

Scheduling Policies to Support Distributed 3D Multimedia Applications *

Thu D. Nguyen and John Zahorjan

Department of Computer Science and Engineering
University of Washington, Seattle, WA 98195

*To Appear In Proceedings of the SIGMETRICS '98/ PERFORMANCE '98 Joint International
Conference on Measurement and Modeling of Computer Systems, June 1998*

Abstract

We consider the problem of scheduling the rendering component of 3D multimedia applications on a cluster of workstations connected via a local area network. Our goal is to meet a periodic real-time constraint.

In abstract terms, the problem we address is how best to schedule tasks with unpredictable service times on distinct processing nodes so as to meet a real-time deadline, given that all communication among nodes entails some (possibly large) overhead. We consider two distinct classes of schemes, *static*, in which task reallocations are scheduled to occur at specific times, and *dynamic*, in which reallocations are triggered by some processor going idle. For both classes we further examine both *global* reassignments, in which all nodes are rescheduled at a rescheduling moment, and *local* reassignments, in which only a subset of the nodes engage in rescheduling at any one time.

We show that global dynamic policies work best over a range of parameterizations appropriate to such systems. We introduce a new policy, Dynamic with Shadowing, that places a small number of tasks in the schedules of multiple workstations to reduce the amount of communication required to complete the schedule. This policy is shown to dominate the other alternatives considered over most of the parameter space.

1 Introduction

The work in this paper is motivated by our effort to build a system that uses a cluster of workstations to improve the performance of multimedia applications requiring real-time 3D rendering. In our prototype system, a VRML viewer [23],

*This work supported in part by the National Science Foundation (Grants CCR-9704503 and CCR-9200832), Microsoft Corporation, and Intel Corporation. An extended version of this paper is available as reference [14].

which allows interactive manipulation of 3D scenes downloaded over the World Wide Web, is the canonical application. In this context, improving performance means extending the complexity of the scenes that can be rendered at a sufficiently fast as well as consistent frame rate. Our goal is to achieve this performance gain through the use of distributed rendering on a cluster of commodity workstations connected by a commodity local area network (LAN).

While building the prototype presents many challenges, this paper is restricted to formulating a model for and proposing solutions to the following problem: given a decomposition of the work into independent tasks, how should the processors be scheduled to maximize the probability that all tasks are executed at least once before a real-time deadline expires? We make the following assumptions:

- *The application can be decomposed into multiple, independent tasks.*
In our application, a task can correspond to the rendering of a portion of the final 2D image or of a specific set of scene objects.
- *The tasks are executed on multiple processors connected via a broadcast LAN.*
Our prototype system consists of five 180 MHz SGI O2 workstations connected via a dedicated 100 Mb/s switched Ethernet.
- *The goal of the scheduling policy is to maximize the probability that all tasks are completed before a real-time deadline expires.*
In our application, the deadline corresponds to the frame time, the inverse of the frame rate. Smooth motion requires at least 10 frames per second (fps), implying a real-time period of at most 100 ms. At the current TV rate of 30 fps, the period is reduced to 33 ms.
- *All tasks are available at the beginning of the computation.*
In our application, the set of objects in the scene is assumed to be static. Each frame begins with the broadcast of the new viewpoint to be used in rendering that

frame, but (essentially) no other information needs to be exchanged.

- *The tasks have unpredictable execution time requirements.*

This is reasonable in our environment because the work required to render a single object can change dramatically from frame to frame. In particular, if an object is completely out of view in a frame, it can be culled very quickly. However, a small change in the viewpoint in the next frame can bring it (partially) into view, greatly increasing the amount of work required to deal with it.

- *The tasks have statistically identical execution time requirements.*

We assume that a higher level component of the scheduler, not addressed here, uses information about object rendering times obtained over a sequence of frames to occasionally regroup scene objects into roughly equal sized tasks. (Example of such schemes have been described in [5, 15].)

- *The cost of communicating for the purposes of scheduling is relatively large, on the order of a few percent of the deadline time.*

Because we assume the application communicates over a commodity LAN, scheduling overhead includes the latency and computational costs associated with messages sent over such a network. Viewed as a fraction of the frame time, the fixed overhead of sending a message can be quite high.

- *The CPU is a scarce resource.*

On commodity workstations, rendering typically uses both the CPU and the graphics hardware accelerator: the CPU is responsible for the geometric transformation phase while texturing and rasterization is done on the graphics hardware. With such an architecture, the CPU is often the system bottleneck.

In the work presented here, we assume that the scene description has been replicated before rendering begins, so assigning tasks to processors is efficient, requiring only the transmission of task IDs. While it may appear that replicating the scene description limits the size of the scenes that can be rendered, in practice this is not a problem. Rendering is a CPU/graphics accelerator and memory bandwidth bound problem. Thus, scene descriptions that stress the main memory capacities of current workstations are almost certainly too complicated to be rendered in real-time, even on multiple processors.

1.1 Related Work

Although our work is motivated by the parallel rendering problem, unlike much of the previous work on parallel rendering [3], we are as concerned with minimizing the inter-frame variance as we are with maximizing the mean frame

rate. Thus, in this paper, we address the problem of scheduling to maximize the probability of meeting a real-time deadline as opposed to structuring the parallel rendering system to maximize the mean frame rate. Our work is perhaps closest in spirit to that of Shekhar et al. [20]. While Shekhar et al. consider some similar policies, they are not concerned with meeting a real-time deadline.

Our work also differs in a number of respects from the established literature on real-time scheduling. For one thing, in contrast to the classical work on schedulability (e.g., [10, 19]) as well as scheduling of fault-tolerance real-time systems (e.g., [8, 2]), there is a single deadline by which all of our tasks must be completed, rather than a deadline per task. Additionally, we do not make any assumption about the maximum service time of individual tasks (other than that it is shorter than the frame time), and do not use task service time information in any of our policies.

Our work is related to efforts in loop scheduling for parallel processors (e.g., [18, 21, 11, 16]) in that the basic problem a loop scheduling discipline must solve is how to balance the performance loss due to processors going idle when work remains against the overhead of finding that work. Our environment differs from loop scheduling, however, as our overheads are considerably larger, and so there is greater emphasis on reducing the number of scheduling operations performed. Additionally, because we are dealing with a broadcast communication medium, we have the potential to amortize a single rescheduling overhead by communicating a new schedule to multiple processors, an opportunity all of our suggested policies exploit. Lastly, we evaluate the policies based on the probability that a deadline is met, rather than on the average time to complete all the work.

Finally, our work is also related in spirit to earlier results on load balancing in distributed systems (e.g., [12, 25, 4, 6]), especially those that dealt with real-time tasks (e.g., [9, 1]). However, the former had as a goal minimizing response time, rather than meeting real-time deadlines, while the latter addressed workloads in which each task had its own deadline, rather than our situation in which there is a single deadline for the ensemble of tasks. Additionally, we deal with a system containing a fixed number of tasks, rather than one subject to a stream of arrivals.

1.2 Paper Structure

In the next section, we present the abstract model we use to examine how to schedule systems like the one described above. Section 3 provides an overview of the policies we consider, while Section 4 compares their performance. Section 5 concludes the paper.

2 Model Overview

Our model is quite simple. There are P processors, each with a queue of tasks. There is an average of N tasks per

processor in the system, or NP tasks in all. The goal is to complete the NP tasks within a frame time, which is taken to be the unit of time in the model.

For the reasons mentioned in Section 1, we assume that task times are random variables. More specifically, we assume that task execution times are exponentially distributed. A major advantage of the exponential over other potential distributions is that it allows us to find an optimal schedule for the static policy we propose. Additionally, it is the maximum entropy distribution [7], and so is motivated by the absence of information available at this time on actual task time distributions (which is highly data dependent in any case).

We denote the mean task service time by $1/\mu$. When specifying a model, though, we typically give the mean computational load per processor, denoted by ρ . As the unit of time in our model is the frame time, the mean task service time and the load are related by $1/\mu = \rho/N$.

The final parameters of our model reflect the overhead of communication. We break this into two parts, the computational requirement, C , and the “lag”, L . The former is the CPU time consumed on a single processor by the communication. The latter reflects the additional delay (beyond the cost C) for a processor that has gone idle to receive new work. Thus, L is defined to be the total elapsed time between a processor sending a message and its receiving a reply, minus that processor’s computational cost C . (In addition to the “on the wire” latency of the LAN, the lag L includes the time required by the system receiving the message to field the interrupt, to pass the message through the protocol stack to the application, to perform whatever application level work is required to compose a reply, and to send the reply message.) The local processor is available to perform useful work, if there is any available, during the lag time L , but not during the computational overhead C .

We evaluate a policy by computing the probability that all NP tasks are completed by the deadline when the policy is employed. However, because individual task times are exponential random variables, it is possible that a specific set of task times exceeds the processing power available, and so cannot be scheduled by any discipline. To factor out these impossible task sets, we normalize our results by dividing by the probability that a randomly chosen set of NP exponential tasks would complete on a system with P processors, a single shared queue, and zero scheduling overhead. We denote the normalized probabilities by $P[Success]$, and use them as the performance metric for comparison throughout the paper. (Recall that in our application domain, it is acceptable to miss a small number of deadlines, i.e., we do not insist that $P[Success]$ equals 1.)

3 Overview of Policies

All of our policies begin the frame time with an initial schedule that partitions the tasks equally among the processors.

This is the “best guess” initial policy, since the tasks are assumed to have statistically identical execution times.

To help meet the deadline, a policy may reassign tasks from one processor to another as the frame progresses. When such reassignments take place, the policy must specify how many of the remaining tasks to assign to each processor involved in the reassignment. All our policies rebalance the tasks each time a reassignment takes place. By balancing the workload at each rescheduling moment, we hope to reduce the total number of rescheduling operations required. Additionally, there is little motivation to consider schemes that move fewer tasks than must be moved to rebalance, because we must transmit only task IDs to communicate the new schedule (and so message costs increase only negligibly with the number of tasks) and because we are communicating over a broadcast medium (so that a single message suffices to update the schedules on multiple processors).

The questions that define a policy, then, are *when* to perform task reassignments and *which processors* to involve in each reassignment.

There are two approaches to deciding when to reassign, which we call *static* and *dynamic*. The distinguishing characteristic of static policies is that all task reassignments take place at specific, pre-computed times. In contrast, under dynamic policies, reassignments occur when the system enters a specific state. For all our dynamic policies, reassignments are triggered by some single processor going idle.

It should not be immediately evident which class of policy is preferable. As well as the obvious tradeoffs between the static and dynamic approaches, there is a practical consideration of some importance that gives an advantage to static schedules: because they schedule reassignments at particular times, the lag component of communication can be overlapped with useful computation. This overlap is not possible under dynamic schemes.

There are also two approaches to deciding which processors to involve in a reassignment, which we call *local* and *global*. In global policies, the loads of all processors are rebalanced at reassignment times. In local policies, each processor is associated with a fixed subset of other processors and exchanges loads only with those processors. Global policies are possible for the rendering application only when the scene description can be fully replicated.

We now present four alternative scheduling policies for our real-time environment.

3.1 Static Multiple Reassignment (SMR)

Under Static Multiple Reassignment (SMR), a predetermined list of reassignment times is used to trigger reassignments: at each reassignment time, any unfinished tasks are redistributed as evenly as possible over all participating processors. Reassignments can be either global or local.

To fully define SMR, we must specify a static set of reassignment times. This can be done in one of two ways.

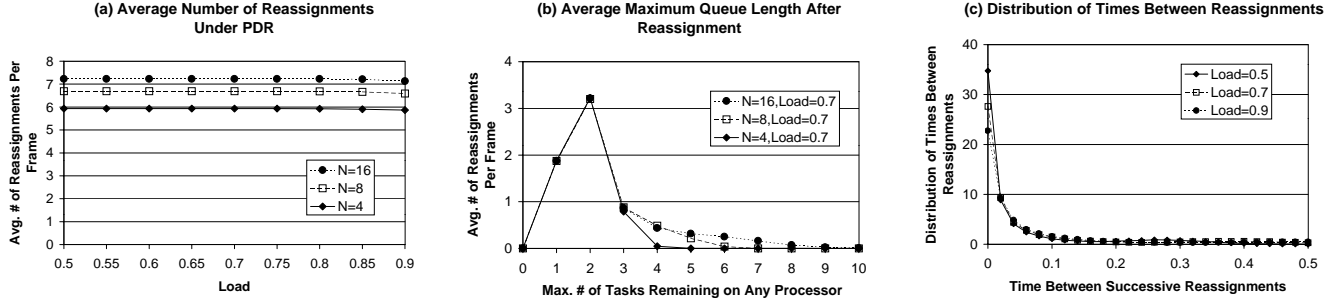


Figure 1: *Reassignment Statistics for an 8 Processor System Under PDR.*

In the first, there is a single list of times at which reassignments will take place, independently of the state of the system at each reassignment. In the second, the precomputed time at which the k th reassignment takes place is specified as a function of the number of tasks remaining at the $k - 1$ st reassignment. The second approach is clearly more flexible and so has an advantage over the first. Thus, we consider schedules of this more general type.

The technique we use to evaluate SMR allows us to compute static reassignment schedules that are optimal, subject to some restrictions: we assume that we have precise information on the average number of tasks per processor and the expected service time of each task, that the tasks are exponentially distributed, and that the reassignments are constrained to take place on a discretized time scale. This technique is described in the Appendix.

3.2 Pure Dynamic Reassignment (PDR)

Under Pure Dynamic Reassignment, a reassignment is triggered each time a processor goes idle, except that once no processor is assigned more than one task the reassignments cease. When a reassignment takes place, unfinished tasks are balanced across the participating processors as evenly as possible. Reassignments can be either global or local.

One potential problem with Pure Dynamic Reassignment is the number of reassignments that occur. Figure 1a shows the average number of reassignments per processor (using global reassignment) for an eight processor system under various loads and numbers of tasks. It is clear that the number of reassignments is largely insensitive to load (which is expected, since the number of times some processor goes idle under this policy is unaffected by load). We see also that the number of reassignments can be quite large.

It is intuitively clear that most of the reassignments that take place under Pure Dynamic must occur when there are relatively few tasks left per processor. Measurements confirm this intuition. Figure 1b shows the average number of reassignments per frame that leaves a maximum number of tasks assigned to any processor given by the X-axis value.

Figure 1c shows the distribution of the time between successive reassignments. Clearly, there is a flurry of reassignments that occurs towards the end of the schedule.

The next two disciplines attempt to improve upon Pure Dynamic Reassignment by reducing the number of reassignments that occur during this final period.

3.3 Dynamic with Delay Reassignment (DDR)

Dynamic with Delay operates identically to Pure Dynamic until the average number of tasks assigned to each processor at some reassignment is two or less. At that point, a processor going idle waits a delay time that is a parameter of the policy before triggering a reassignment. The delay time allows multiple processor idle events to be responded to with a single reassignment, thus reducing communication overheads at the cost of some increase in processor idleness.

3.4 Dynamic with Shadowed Reassignment (DSR)

Like Dynamic with Delay, Dynamic with Shadowed Reassignment operates identically to Pure Dynamic until the average number of tasks assigned to each processor is two or less¹. At that point, Dynamic with Shadowed Reassignment creates a final schedule, i.e., one that will be followed without further reassignment. Thus, Dynamic with Shadowed Reassignment is guaranteed to perform no more reassignments than Dynamic with Delay, and is likely to perform many fewer.

The danger in discontinuing reassignments, of course, is that load imbalance among the remaining tasks will cause the deadline to be missed. To reduce this probability, Dynamic with Shadowed Reassignment places each of a subset of the remaining tasks in the final schedules of all processors. (We call these multiple assignments *shadowed assignments*, and the corresponding tasks *shadowed tasks*. Note that shadowed assignments are legal because the tasks are

¹In fact, both DDR and DSR are easily defined for threshold numbers of tasks other than two. In our evaluations, though, we found that the best performance is obtained when this number is two. Thus, we present the policies and their results using that value.

(a)	(b)	(c)																																																																																																		
<table style="margin: auto; border-collapse: collapse;"> <tr><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">0</td></tr> <tr><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">1</td></tr> <tr style="border-top: 1px solid black;"><td style="padding: 2px 10px;">P0</td><td style="padding: 2px 10px;">P1</td></tr> </table>	1	0	0	1	P0	P1	<table style="margin: auto; border-collapse: collapse;"> <tr><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">0</td></tr> <tr><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">1</td></tr> <tr><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">2</td></tr> <tr><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">3</td></tr> <tr style="border-top: 1px solid black;"><td style="padding: 2px 10px;">P0</td><td style="padding: 2px 10px;">P1</td><td style="padding: 2px 10px;">P2</td><td style="padding: 2px 10px;">P3</td></tr> </table>	3	2	1	0	2	3	0	1	1	0	3	2	0	1	2	3	P0	P1	P2	P3	<table style="margin: auto; border-collapse: collapse;"> <tr><td style="padding: 2px 10px;">7</td><td style="padding: 2px 10px;">6</td><td style="padding: 2px 10px;">5</td><td style="padding: 2px 10px;">4</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">0</td></tr> <tr><td style="padding: 2px 10px;">6</td><td style="padding: 2px 10px;">7</td><td style="padding: 2px 10px;">4</td><td style="padding: 2px 10px;">5</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">1</td></tr> <tr><td style="padding: 2px 10px;">5</td><td style="padding: 2px 10px;">4</td><td style="padding: 2px 10px;">7</td><td style="padding: 2px 10px;">6</td><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">2</td></tr> <tr><td style="padding: 2px 10px;">4</td><td style="padding: 2px 10px;">5</td><td style="padding: 2px 10px;">6</td><td style="padding: 2px 10px;">7</td><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">3</td></tr> <tr><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">7</td><td style="padding: 2px 10px;">6</td><td style="padding: 2px 10px;">5</td><td style="padding: 2px 10px;">4</td></tr> <tr><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">6</td><td style="padding: 2px 10px;">7</td><td style="padding: 2px 10px;">4</td><td style="padding: 2px 10px;">5</td></tr> <tr><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">5</td><td style="padding: 2px 10px;">4</td><td style="padding: 2px 10px;">7</td><td style="padding: 2px 10px;">6</td></tr> <tr><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">4</td><td style="padding: 2px 10px;">5</td><td style="padding: 2px 10px;">6</td><td style="padding: 2px 10px;">7</td></tr> <tr style="border-top: 1px solid black;"><td style="padding: 2px 10px;">P0</td><td style="padding: 2px 10px;">P1</td><td style="padding: 2px 10px;">P2</td><td style="padding: 2px 10px;">P3</td><td style="padding: 2px 10px;">P4</td><td style="padding: 2px 10px;">P5</td><td style="padding: 2px 10px;">P6</td><td style="padding: 2px 10px;">P7</td></tr> </table>	7	6	5	4	3	2	1	0	6	7	4	5	2	3	0	1	5	4	7	6	1	0	3	2	4	5	6	7	0	1	2	3	3	2	1	0	7	6	5	4	2	3	0	1	6	7	4	5	1	0	3	2	5	4	7	6	0	1	2	3	4	5	6	7	P0	P1	P2	P3	P4	P5	P6	P7
1	0																																																																																																			
0	1																																																																																																			
P0	P1																																																																																																			
3	2	1	0																																																																																																	
2	3	0	1																																																																																																	
1	0	3	2																																																																																																	
0	1	2	3																																																																																																	
P0	P1	P2	P3																																																																																																	
7	6	5	4	3	2	1	0																																																																																													
6	7	4	5	2	3	0	1																																																																																													
5	4	7	6	1	0	3	2																																																																																													
4	5	6	7	0	1	2	3																																																																																													
3	2	1	0	7	6	5	4																																																																																													
2	3	0	1	6	7	4	5																																																																																													
1	0	3	2	5	4	7	6																																																																																													
0	1	2	3	4	5	6	7																																																																																													
P0	P1	P2	P3	P4	P5	P6	P7																																																																																													

Figure 2: *Shadowing Schedules on (a) 2, (b) 4, and (c) 8 Processors. (Only shadowed tasks are shown. The full schedule on each processor also includes a single, unshadowed task that precedes the processor’s shadowing schedule.)*

completely parallelizable and there is no shared memory in the systems we consider.) Each processor then runs until the deadline is reached, executing tasks in the order given by its final schedule. The frame is successful if every task is completed at least once by the deadline, and unsuccessful otherwise.

An important part of the Dynamic with Shadowed Reassignment policy is the “shadowing schedule,” that is, the ordering on each processor of the shadowed tasks. The scheme we use to create the shadowing schedule attempts to satisfy the following properties:

1. Any task that appears first in the schedule of any processor once the initial rebalancing has taken place is never shadowed; if these tasks fail to complete by the deadline on the processor to which they are assigned, they certainly will not complete by appearing later in the schedule of some other processor.
2. All other tasks (i.e., the shadowed tasks) are scheduled exactly once on every processor.
3. Each shadowed task appears at position k in the schedule of exactly one processor, $1 \leq k \leq P$. (The unshadowed task on a processor is in position 0, and the shadowing schedule follows.)
4. For each pair of shadowed tasks X and Y , X appears before Y in the schedules of half the processors, and after Y in the schedules of the other half. The purpose of this is to minimize the impact of an unusually long task on the likelihood that other tasks will be completed by the deadline.

These goals can be met exactly when the number of processors is a power of two, and the number of shadowed tasks equals the number of processors. We first show how to construct such a schedule in this case, and then explain how we adapt these schedules for the general case.

Theorem 1 *Let the shadowing schedule for a single processor be a single shadowed task. Represent the shadowing schedule for a P -processor system as a $P \times P$ matrix, with the schedule for processor p being column p , read bottom to top. The shadowing schedule for $2P$ processors formed by*

composing two P -processor shadowing schedules, M and M' , containing distinct sets of shadowed tasks, according to the pattern

$$\begin{matrix} M' & M \\ M & M' \end{matrix} \quad (1)$$

has properties 1-4 listed above.

Proof: Property 1 holds trivially because no such tasks are included in the base case schedules. The other properties hold by induction. All three are true for the base case of a single processor schedule. For the induction step, property 2 holds because each column of M (M') contains one instance of each task in M (M'), and M and M' contain distinct task sets. Property 3 holds because each row of M (M') contains one instance of each task in M (M'), and M and M' contain distinct task sets. Finally, property 4 holds by definition when the two tasks X and Y both come from M (M'). For X in M and Y in M' , it holds because M precedes M' the same number of times that it follows M' in the $2P$ -processor schedule. \square

Figure 2 shows the shadowing schedules for 2, 4, and 8 processors when the number of shadowed tasks equals the number of processors. In this (and following) figure(s), we use integers to represent task IDs. The schedule for an individual processor is given as a column, with the schedule read from bottom to top.

To treat the general case, we need to allow sets of shadowed tasks that are smaller than the number of processors, and numbers of processors that are not powers of two. We do the former by adding dummy tasks to increase the number of shadowed tasks to equal the number of processors. An intermediate schedule is built as described above. The dummy tasks and their shadowed assignments are then removed to form the final schedule. Of course, when a schedule is constructed in this manner, some task(s) will be shadowed more than once at each level; this is inevitable when there are less shadowed tasks than there are processors. However, properties 1, 2, and 4 above are still guaranteed to hold.

When multiple dummy tasks must be added, they should be placed as “far apart” as possible. This is significant because the regularity of the schedules increases the duplication of particular tasks at each level if the dummy tasks are placed close to each other.

We construct a simple scheme for placing the dummy tasks when the number of processors is a power of 2. For a system with P processors, we construct a complete binary tree with $\log_2(P) + 1$ levels. At each internal node, the left child edge is labeled with 0 and the right with 1. Each leaf is labeled with the number corresponding to the binary value of the path from that leaf back to the root, with the label of the last edge (i.e., the edge next to the root) being the least significant bit. Figure 3 shows such a tree for an 8-processor system. Reading the leaves’ labels from left to right then gives a permutation of $0, 1, 2, \dots, P - 1$. When multiple dummy tasks are to be added, the i^{th} dummy task is placed at the processor given by the i^{th} number in this permutation ($0 \leq i < P - 1$).

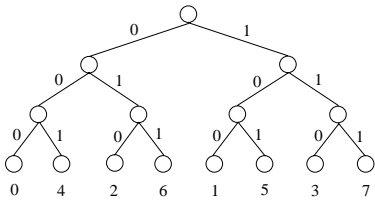


Figure 3: *Dummy Task Placement Tree for an 8-Processor System.*

We deal with numbers of processors that are not powers of two in a similar way: we insert dummy processors and dummy shadowed tasks to construct an intermediate schedule. We then remove the columns corresponding to dummy processors and all shadowed assignments of dummy tasks to obtain the final schedule. When adding multiple dummy processors, we place them in the same way that we place dummy tasks. Figure 4 gives an example for a 6-processor system, with dummy processors denoted by Xs and dummy tasks by asterisks.

7	6	5	*	3	2	1	*
6	7	*	5	2	3	*	1
5	*	7	6	1	*	3	2
*	5	6	7	*	1	2	3
3	2	1	*	7	6	5	*
2	3	*	1	6	7	*	5
1	*	3	2	5	*	7	6
*	1	2	3	*	5	6	7
X	p0	p1	p2	X	p3	p4	p5

Figure 4: *Shadowing Schedules for 6 Processors.*

4 Policy Comparison

In this section, we compare the performance of the policies described in the previous section. Recall from Section 2 that our performance metric is $P[\text{Success}]$, the (normalized) probability that all NP tasks are completed by the deadline when a particular policy is employed.

We calculate $P[\text{Success}]$ values for the static policy using the dynamic program described in the Appendix. Results for all other policies are obtained via simulation. Enough trials were run in all cases that the 95% confidence interval for all $P[\text{Success}]$ probabilities falls in the (absolute) range $\pm .008$. To help ensure a fair comparison among policies, all simulations used the same random sequence of task times to drive the trials.

In addition to the four policies from Section 3, we also consider a policy intended to represent the large class of parallel loop scheduling techniques that have been developed. We call this policy Idealized Loop Scheduling (ILS). We include ILS in our comparisons to emphasize that the real-time nature of our environment and the magnitudes of the scheduling overheads are important to designing appropriate policies, and that disciplines designed for a different, but related, domain will not perform well.

While it is not possible to include every aspect of all the variants of loop scheduling in ILS, we believe that ILS is optimistic with respect to this class of disciplines because, in evaluating it, we have artificially set many of the scheduling overheads these policies incur to zero. Specifically, under ILS, all processors begin the frame with an assignment of $\frac{N}{2P}$ tasks. The remainder of the tasks are kept in a common work pool. When a processor completes its assigned work, it obtains $\frac{1}{2P}$ of the remaining work from the work pool². In our evaluation of ILS, we ignore the overhead required to maintain the work pool and any possible contention to access it. When a processor accesses the work pool, it alone is charged the cost of communication.

4.1 Model Parameterization

Recall from Section 2 that our model has five parameters: the number of processors, P , the number of tasks per processor, N , the mean computational load per processor, ρ , and the communication overhead per reassignment, which is comprised of two components, the computational requirement, C , and the lag, L .

We consider systems with P ranging from 4 to 16, realistic sizes for the application domain that we are most interested in, and N ranging from 4 to 16. (Recall that each task represents many scene objects.) Because of our assumption that task times are exponential, the number of tasks per processor determines the coefficient of variation of the per processor workload. For this range of N , the coefficient of variation ranges from 0.5 to 0.25.

We vary the system load, ρ , from 50% to 90%.

Finally, we parameterize C and L using measurements of the overheads experienced on our prototype system. Broadcasting a single message to all processors and then waiting until a message is received from each (using UDP/IP con-

²This policy is very similar to a number of loop scheduling policies that have been proposed for NUMA systems [11, 13, 16, 24], and shares with nearly all loop scheduling strategies the essential property that chunk sizes decrease as the size of the work pool decreases.

sumes about 0.3 ms of CPU on both the sender and the receivers, with a lag of about 0.7 ms^3 . Surprisingly, the effect of the number of receivers on these times is below the threshold of what we can measure. We speculate that this is because multiple messages that arrive close together in time are dequeued and enqueued together as they move through the protocol stack, and that the major expense of handling small messages has to do with the overhead of moving through the stack, but we have no way to verify this explanation.

Based on these measurements, we set the ratio of C to L to be 3:7 and consider a range of total communication overhead, $C + L$, from 1% to 3% of the frame time. We consider this range because it covers what we expect to be the most common cases for real-time rendering applications: for systems with communication overheads like that measured on our prototype system, this range spans frame rates from 10 to 30 fps; at a constant 30 fps, this range spans total (round-trip) communication overheads from the 1 ms of our prototype system down to 333 μs .

4.2 Results

We evaluated all policies for the 27 possible sets of model parameters given by $P = \{4, 8, 16\}$, $N = \{4, 8, 16\}$, and $(C + L) = \{1\%, 2\%, 3\%\}$ for loads ρ from 0.5 to 0.9. Because of space limitations, we present only some representative results here, and limit our discussion to the cases where full replication of the scene description is possible (the global policies). A longer version of this paper [14] (available online) presents the full set of results, and shows that global variants of the policies dominate the local ones.

Figures 5 and 6 show six representative graphs comparing the performance of the global variants of SMR, DSR, PDR, and DDR, and of ILS, for communication overheads of 1% and 3%, respectively. These results lead us to the following conclusions.

Over the range of P we have examined, it is possible to scale a real-time application of the type we consider by a factor of at least $P/2$ but (considerably) less than P , even when communication overheads are large relative to the deadline time.

All the scheduling disciplines we studied are able to meet the deadline essentially every frame when the per processor load is at most 50%. Thus, problems that are at least $P/2$ times the size of those that can be computed by the deadline on a single processor should be manageable on P processors. On the other hand, none of the disciplines is able to meet an acceptably high fraction of the deadlines when the per processor load is high.

For systems of the type and sizes we have examined, a dynamic policy seems preferable to the static policy.

The static policy (SMR) is dominated by one or more dynamic policies in nearly all of our experiments. The exceptions are when there are many processors and many tasks

³Unfortunately, current production operating systems impose overheads that are considerably larger than those achievable experimentally (e.g., [17, 22]).

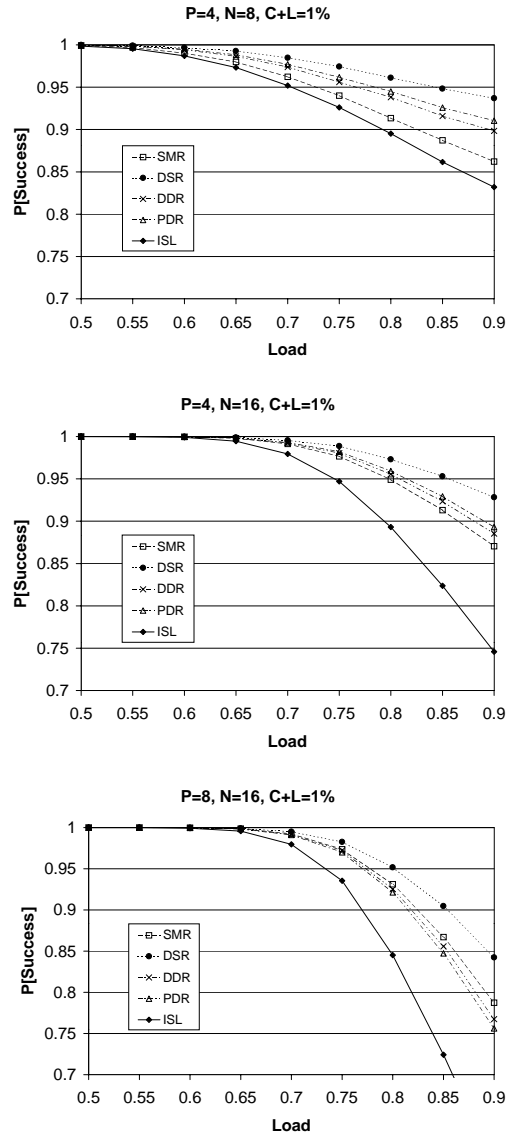


Figure 5: Sample Performance Results for Total Communication Overhead $C + L = 1\%$.

(which corresponds to low workload variance), and the communication overhead is large.

As our evaluation of the static policy is optimistic with respect to any real implementation, we conclude that dynamic policies are most appropriate for the prototype system we are building and others like it. However, for systems intended to support many more processors than we have considered, and/or have communication overheads that are larger fractions of the deadline time than we have considered, further effort to find a practical static policy might be warranted.

The use of shadowed assignment to efficiently reduce the amount of communication required to complete the schedule provides best performance among the dynamic policies.

Both DDR and DSR were designed to reduce the num-

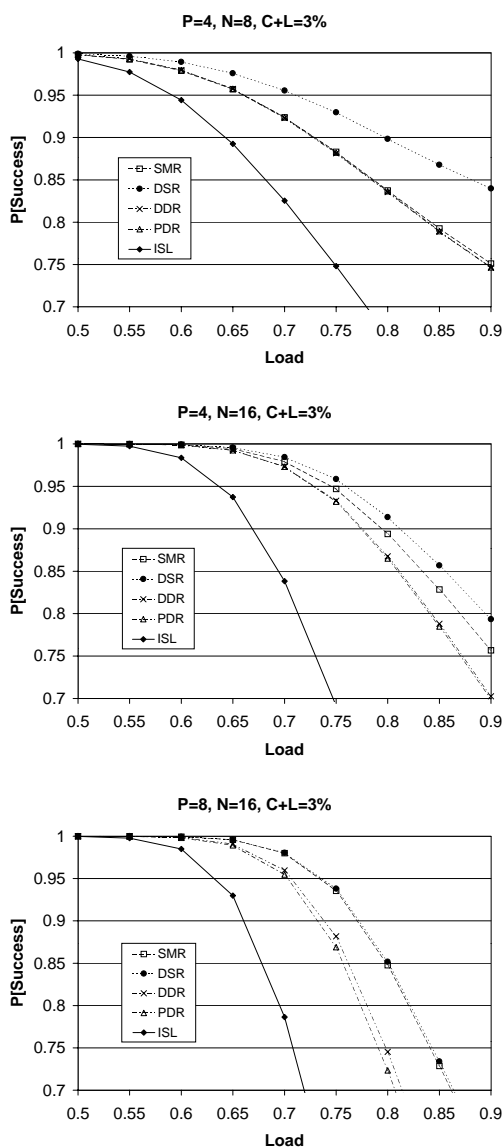


Figure 6: Sample Performance Results for Total Communication Overhead $C + L = 3\%$.

number of reassignments that occur near the end of the frame time under PDR. Both policies achieve this immediate goal: for N between 4 and 16, DDR performs just over 4 reassignments per frame, while DSR performs between 2 and 3. However, Figures 5 and 6 show that DSR makes the best tradeoff between the number of reassignments and load imbalance losses: DSR performance dominates PDR and DDR in every experiment we examined.

Before concluding that shadowed assignment is the key characteristic, however, we must verify that DSR does not outperform DDR simply because it stops reassigning work altogether when the average number of tasks per processor falls below 2. We do this by introducing a new policy, PDR-SE, that is identical to DSR except that the final schedule includes only a rebalancing of unfinished tasks, and has no

shadowed assignment. (Thus, PDR-SE is also equivalent to a PDR policy that “stops early,” i.e., when the maximum number of tasks assigned to any processor at a reassignment is no more than two.) Figure 7a shows a representative graph comparing DSR, DDR, and PDR-SE. Clearly, the fact that PDR-SE under-performs DDR while DSR outperforms it shows that shadowed assignment is key to obtaining best performance.

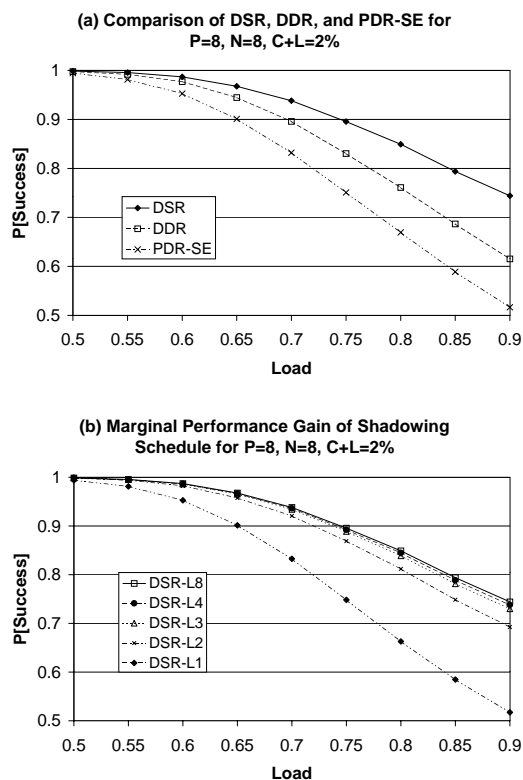


Figure 7: (a) Comparison of DSR, DDR, and PDR-SE, and (b) Marginal Performance Gain of Successive Levels of DSR’s Shadowing Schedule.

A related question is how much of the performance advantage of DSR over PDR-SE is gained from the execution of successive levels of the shadowing schedule. That is, how deep do processors typically get into their shadowing schedules before the deadline expires? To answer this question, we look at the performance of DSR when the shadowing schedule has been truncated at different levels. Figure 7b shows a representative graph, where DSR- L_n denotes the policy DSR with its shadowing schedule truncated to n levels. Note that DSR-L1 is equivalent to PDR-SE when the number of shadowed tasks is equal to the number of processors. We observe that much of DSR’s performance advantage derives from the execution of the 2nd level of the shadowing schedule, which corresponds to the first shadowing assignment of each shadowed task. However, the next 2 to 3 levels also afford some additional performance gain.

The loop scheduling policy is not competitive even under

our idealized assumptions.

ILS has performance similar to the other policies at small P , N , and $C + L$, but degrades rapidly as any one of these parameters grows. This is not surprising given that loop scheduling policies were designed for an environment with different capabilities and cost model than ours.

5 Conclusions

As well as being an important application on its own, real-time rendering is an essential component of an increasing number of multimedia applications (e.g., VRML, virtual reality, data visualization, and geographical information systems). The work in this paper is motivated by our effort to build a system that improves the performance of real-time rendering through the use of multiple commodity workstations connected by a commodity LAN. A key problem that must be confronted in building this system is how to schedule the tasks associated with rendering a frame to maximize the probability that the frame time deadline is met. In a cluster connected by a commodity LAN, this problem is especially challenging because communication overheads can be high relative to the frame time. Thus, scheduling policies designed for this environment must deal effectively with large communication overheads when balancing the overhead of scheduling against the performance loss due to processors being left idle when unfinished work remains.

We have proposed and evaluated a set of policies that can take advantage of the degree of data replication possible in the system to improve performance. We have shown how to analyze static policies that reschedule at specific, pre-computed times, and have compared them with dynamic policies. We found that static policies are not the right choice for the range of system parameters most appropriate for our prototype distributed rendering system, but that they may hold promise in more extreme portions of the parameter space. Instead, a dynamic policy employing a new technique called shadowing that assigns each of a small number of tasks to multiple processors provides best performance, regardless of the amount of data replication possible.

References

- [1] A. Bestavros. Load Profiling in Distributed Real-Time Systems. *Information Sciences*, 101(1–2):1–27, Sept. 1997.
- [2] C.-I. H. Chen and V. Cherkassky. Task Reallocation for Fault Tolerance in Multiprocessor Systems. In *Proceedings of the IEEE 1990 National Aerospace and Electronics Conference*, pages 495–500, May 1990.
- [3] T. W. Crockett. An Introduction to Parallel Rendering. *Parallel Computing*, 23(7):819–843, July 1997.
- [4] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.
- [5] D. A. Ellsworth. A New Algorithm for Interactive Graphics on Multicomputers. *IEEE Computer Graphics and Applications*, 14(4):33–40, July 1994.
- [6] M. Harchol-Balter and A. B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. In *Proceedings of the ACM SIGMETRICS Conference*, pages 13–24, May 1996.
- [7] D. D. Kouvatsos. Entropy Maximisation and Queuing Network Models. *Annals of Operations Research*, 48(1–4):63–126, Jan. 1994.
- [8] C. M. Krishna and K. G. Shin. On Scheduling Tasks with a Quick Recovery from Failure. *IEEE Transactions on Computers*, c-35(5):448–455, May 1986.
- [9] J. F. Kurose and R. Chipalkatti. Load Sharing in Soft Real-Time Distributed Computer Systems. *IEEE Transactions on Computers*, c-36(8):993–1000, Aug. 1987.
- [10] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.
- [11] J. Liu and V. A. Saletore. Self-Scheduling on Distributed-Memory Machines. In *Supercomputing '93*, pages 814–823, Nov. 1993.
- [12] M. Livny and M. Melman. Load Balancing in Homogeneous Broadcast Distributed Systems. In *Proceedings of the ACM Computer Network Performance Symposium*, pages 47–55, April 1982.
- [13] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, Apr. 1994.
- [14] T. D. Nguyen and J. Zahorjan. Scheduling Policies to Support Distributed Multi-Media Applications. Technical Report 97-11-03, University of Washington, Department of Computer Science and Engineering, (http://www.cs.washington.edu/homes/zahorjan/ddddrraw/papers/intraframe_lb_abstract.html), Nov. 1997.
- [15] D. Nicol and J. Saltz. Dynamic Remapping of Parallel Computations with Varying Resource Demands. *IEEE Trans. on Computers*, 37(9):1073–1087, September 1988.
- [16] S. Orlando and R. Perego. Exploiting Partial Replication in Unbalanced Parallel Loop Scheduling on Multicomputers. *Microprocessing and Microprogramming*, 41(8–9):645–658, Apr. 1996.

- [17] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. *IEEE Concurrency*, 5(2):60–72, Apr. 1997.
- [18] C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, Dec. 1987.
- [19] K. Ramamritham, J. A. Stankovic, and P.-F. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), Apr. 1990.
- [20] S. Shekhar, S. Ravada, V. Kumar, D. Chubb, and G. Turner. Declustering and Load-Balancing Methods for Parallelizing Geographic Information Systems. *IEEE Transactions on Knowledge and Data Engineering*, To Appear.
- [21] T. H. Tzen and L. M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, Jan. 1993.
- [22] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 40–53, Dec. 1995.
- [23] The VRML Consortium. *VRML*, <http://www.vrml.org>.
- [24] Y.-M. Wang, H.-H. Wang, and R.-C. Chang. Clustered Affinity Scheduling on Large-Scale NUMA Multiprocessors. *Journal of Systems and Software*, 39(1):61–70, Oct. 1997.
- [25] Y.-T. Wang and R. J. T. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, c-34(3):204–217, Mar. 1985.

A Computing Performance Results for Static Multiple Reassignment

We describe here how we compute $P[Success]$ under Static Multiple Assignment. We begin with some preliminary notation, then describe the dynamic program that is the key to this computation.

Let $R_{n,t}$ be a row vector of truncated Poisson probabili-

ties of length $N + 1$ defined by⁴

$$R_{n,t}[k] = \begin{cases} 1 - \sum_{i=0}^{n-1} \frac{(\mu t)^i}{i!} e^{-\mu t} & k = 0 \\ \frac{(\mu t)^{n-k}}{(n-k)!} e^{-\mu t} & 1 \leq k \leq n \\ 0 & n+1 \leq k \leq N \end{cases} \quad (2)$$

where $R_{n,t}[k]$ denotes the k th element of $R_{n,t}$. $R_{n,t}[k]$ is the probability that k tasks remain unfinished after a service interval of length t by a single server, given that the tasks are exponentially distributed with mean $\frac{1}{\mu}$ and that the server starts with n customers total.

Now define a row vector $S_{n,t}$ of length $NP + 1$ where the k th element represents the probability that k tasks remain unfinished after a service interval of length t on a system with P independent servers that starts with a total of n tasks assigned as evenly as possible to the queues of those servers. By definition, $M_L = n \bmod P$ processors begin with $m_L = \lceil \frac{n}{P} \rceil$ tasks and $M_S = P - M_L$ processors begin with $m_S = \lfloor \frac{n}{P} \rfloor$ tasks. Then we have

$$S_{n,t} = \begin{pmatrix} M_L \\ * \\ R_{m_L,t} \end{pmatrix} * \begin{pmatrix} M_S \\ * \\ R_{m_S,t} \end{pmatrix} \quad (3)$$

where $*$ denotes vector convolution and $\begin{pmatrix} m \\ * \\ V \end{pmatrix}$ means the m -fold convolution of vector V with itself.

We are now prepared to describe the dynamic program we use to compute $P[Success]$ under SMR. Let $Q_t^{(j)}$ be a column vector of length $NP + 1$ whose k th element is the probability that a system starting with k tasks allocated as evenly as possible on P processors can complete those tasks using no more than j statically assigned reassignments during an interval of length t . Then we have

$$Q_t^{(j+1)}[n] = \max \left(\max_{0 \leq s \leq t-C} S_{n,s} Q_{t-s-C}^{(j)}, Q_t^{(j)}[n] \right) \quad (4)$$

with the base condition

$$Q_t^0[n] = S_{n,t}[0] \quad (5)$$

The final step is dealing with the maximization problem. We do this by discretizing time: we assume that all reassignments take place at times $t = k\epsilon$, where k is an integer and $\epsilon = \frac{1}{D}$ for some large integer D . This allows us to determine the value of s that maximizes the first term in Equation (4) by examining all D possibilities.

Additionally, this discretization bounds the number of reassignments that can possibly be performed in an optimal reassignment schedule to D . Thus, we have

$$P[Success] = Q_1^{(D)}[NP]/G(N, P, \rho) \quad (6)$$

where $G(N, P, \rho)$ is the normalizing probability that an idealized single queue, P processor, zero overhead system will complete NP tasks by the deadline.

⁴The symbols N , P , μ , and C have the same meanings in this appendix as throughout the paper, and are the only symbols taken from the body of the paper. All functions defined here depend on these values, but we do not explicitly include them in our notation to avoid clutter.