the first would become staggered.

We see, then, that an execution time of about 40,000 is the worst we can do, and that combining helps except in one case, as seen in Figure 5.9. Under constant compute time with only one outstanding request allowed per processor, constant window size does not help, whereas constant batch size reduces hypothetical mean execution time from 40,000 to 20,000. This difference is an artifact of the fact that under constant compute time, processor siblings are in lock step, thus once one request arrives in a combining window, the other request has also arrived, so further waiting needlessly delays the transit. The hypothetical mean execution time of 20,000 for constant batch size is due to the two one-unit delays of compute time and memory service time, times 10,000 replies per processor.

The uniform(0,2) compute-time distribution yields more interesting results. In this case, with only one outstanding request allowed per processor, constant window size is worse than constant batch size: constant window size produces a hypothetical mean execution time of 34,000 while constant batch size produces a

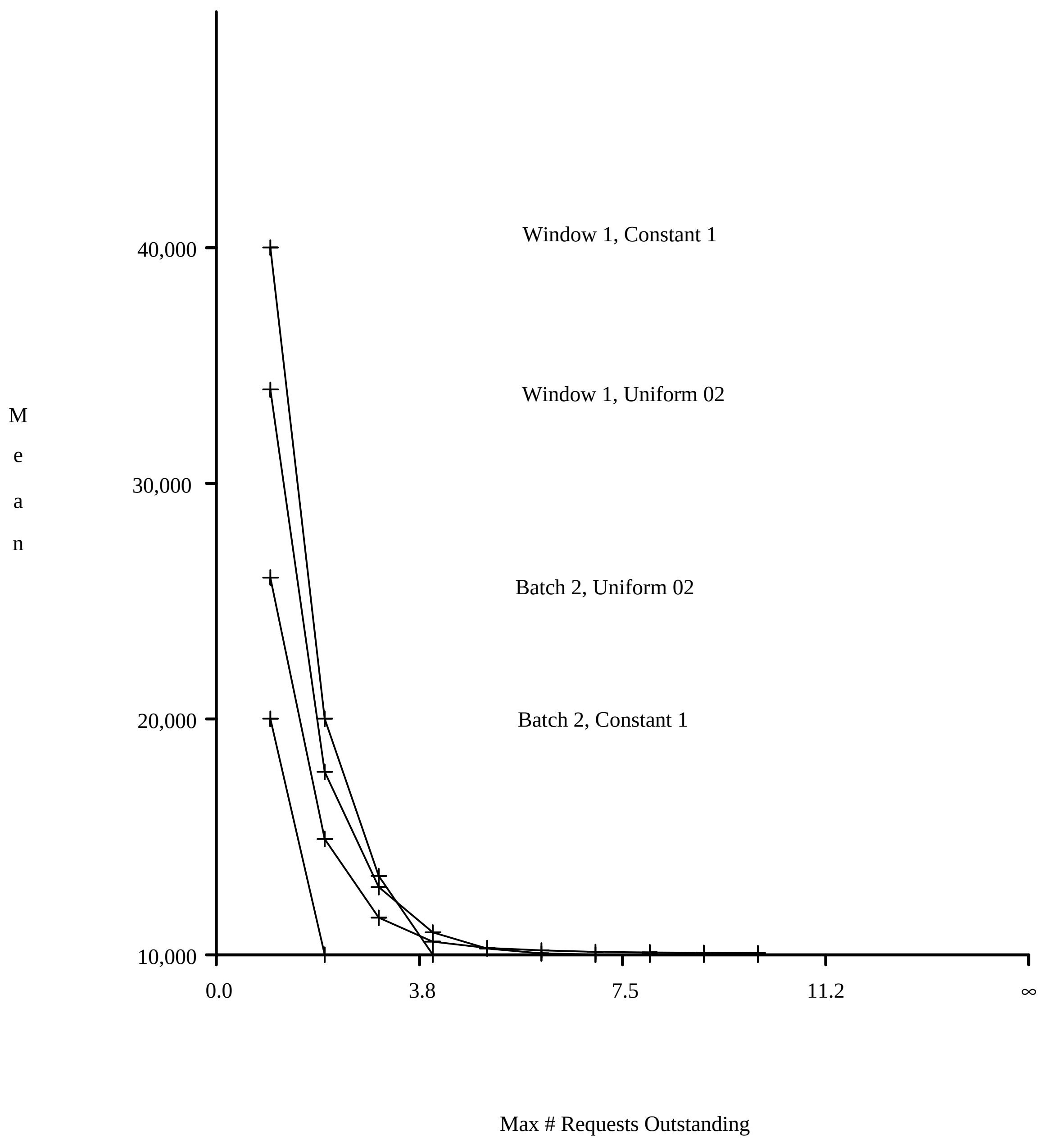


Figure 5.9. Hypothetical Execution Time.

hypothetical mean execution time of 26,000. Why the difference? Other data shows that batch sizes at both levels 1 and 0 are nearly one for constant window size, indicating that little combining is occurring. Yet constant window size forces a longer transit time from level 1, and thus from the processor. Hence, the window at level 1 lengthens the transit time while achieving little combining. Consequently, under constant window size, a shorter window would be better in this configuration. The reason is that with only one outstanding request per processor, once a processor issues a request, it will stall until it it receives the respective reply, so it is important to return the reply as soon as possible while still ensuring a sufficient amount of combining.

Hypothetical mean execution time shows, then, that as the limit on the number of outstanding requests is increased, constant window size soon outperforms constant batch size because of the reduction in inter-request-time variance that constant window size produces. However, the magnitude of the execution-time advantage is not great in our configuration. The primary advantage of constant window size, here, is reduced decombining buffer + memory queue utilization.

Combining *vs.* **non-combining.** In the second experiment, we compare the effect of combining, under constant window size, with non-combining when total processor request rate λ is the same as the memory service rate μ_{mem}. The results are shown in Figures 5.10-5.12.

Overall, the comparison of non-combining with combining under constant window size is similar to the comparison of constant batch size with constant window size from the first experiment, where all conditions have the compute-time distribution uniform(0,2). However, the effect on memory queue length is not evident from the combining buffer inter-request-time standard deviations, so we turn to the distribution of level 0 inter-request time, shown in Figure 5.11. Here we see that 60% of the inter-request times to memory under non-combining are less than .25. Thus, combining, under constant window size, eliminates bursts that outpace memory. And, of course, it smoothes out the bursts by combining incoming requests, rather than queuing them, thereby giving combining the advantage over non-combining.

Finally, combining, under constant window size, improves performance, but at what cost? The decombining buffer cost plus the memory queue cost must be compared for non-combining and combining. Figure 5.12 reveals that this combined cost becomes greater under non-combining than under combining as $n_{\text{max.reqs.out}}$ increases sufficiently. Keep in mind that for Figure 5.12 we can deduce that decombining buffer utilization is zero for $\tau = 0$, and that memory queue utilization is one or less for $\tau \geq 1/\mu_{\text{mem}} = .25$.

Window size. Let us move on to explore the effect of reducing window size τ below processor mean inter-request time $1/r$. The configuration for this experiment is the same as that for the second experiment except that the network inter-unit delay is raised to a positive value in order to make the non-monotonic behavior of decombining buffer utilization more pronounced. In addition, we do not take τ down to zero.

Why do we wish to reduce window size below $1/r$? The primary reason for reducing window size below $1/r$ is to come closer to satisfying (2.15) for small values of $n_{\text{max.reqs.out}}$ by reducing round-trip time $T_{\text{trip},\,i}$. A secondary reason is that even if (2.15) is already satisfied, if $r_{\min} \leq r < r_{\max}$, then program execution rate could be sped up by increasing the processor request rate, taking r closer to $r_{\max}$: $r_{\min} < r \leq r_{\max}$. We wish to observe the cost in decombining buffer utilization of reducing window size.

Figure 5.13 shows the reduction in hypothetical mean execution time that is obtained by reducing window size. For small values of $n_{\text{max.reqs.out}}$, this reduction is relatively small but noticeable, while it vanishes for sufficiently large values of $n_{\text{max.reqs.out}}$. This is due to reduction in mean round-trip time, which can be calculated directly from the window size and memory service time, since memory queue length will be negligible, as it was in the second experiment.

Turning to the issue of decombining buffer utilization, we see from Figure 5.14 that for sufficiently large values of $n_{\text{max.reqs.out}}$, total utilization decreases monotonically as window size decreases. In contrast, for moderate and small values of $n_{\text{max.reqs.out}}$, we obtain the non-monotonic effect referred to in Section 5.2: as

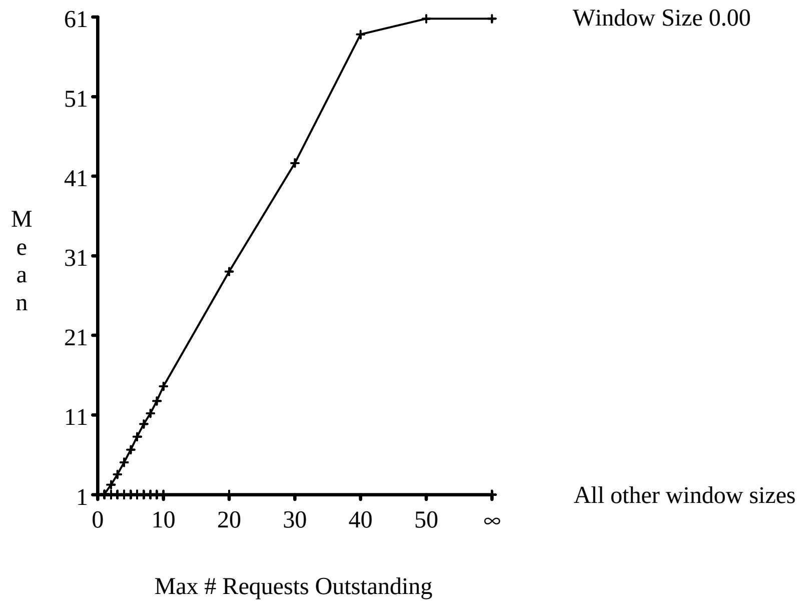


Figure 5.10. Memory Queue Length.

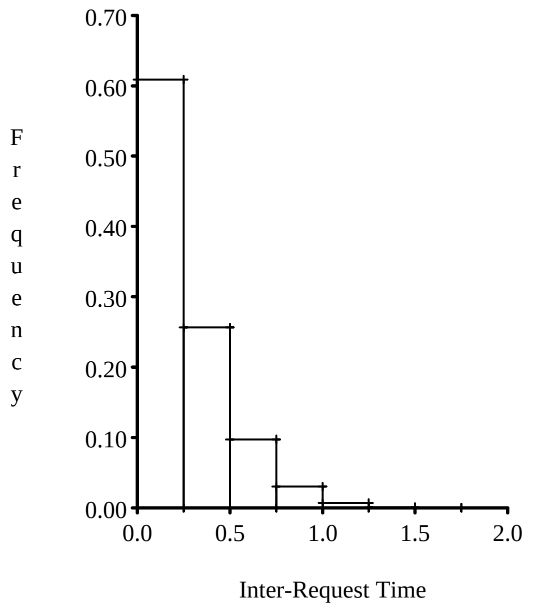


Figure 5.11. Level 0 Inter-Request Time to Memory.

Note: The configuration is $\tau = 0$, $n_{\text{max.reqs.out}} = \infty$. The interval endpoint modes are [xLo,xHi).

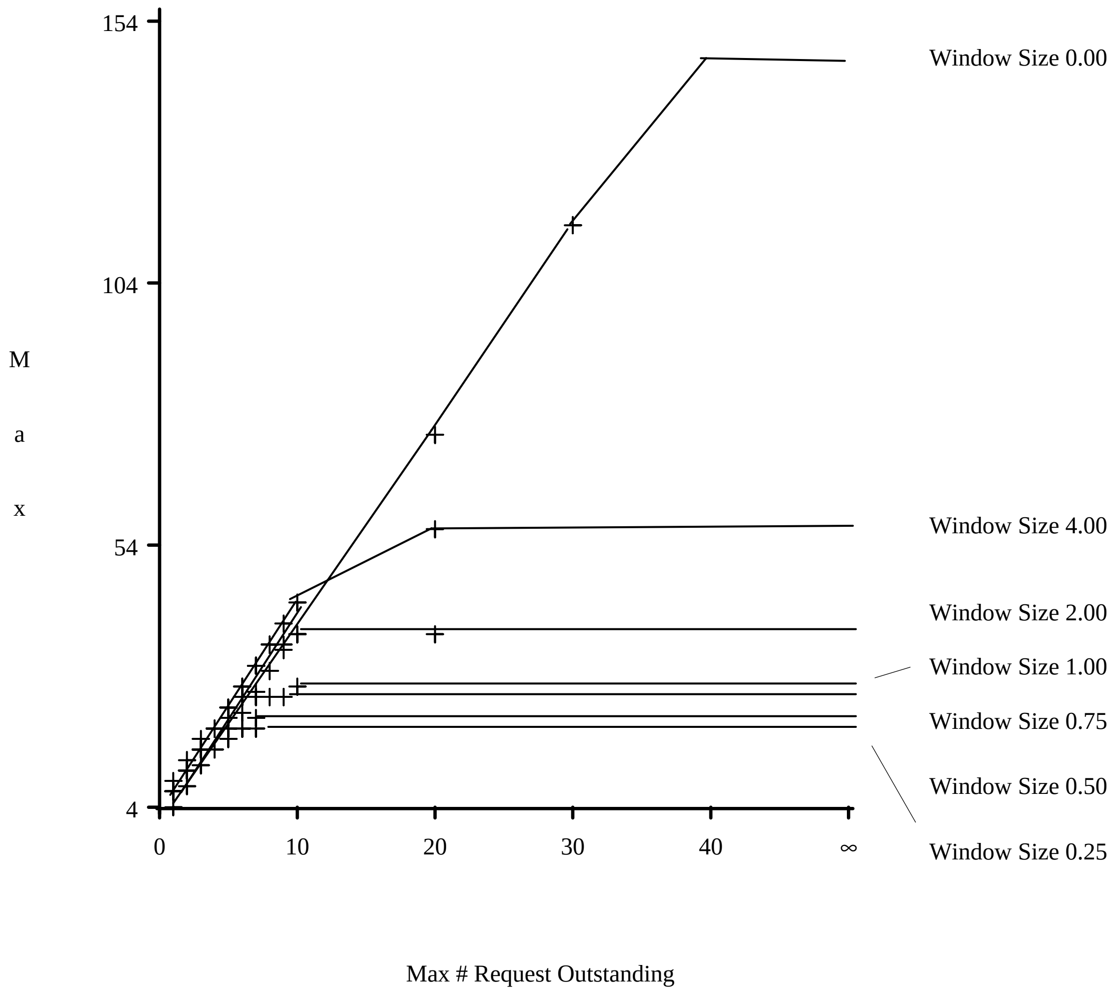


Figure 5.12. Memory Queue Plus Decombining Buffer Utilization.

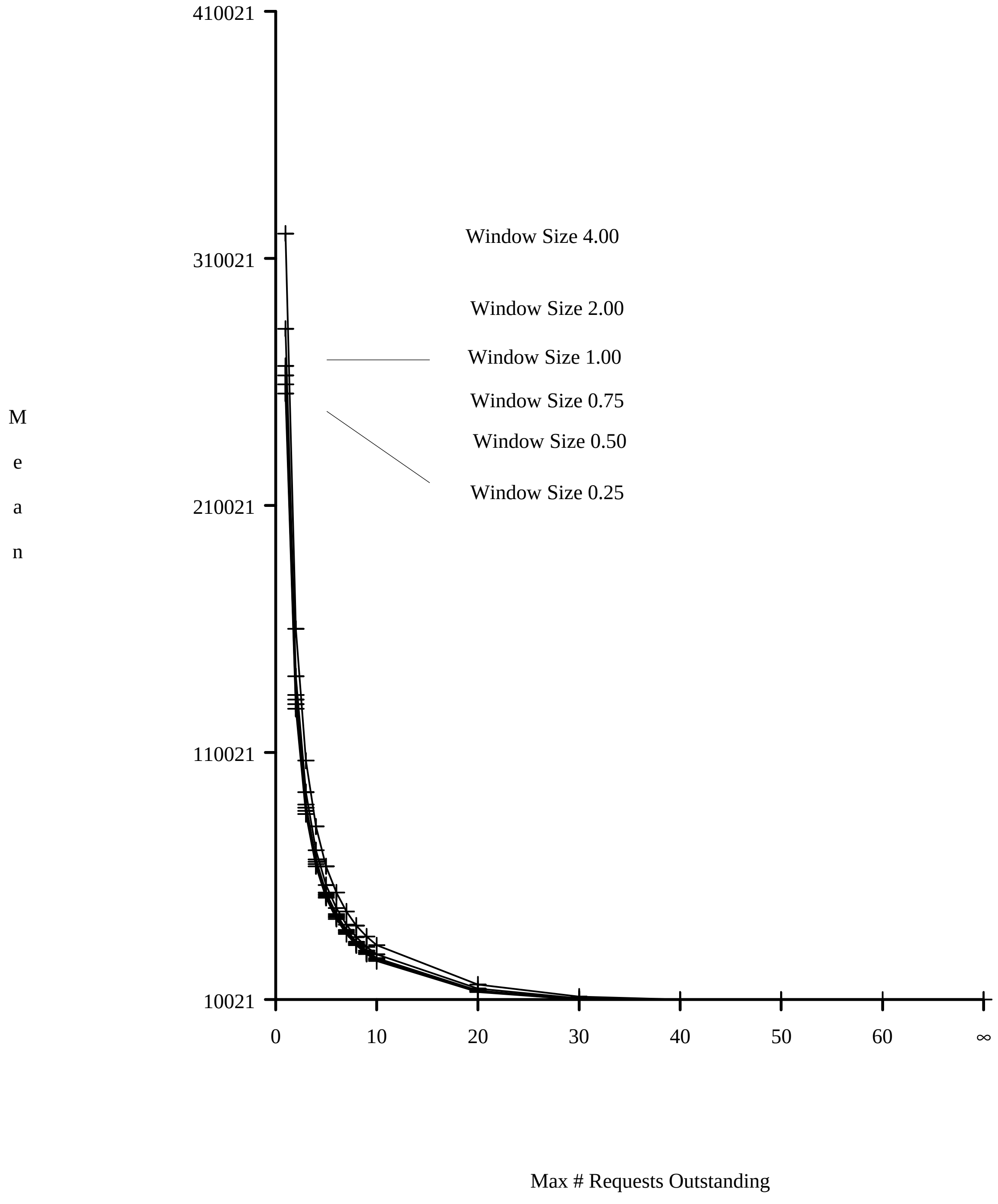


Figure 5.13. Hypothetical Execution Time.

window size decreases from four to two, buffer utilization increases, but as window size continues to decrease to one, buffer utilization starts decreasing. What is the cause of the non-monotonicity? Let us consider the fraction of batches that are singletons. Singletons are not stored in the decombining buffer, so an increase in this fraction, due to reduced window size, will work toward reducing decombining buffer utilization, as seen in the right-hand term of (5.4). In dropping from $\tau = 4$ to $\tau = 2$, we may expect to find little or no increase in the fraction of singleton batches because window size is so large. However, Figures 5.15 and 5.16 show that mean batch size decreases monotonically in window size except for one case: level 0 with $n_{\text{max.reqs.out}} = 10$. In fact, for small values of $n_{\text{max.reqs.out}}$ at level 0, the drop in mean batch size from $\tau = 4$ to $\tau = 2$ is very large and takes mean batch size close to one. In addition, for all values of $n_{\text{max.reqs.out}}$ at level 1, this drop is very large. The frequency distributions are not shown, but they indicate the same direction in the frequency of a singleton batch. Consequently, the singleton-batch curve, as a function of window size, does not explain the non-monotonicity of utilization. That is, an increase in the value of the right-hand term of (5.4) from zero to non-zero does not explain the non-monotonicity of (5.4).

We must turn, then, to arrival rate, i.e., the left-hand term of (5.4), to find the explanation. Figures 5.17 and 5.18, respectively, show mean inter-request time from the processors to level 1, and from level 1 to level 0. For the requests from level 1 to level 0, we can see that the drop in mean inter-request time is much greater in going from $\tau = 4$ to $\tau = 2$ than in going from another value of τ to a smaller value. This, then, explains the non-monotonicity: the increase in arrival rate outweighed the increase in the fraction of singleton batches for small and moderate values of τ_{indep}.

In conclusion, we see that the reduction in window size below $1/r$ down to the memory service rate reduces execution time noticeably for small values of $n_{\text{max.reqs.out}}$. In addition, we have observed the non-monotonicity of decombining buffer utilization as a function of window size. However, neither the reduction in execution time nor the temporary increase in decombining buffer utilization is large in this configuration.

5.4. Conclusion

What have we learned from the simulations?

We have developed a simulation model and simulator for running combining tree simulations, and we have begun using this model by executing three experiments of interest. Specifically, we have explored the effect of constant batch size *vs.* constant window size, the effect of combining (under constant window size) *vs.* non-combining when total processor request rate λ_{min} does not exceed the memory service rate μ_{mem}, and the effect of reducing constant window size τ below processor mean inter-request time $1/r_{\text{min}}$. Let us discuss each experiment briefly.

Window discipline. In the first experiment, we saw that for a sufficiently large limit on the number of outstanding requests for a processor, constant window size outperforms constant batch size because it reduces the inter-request time variance to the queues along the pathway to memory, in particular, the inter-request time variance to the memory queue itself. This reduces the memory queue length, and hence the transit time, the decombining buffer utilization, and the hypothetical mean execution time. As the limit on the number of outstanding requests goes to infinity, the advantage in smaller execution time tapers off, while the advantage in smaller memory queue utilization increases dramatically.

Keep in mind that in this experiment there was only one type of memory request, viz., a combine request to a single memory module. More generally, after each of its compute periods a processor will issue a combine request for module 0 with probability p and a non-combine request with probability $\bar{p} = 1 - p$, where a non-combine request is uniformly distributed across the memory modules. This model was conceived by Pfister and Norton (1985), as discussed in Chapter 3. Now for our experiment $p = 1$. But what would happen if we reduced p? As $p \to 0$, $\bar{p} \to 1$, the processor requests become uniformly distributed across the memory modules, and the combining requests to module 0 play a smaller and smaller role in system performance. Thus, as $p \to 0$, the advantage of constant window size over constant batch size will disappear. In

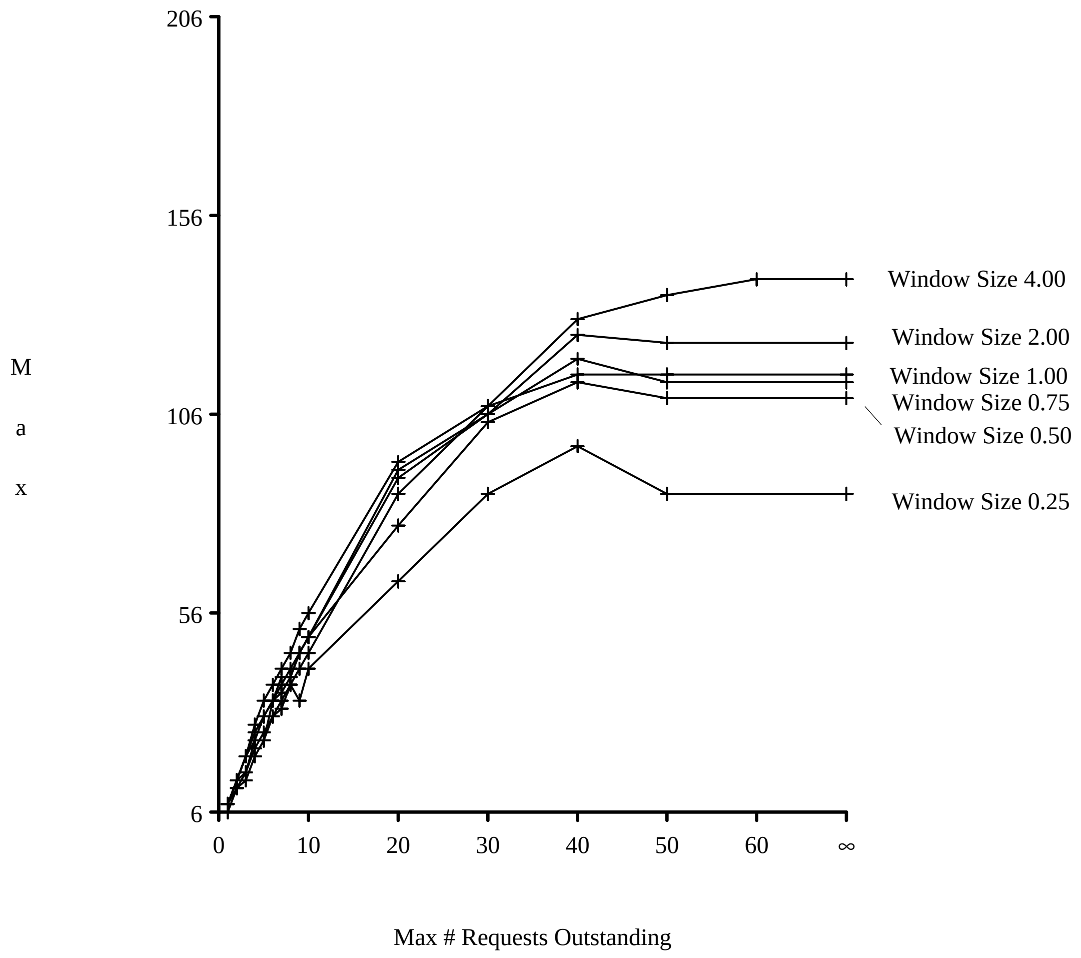


Figure 5.14. Decombining Buffer Utilization.

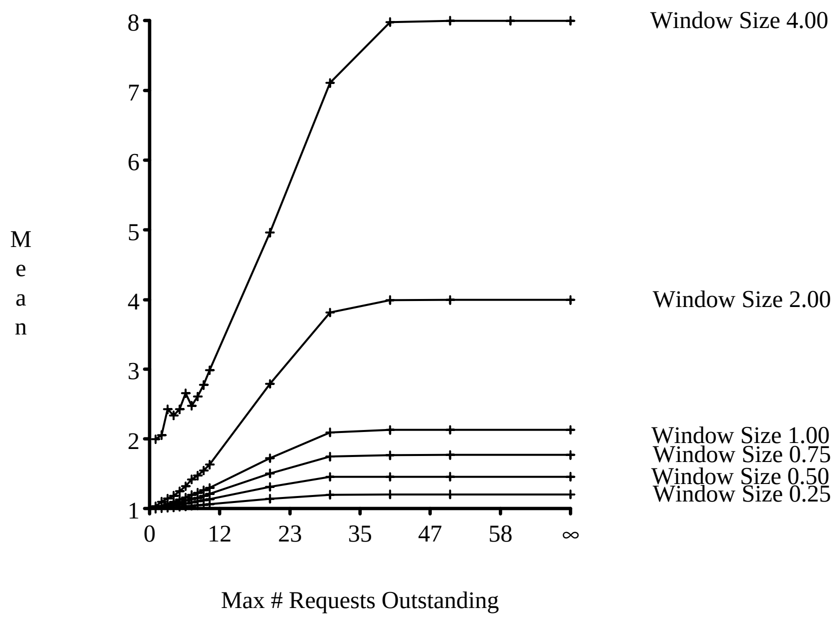


Figure 5.15. Level 1 Batch Size.

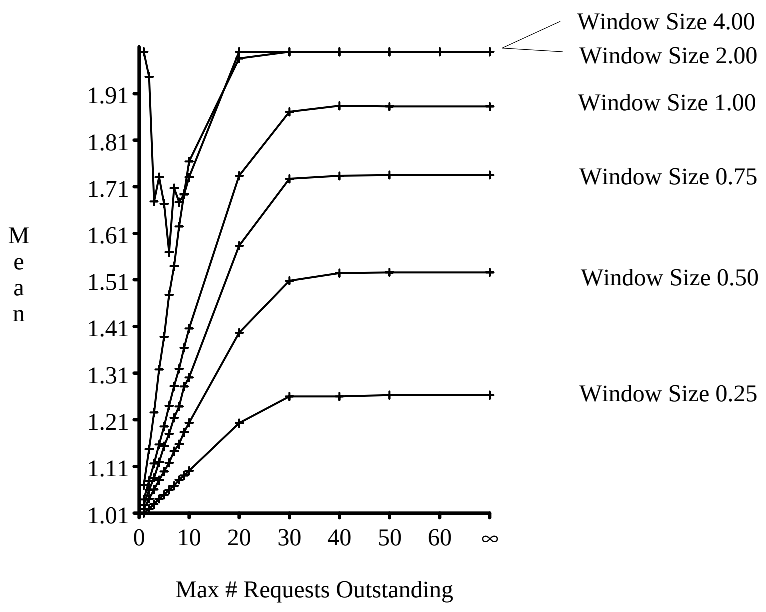


Figure 5.16. Level 0 Batch Size.

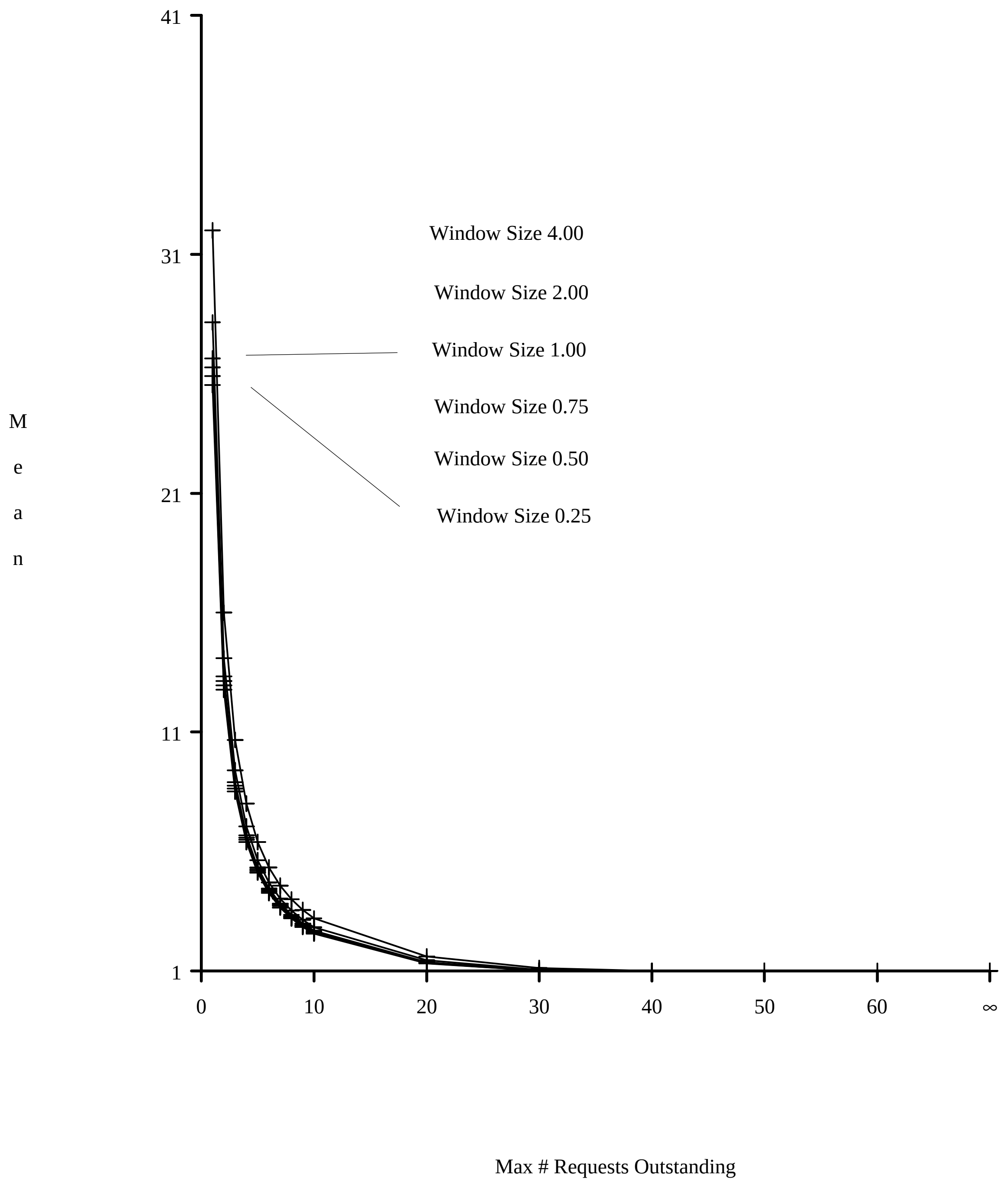


Figure 5.17. Processor Inter-Request Time to Level 1.

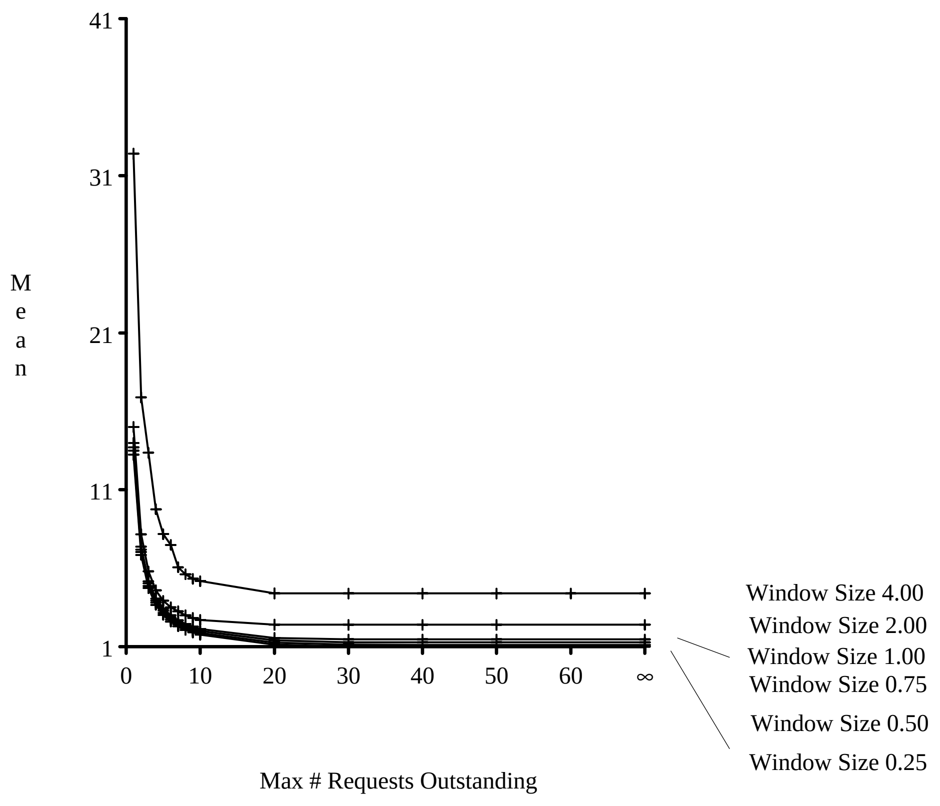


Figure 5.18. Level 1 Inter-Request Time to Level 0.

short, the extent to which constant window size is better than constant batch size in a given system depends on the fraction of memory requests that are combine requests.

Combining *vs.* non-combining. In the second experiment, we examined a configuration in which the total processor request rate λ does not exceed the memory service rate μ_{mem}. In this configuration, we saw that the reduction in inter-request time variance afforded by constant window size in the first experiment has a similar effect when compared to non-combining (constant batch size of one), dramatically reducing the decombining buffer + memory queue length utilization for sufficiently large $n_{max.reqs.out}$.

Window size. Finally, in the third experiment, we used the configuration of the second experiment, except that we substantially decreased the fraction of round-trip time comprised of combining window sojourn time. The result, for small values of $n_{max.reqs.out}$, is a reduction in execution time, but the reduction is not large. We also observed non-monotonic behavior of decombining buffer utilization as window size is reduced.

6. Conclusion

What is the key to managing a combining tree?

We have seen that the key to managing a combining tree is to implement combining windows. This provides both sufficient execution concurrency and sufficient storage concurrency, among the nodes at each level of the tree, which are the dual problems in the management of a combining tree.

Our strategy in this initial study has been to adopt a steady-state queuing model. In this context, our analytic model illustrates the analytic solution strategy. And our simulation experiments, which allow weaker, more realistic assumptions than analytic models allow, show that combining window discipline can have a substantial effect on execution speed and decombining buffer utilization.

What work remains for future research?

The combining-tree paradigm developed here is rich in possibilities for research, using both analytic and simulation models.

Regarding analytic models, the analytic solution strategy may be taken in the directions discussed in the conclusion to Chapter 4, exploring more realistic models than the model presented there by weakening that model's assumptions where it appears promising to do so.

Regarding simulation models, simulation experiments may be used to study the same issues that were raised for analytic models. In addition, simulation experiments could be used to study the design of *hardware combining trees* to determine how best to deal with the inflexibility of hardware in the face of multiple combinable variables, each having its own ideal mean combining window size τ. This would entail studying the cost/performance tradeoffs that arise in designing combining and decombining operations and buffers. Simulation experiments could also be used to study *software combining trees* to determine how best to utilize the flexibility they offer in mapping abstract combining nodes to hardware nodes. In particular, the tradeoff between tree depth and system balance could be explored.

Simulation could also be used to explore the procedure that a process, or processor, may use to assess its request rate r to a combining tree, upon which a time-based window discipline depends. This would entail a procedure for making an initial estimate — possibly based on prior execution of the program — along with a procedure for tuning the estimate. Further, the cost and performance advantage could be examined of estimating the value of τ_{indep} that will yield the desired value of τ_{dep}, say $1/r$.

Finally, another question that could be explored further, both analytically and experimentally, is the analysis of transient behavior. The model presented in this dissertation is a steady-state model, and we have shown how to evaluate window discipline using constant window parameters based on steady-state means. We showed that for a sufficiently large limit on the number of outstanding requests, constant window size is better than constant batch size because it smooths out the lulls and bursts in the processor request stream, thereby reducing execution time and decombining buffer utilization. Thus, the discipline of constant window size has built into it a mechanism for dealing with transient behavior.

Yet, suppose that requests to some combining tree can be usefully modeled, not only as a steady-state stream of individual requests, but also as a steady-state stream of bursts. That is, bursts occur at stochastically-determined intervals, such as at the end of a computation phase like that defined by a **doall** loop. Then k-fold combining could be obtained during a burst by using a mean window size that is shorter than overall mean processor inter-request time $1/r$, which incorporates the mean inter-burst time. To optimize the handling of this case, we need to model the intra-burst request rate and specify the intra-burst window size accordingly.

7. References

Almasi, G.S., Gottlieb, A. 1989. ‘‘The NYU Ultracomputer.’’ Section 10.3.6 in *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, CA, 1989, 430-450.

Bitar, P. 1990a. ‘‘Combining windows: The key to managing MIMD combining trees.’’ Presented at architecture workshop 1990. Published in Dubois, M., Thakkar, S. (Eds.), *Scalable Shared Memory Multiprocessors,* Kluwer Academic Publishers, Norwell, Mass., 1992.

Bitar, P. 1990b. ‘‘MIMD Synchronization and Coherence.’’ November 1990, Version 92/03/26. Tech. Report UCB/CSD 90/605, Computer Science Division, U.C. Berkeley, Berkeley, CA 94720.

Bitar, P. 1992. ‘‘The weakest memory-access order.’’ *J. of Parallel & Distributed Computing,* 15, August 1992, 305-331.

Cheong, H.C., Veidenbaum, A.V. 1990. ‘‘Compiler-directed cache management in multiprocessors.’’ *Computer,* 23(6), June 1990, 39-47.

Dias, D.M., Kumar, M. 1989. ‘‘Preventing congestion in multistage networks in the presence of hotspots.’’ *18th Int’l Conf. on Parallel Processing,* 1989, I-9 - I-13.

Dickey, S.R., Percus, O.E. 1992. ‘‘Performance differences among combining switch architectures.’’ *21st Int’l Conf. on Parallel Processing,* 1992.

Goodman et al. 1989. Goodman, J.R., Vernon, M.K., Woest, P.J. 1989. ‘‘Efficient synchronization primitives for large-scale cache-coherent multiprocessors.’’ *3rd Int’l. Conf. on Architectural Support for Prog. Lang. & Op. Sys.*, 1989, 64-75.

Gottlieb, A. 1987. ‘‘An overview of the NYU Ultracomputer project.’’ In Dongarra, J.J. (Ed.), *Experimental Parallel Computing Architectures,* North-Holland / Elsevier Science Publishers, New York, NY, 1987.

Gottlieb et al. 1983a. Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L, Snir, M. ‘‘The NYU Ultracomputer — designing an MIMD shared memory parallel computer.’’ *IEEE Trans. Computers*, C-32(2), February 1983, 175-189.

Gottlieb et al. 1983b. Gottlieb, A., Lubachevsky, J., Rudolph, L. ‘‘Basic techniques for the efficient coordination of very large numbers of cooperating sequential processes.’’ *ACM Trans. Prog. Lang. & Sys.*, 5(2), April 1983, 164-189.

Ho, W.S., Eager, D.L. 1989. ‘‘A novel strategy for controlling hot spot congestion.’’ *18th Int’l Conf. on Parallel Processing,* 1989, I-14 - I-18.

James et al. 1990. James, D.V., Laundrie, A.T., Gjessing, S., Sohi, G.S. ‘‘Scalable Coherent Interface.’’ *Computer,* 23(6), June 1990, 74-77.

Kang et al. 1991. Kang, B.-C., Lee, G., Kain, R. ‘‘Performance of multistage combining networks.’’ *20th Int’l Conf. on Parallel Processing,* 1991, I-550 - I-553.

Lee et al. 1986. Lee, G., Kruskal, C.P., Kuck, D.J. ‘‘The effectiveness of combining in shared memory parallel computers in the presence of hot spots.’’ *15th Int’l Conf. on Parallel Processing,* 1986, 35-41.

Lee, G. 1989. ‘‘A performance bound of multistage combining networks.’’ *IEEE Trans. Computers,* C-38(10), October 1989, 1387-1395.

Merchant, A. 1992. ‘‘Analytical models of combining banyan networks.’’ SIGMETRICS 1992. Proceedings are published as *Performance Evaluation Review*, 20(1), June 1992, 205-212.

Park, S.K., Miller, K.W. 1988. ‘‘Random number generators: good ones are hard to find.’’ *Comm. of the ACM.* 31(10), October 1988, 1192-1201.

Pfister et al. 1985. ‘‘The IBM research parallel processor prototype (RP3): Introduction and architecture.’’ *14th Int’l Conf. on Parallel Processing,* 1985, 764-771.

Pfister, G.F., Norton, V.A. 1985. ‘‘Hot spot contention and combining in multistage interconnection networks.’’ *IEEE Trans. Computers,* C-34(10), October 1985, 943-948.

Ranade, A.G. 1989. ‘‘Fluent Parallel Computation.’’ Ph.D. dissertation, May 1989. TR 663, CS Dept., Yale U., 10 Hillhouse Avenue, New Haven, CT 06511.

Scott, S.L., Sohi, G.S. 1990. ‘‘The use of feedback in multiprocessors and its application to tree saturation control. *IEEE Trans. Parallel & Distributed Computing,* 1(4), October 1990, 385-398.

Thomas, R.H. 1986. ‘‘Behavior of the butterfly parallel processor in the presence of memory hot spots.’’ *15th Int’l Conf. on Parallel Processing,* 1986, 46-50.

Wolff, R.W. 1989. *Stochastic Models and the Theory of Queues.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

Yew et al. 1987. Yew, P.-C., Tzeng, N.-F., Lawrie, D.H. ‘‘Distributing hot-spot addressing in large-scale multiprocessors.’’ *IEEE Trans. Computers,* C-36(4), April 1987, 388-395.

8. Appendix: Fetch-and-Add

Sleep-wait queuing using fetch-and-add. Here we present algorithms for sleep-wait queuing, extending the discussion of fetch-and-add algorithms presented in Chapter 2.

Figure 8.1. Sleep-Wait P with FIFO Enqueue Using Fetch-and-Add

```
P(input: adr_entry; output: sleep_flag ):

global variable:  Q_size, sem, I, Q[Q_size], Next[Q_size], Sem[Q_size];
local, register variable:  my_I, cell, ticket;

begin
    sleep_flag := 1;
    if fetch-and-add(sem, -1) ≤ 0 then              /* may need to enqueue entry */
    begin
        my_I := fetch-and-add(I, 1);
        /* now (in effect) divide my_I by Q_size */
        /* then take remainder to get cell, or truncate to get ticket */
        cell := remainder(my_I/Q_size);
        ticket := floor(my_I/Q_size);               /* enqueue, dequeue use same ticket */
        while Next[cell] ≠ ticket do null;          /* busy-wait for turn */
        Q[cell] := adr_entry;
        if fetch-and-add(Sem[cell], -1) = 1 then    /* V has already arrived */
        begin
            sleep_flag := 0;
            fetch-and-add(Next[cell], 1);
        end;
    end;
end;
```

Notes for Figures 8.1, 8.2

1. The variable *sem* is the semaphore on which the P and V are operating.
2. Each cell of the array *Sem* is initialized to zero and ends up back at zero at the end of each turn.
3. If *sleep_flag* is returned as 1, the process was enqueued and must sleep. If *wake_flag* is returned as 1, a process was dequeued and must be awakened.

Figure 8.2. Sleep-Wait V with FIFO Dequeue Using Fetch-and-Add

```
V(output: adr_entry, wakeup_flag ):

global variable:  Q_size, sem, D, Q[Q_size], Next[Q_size], Sem[Q_size];
local, register variable:  my_D, cell, ticket;

begin
    wakeup_flag := 0;
    if fetch-and-add(sem, 1) < 0 then                /* may need to dequeue entry */
    begin
        my_D := fetch-and-add(D, 1);
        /* now (in effect) divide my_D by Q_size */
        /* then take remainder to get cell, or truncate to get ticket */
        cell := remainder(my_D/Q_size);
        ticket := floor(my_D/Q_size);                /* enqueue, dequeue use same ticket */
        while Next[cell] ≠ ticket do null;           /* busy-wait for turn */
        if fetch-and-add(Sem[cell], 1) = -1 then     /* P has already arrived */
        begin
            adr_entry := Q[cell];
            wakeup_flag := 1;
            fetch-and-add(Next[cell], 1);
        end;
    end;
end;
```

Notes for Figures 8.1, 8.2

1. The variable *sem* is the semaphore on which the P and V are operating.
2. Each cell of the array *Sem* is initialized to zero and ends up back at zero at the end of each turn.
3. If *sleep_flag* is returned as 1, the process was enqueued and must sleep. If *wake_flag* is returned as 1, a process was dequeued and must be awakened.

Sleep-wait queuing using efficient busy-wait locking/waiting/unlocking. For comparison with fetch-and-add algorithms, we present sleep-wait queuing algorithms based on efficient busy-wait locking/waiting/unlocking, as discussed in Bitar (1990b). The strategy in these algorithms is to minimize lock hold-time, and to lock *head* and *tail* concurrently only if necessary.

Figure 8.3. Sleep-wait P with FIFO Enqueue
Using Efficient Busy-Wait Locking/Waiting/Unlocking

```
P(input: adr_new_entry, sleep_flag):

global variable:  sem, head, (tail,V_count);
local, register variable:  old_sem, new_sem, (old_tail,old_V_count), (nil,new_V_count);

begin
    fetch(instruction blocks);
    wait;                                       /* wait for cache to finish fetching */

    lock(sem, old_sem);
    new_sem := old_sem - 1;
    unlock(sem, new_sem);

    if old_sem > 0 then                         /* will not sleep-wait */
    begin
        sleep_flag := 0;
        return;
    end;
                                                /* old_sem ≤ 0, may need to enqueue entry and sleep-wait */
    fetch(instruction blocks);
    nil := NIL;
    adr_new_entry→link := NIL;
    sleep_flag := 1;
    wait;                                       /* wait for cache to finish fetching */

    lock((tail,V_count), (old_tail,old_V_count));

    if old_V_count > 0 then                     /* outstanding V's, don't enqueue entry */
    begin
        sleep_flag := 0;
        new_V_count := old_V_count - 1;
        unlock((tail,V_count), (nil,new_V_count));   /* tail is already NIL */
        return;
    end;
    else                                        /* no outstanding V's, enqueue entry */
    begin
        if old_tail = NIL then                  /* queue empty, update head */
            head := adr_new_entry;
        else                                    /* else queue not empty, update tail link */
            old_tail→link := adr_new_entry;
        unlock((tail,V_count), (adr_new_entry,0));
    end;
end;
```

Notes for Figures 8.3, 8.4

1. The instruction ‘lock(x, old_x)’ locks the variable x, then executes $old\_x := x$, for register old_x.
2. The instruction ‘unlock(x, new_x)’ executes $x := new\_x$, for register new_x, then unlocks the variable x.
3. The instruction ‘unlock(x)’ simply unlocks the variable x.

Figure 8.4. Sleep-wait V with FIFO Dequeue
Using Efficient Busy-Wait Locking/Waiting/Unlocking

V(output: *old_head, wakeup_flag*)**:**

global variable: *sem, head, (tail,V_count);*
local, register variable: *old_sem, new_sem, old_head, new_head, (old_tail,old_V_count), (nil,new_V_count);*

```
begin
    fetch(instruction blocks);
    wait;                                     /* wait for cache to finish fetching */

    lock(sem, old_sem);
    new_sem := old_sem + 1;
    unlock(sem, new_sem);

    if old_sem ≥ 0 then                       /* will not wakeup a process */
    begin
        wakeup_flag := 0;
        return;
    end;
    fetch(instruction blocks);                /* old_sem < 0, may need to dequeue entry for wakeup */
    nil := NIL;
    wakeup_flag := 1;
    wait;                                     /* wait for cache to finish fetching */

    lock(head, old_head);

    if old_head = NIL then                    /* queue may be empty, check tail */
    begin
        lock((tail,V_count), (old_tail,old_V_count));
        if old_tail = NIL then                /* queue is empty (could alternatively check head here) */
        begin
            new_V_count := old_V_count + 1;
            unlock((tail,V_count), (nil,new_V_count));
            unlock(head);                     /* no need to write; but if did write, must do so before unlocking tail */
            return;
        end;
        else                                  /* head updated since locked: no longer NIL, so read it again */
        begin
            unlock(tail);                     /* no need to write */
            old_head := head;
        end;
    end;
                                              /* old_head ≠ NIL: queue not empty, dequeue head entry */
    new_head := old_head→link;
    if new_head = NIL then                    /* queue now empty, update tail too */
    begin
        lock(tail, old_tail);                 /* lock tail to prevent concurrent enqueuing */
        if old_tail = old_head then           /* queue still empty, update tail */
        begin
            unlock(head, nil);                /* head must be written before tail is set to NIL and unlocked */
            unlock(tail, nil);
        end;
        else                                  /* queue no longer empty */
        begin
            unlock(tail);                     /* no need to write */
            new_head := old_head→link;        /* get new link written by enqueue algorithm */
            unlock(head, new_head);
        end;
    end;
    else                                      /* queue not empty */
        unlock(head, new_head);
end;
```