Dynamic versus Static Load Balancing in a Pipeline Computation *

Anna Brunstrom
brunstro@cs.wm.edu

Rahul Simha
simha@cs.wm.edu

Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185

Abstract

We examine load balancing in a simple pipeline computation, in which a large number of data sets is pipelined through a series of tasks and load balancing is performed by distributing several available processors among the tasks. We compare the performance of the optimal static processor assignment to the performances of three dynamic processor assignment algorithms. Models are derived which allow us to approximate the performance of the dynamic algorithms theoretically. The relative performances of the algorithms are investigated for various amounts of overhead, using a combination of modeling and simulation. We indicate that an appropriate dynamic algorithm can improve performance even when the overhead induced by the algorithm is relatively high.

Keywords: pipeline computation, load balancing, performance evaluation

*A condensed version of this paper was presented at the IASTED Sixth International Conference on Parallel and Distributed Computing and Systems, 1994. This research was partially supported by a National Science Foundation grant, # NCR-9109792.

1 Introduction and Motivation

Parallel machines with large numbers of processors will soon be in wide use. As they get cheaper and more accessible, several generic computations which are today executed on serial machines will be ported to parallel machines to improve response times. One such generic computation is a pipeline computation in which data sets are processed by a series of interrelated tasks [1, 4, 22]. It is well-known that the performance of a parallel implementation of almost any computation (including that of a pipeline computation) is strongly dependent on *load balancing*, a problem at the very heart of parallel computing. To improve performance and fully utilize the power of parallel machines the load of the computations must be distributed as evenly as possible among the processors. For a pipelined computation, load balancing can be achieved by distributing the processors among the tasks; this distribution can be done statically or dynamically. In a static load balancing scheme the processors are assigned to the tasks only once and prior to execution. On the other hand, a dynamic load balancing scheme will attempt to adjust the processor assignments dynamically in response to changing system loads. The potential gain of a dynamic scheme is evident; however a dynamic scheme may impose high system overhead if changing the assignments of processors is costly, a tradeoff we examine in this paper.

We consider a pipeline computation consisting of n tasks to be performed in series. We have a pool of c processors to assign to the tasks and focus on large parallel machines, in which $c > n$. A stream of data sets are to be processed by the computation. Our objective is to minimize their average response time through the system. We model our system using a tandem queueing network. We compare the performance of several dynamic processor assignment algorithms and the optimal static processor assignment algorithm. The relative performances of the algorithms are examined under varying degrees of overhead. Simple approximations are derived which allow us to compare the dynamic load balancing algorithms to the static algorithm using theoretical calculations. The validity of the developed approximations is verified by simulation.

Our system of tasks is representative of several applications. For example, the transport layer in a network protocol performs a series of operations such as checksum computation, address decoding and framing on a stream of incoming packets [19]. A discussion of the issues involved in designing network protocols which adapt to current high-speed network technology is given in [18], with a particular focus on the performance of the transport layer. Pipelining and parallelization of the transport layer is recognized as a possibility for improving communication protocol performance in [11]. Another application arises in computer vision where motion estimation can be done by performing a series of tasks on a stream of incoming image pairs [1].

Some related processor assignment problems have been studied in the literature, all of which consider static assignments and mostly focus on the combinatorial aspect of assigning processors to tasks with known execution times. Sevcik [20] considers static allocation of processors during the initialization of an application. Algorithms based on varying degrees of information on the parallelism inherent in the applications are compared. Krishnamurti and Ma [13] give an approximation algorithm for assigning processors to a set of independent tasks. The problem of deciding the optimal processor partition sizes is shown to be NP-complete. Choudhary et. al. [1] examine another static processor assignment problem. Given a set of tasks, their precedence graph, and their response time functions, algorithms are derived which are optimal in processor assignment with respect to response time and with respect to throughput. In [4] pipelined data-parallel algorithms are discussed. An analytic model based on petri nets is presented, as well as methods for partitioning the computation. In [22] the tasks to be performed and their dependencies are represented by a DAG. An algorithm for mapping this graph to a particular hardware, the Pipelined Image-Processing Engine, is presented.

Some previous results on queueing systems are also relevant to our work. Our dynamic policies resemble scheduling rules used in routing customers to parallel queues [3, 8, 15]. Our queueing model is based on the approximate performance model for the join-shortest-queue rule described in [15]. The approximation is based on the observation that the behavior of the system of parallel queues can be related to the behavior of a single M/M/c queue. In [14] a mixture of several classes of customers arrive to a system of parallel queues, in accordance with the arrival rate specified for each class. Constraints are imposed on which queues can serve customers belonging to a particular class. A probabilistically optimal load balancing algorithm is derived.

The problem of routing a single stream of customers to parallel queues has been extended to a distributed system with multiple arrival streams by several authors. In [7] trade-offs between different dynamic and static policies are studied through analysis and simulation. A study of the behavior of UNIX processes is used in the evaluation and design of load balancing policies in [26]. Zhou uses trace-driven simulation to evaluate several load balancing algorithms in [28]. Experimental results comparing three dynamic load balancing polices can be found in [9]. The benefits of job migration for load balancing purposes has also been studied in the literature [6].

Other queueing models which involve servers handling multiple queues include the mobile server problem [16] and the much-studied polling system [23]. In [16] service disciplines are studied for multiple mobile servers on a graph when requests arrive randomly at the various nodes and are queued for service. Basic polling models consider a single server who serves multiple queues in cyclic order. A survey of the analytical results established for various polling models can be found in [23], along with an extensive list of references. The most closely related polling system (to our model), in which a single roving server handles a Jackson network of queues, is examined in [21].

This paper is structured as follows. In Section 2 we define the system under consideration and formulate our problem. The different algorithms studied are described in Section 3. Section 4 describes our modeling of the system under the various algorithms. Our results are presented in Section 5 and a summary of the paper in Section 6.

2 System Model

In this section we define our model of the system. A pipeline computation consists of multiple computational tasks which must be applied to each of several data sets. The sequencing constraints present between the computational tasks are described by a task structure. We consider only the simplest form of pipeline computation in which a number of tasks are to be executed in series. We have a pool of processors which need to be assigned to the tasks in order to minimize the average response time of a job going through the system. Note that the "job" above corresponds to a data set in the pipeline application. The data sets processed by the computation are assumed to be independent. We use the following notation to describe our system.

- n is the number of tasks in the computation. Tasks are labeled 1 through n.
- c is the total number of processors available.
- c_i is the number of processors assigned to task i. At all times the relationship $\sum_{i=1}^{n} c_i = c$ must hold.
- λ is the arrival rate of jobs to the system. We will assume that the arrivals are Poisson.
- μ_i is the service rate for task i. We will assume service times to be exponentially distributed. $\mu = (\mu_1, \mu_2, \ldots \mu_n)$.
- $\overline{W}$ is the average delay through the system.
- $\overline{W}_i$ is the average delay experienced at task i. Note that $\sum_{i=1}^{n} \overline{W}_i = \overline{W}$.

Our problem at hand then consists of finding processor assignments c_1 through c_n that minimize $\overline{W}$. The system described above can be modeled as a network of tandem queues as illustrated in Figure 1. Each task in the pipeline computation corresponds to a node in the network. There are c_i processors (servers) assigned to task (node) i and the service rate for the task is μ_i.

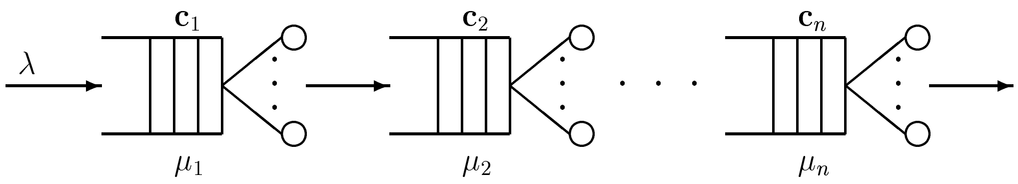

Figure 1: Model of the system.

3 Algorithms

In this section we give a description of the various algorithms considered for assigning processors to the tasks. The optimal static algorithm and three dynamic algorithms are examined. In a static algorithm the processor assignments are determined prior to execution and remain fixed throughout the computation. In contrast, a dynamic algorithm may adjust the processor assignments during execution in response to the current system state. The modeling of the algorithms and their relative performance will be discussed in subsequent sections.

3.1 A Static Algorithm

The simplest form of assignment policy is a static policy, where the processors are distributed among the tasks once and never reassigned thereafter. When the processors are never reassigned the queueing network modeling the computation becomes a Jackson network [12] where each node individually behaves as a M/M/c queue. It was shown for the M/M/c case in [5] and for the general G/G/c case in [25] that delay as a function of the number of servers (processors) is a convex function. Hence the method of marginal analysis [10] can be used for allocating processors to the tasks, yielding optimal static assignments. The steps of the static algorithm can then be outlined as follows:

1. *Allocate to each task the minimum number of processors required to give a utilization less than one.*

2. *Allocate the remaining processors to the tasks one at a time; allocate each processor to the task where it produces the greatest reduction in delay.*

The description above assumes that there is a sufficient number of processors available to keep up with the arrivals of jobs to the system. When the service rates in the system are homogeneous, that is $\mu_1 = \mu_2 = \cdots = \mu_n$, the algorithm simplifies to distributing the processors evenly among the tasks.

The advantages of the static algorithm are its simplicity and the fact that it imposes no overhead on the system. However, the algorithm may lead to poor processor utilization since it does not respond to load imbalance in the system.

3.2 A Simple Dynamic Algorithm

We describe a simple dynamic policy that strives to keep a processor busy whenever possible. Idle processors are reassigned based on the lengths of the queues associated with each task. This dynamic algorithm can be summarized by the following two rules:

1. *If a processor finishes servicing a job and the queue associated with its current task is empty, then the processor is reassigned based on the following criteria:*

   (a) *If there are unattended jobs in the system, then the processor is reassigned to the task with the longest queue. If there are several tasks with the same longest queue length, then one of these tasks is selected at random.*

   (b) *If there are no unattended jobs in the system, then the processor maintains its current assignment.*

2. *If a new job arrives to the system and there are no idle processors assigned to the first task but there are idle processors available in the system, then an idle processor is randomly selected and reassigned to the first task.*

The simple dynamic policy prohibits the simultaneous existence of an unattended job and an idle processor. This guarantees maximum possible utilization of the processors. Reassigning the idle processors to the longest queues in the system improves the chance of a processor servicing several jobs before being reassigned again. It also prohibits large queue build-ups in the system.

It is intuitive that the above policy will perform very well when there is no overhead involved in reassigning processors. Keeping server utilization maximized has shown to be a key to performance in routing customers to parallel queues: shortest queue routing is the optimal routing policy for homogeneous parallel queues [24, 27]. Shortest queue routing attempts to balance the queue lengths in the system which in turn maximizes server utilization. Thus the idea behind the policy is similar to the idea behind our dynamic algorithm. However, if it is costly to reassign processors the algorithm will suffer from high overhead. This observation motivates the next dynamic algorithm.

3.3 A Threshold-Based Algorithm

The threshold algorithm is similar to the simple dynamic algorithm but the reassignment of processors is occasionally restricted. Only when the gain of reassigning a processor exceeds a certain threshold is the reassignment performed under the threshold policy. It is assumed that the mean service time for each task is known and that the expected delay (before service) for the last job in a queue is simply the sum of $k+1$ inter-service times, where k is the number of jobs in the queue ahead of the last job and the inter-service time is the time in-between job completions. The following rule describes the threshold algorithm:

1. *If a processor completes service and at least one more processor is assigned to its current task and the queue for the task is empty, then the processor is eligible for reassignment. The reassignment rules are:*

   (a) *The processor is reassigned to the task whose last customer has the largest expected queueing delay, if this delay exceeds the cost of reassigning the processor.*

   (b) *Otherwise, the processor maintains its current assignment.*

The arrival of a new job never directly causes the reassignment of a processor. The requirement to always keep at least one processor assigned to each task is needed to avoid deadlock. Without this requirement the system could end up in a state where there were no jobs in the system and no processors assigned to the first task. The system would then be deadlocked: all the processors would remain indefinitely idle and new jobs arriving to the system would never get serviced. To always keep at least one processor assigned to each task is a reasonable requirement and is often desirable for meeting throughput requirements [1].

The decision to potentially reassign a processor is based solely on local information. When a processor completing service finds an empty queue and at least one additional processor assigned to its current task, reassignment is considered. Only then does global information need to be acquired. For the simple dynamic algorithm global information must also be acquired each time a job arrives to the system and there are no idle processors assigned to the first task. Observe that the threshold algorithm is more resilient to overhead than the simple dynamic policy. A processor only gets reassigned when the expected gain resulting from the reassignment exceeds the expected reassignment cost. Thus, the threshold policy adapts the reassignment frequency to the amount of overhead involved in reassigning a processor, a feature absent from the simple dynamic algorithm. Note that the threshold algorithm does not guarantee maximum possible processor utilization.

3.4 A Pipeline-Based Dynamic Algorithm

The pipeline algorithm tries to take advantage of the fact that the tasks are arranged in a pipeline. Consider a job in service at a task early in the pipeline. When it leaves service it still has to wait for the jobs at tasks later in the pipeline. Not until these jobs are completed can it get serviced at the remaining tasks and leave the system. On the other hand, a job at the last task in the pipeline can leave the system as soon as it finishes service. Thus, in order to minimize the average response time it can be more advantageous to service the jobs later in the pipeline first. This idea is used in the pipeline policy. The pipeline policy is outlined below:

1. *If a processor completes service and there are unattended jobs in the system, then the processor is reassigned to the last task in the pipeline with a non-empty queue.*

2. *If there are no unattended jobs in the system, then the processor keeps its current assignment.*

3. *If a new job arrives at the first task with no idle processors assigned to it but there are idle processors in the system, then an idle processor is randomly selected and reassigned to the task.*

The pipeline algorithm is a modified version of the simple dynamic algorithm which tries to utilize the structure of the system. The pipeline algorithm inflicts even more processor reassignments than the simple dynamic algorithm. A processor finishing service may be reassigned even when there are jobs waiting in the queue at its current task. Thus, the pipeline algorithm is even more sensitive to the overhead involved in reassigning a processor than the simple dynamic algorithm.

4 Modeling the system

In this section we consider how to calculate the delay in the system under our various processor assignment policies. Models are presented which allow us to calculate or approximate the delay theoretically. A discussion of overhead, as related to our system, is also given.

4.1 Basic Models

As explained in the previous section, when the static policy is used the tasks can be considered individually as M/M/c queues. The delay through the system is found by adding up the delay for the individual tasks where the delay in an M/M/c queue, $\overline{D}$, is given by the equation [12]:

$$\overline{D} = \frac{1}{\lambda}\left[\frac{(c\rho)^c}{c!}\frac{\rho}{(1-\rho)^2}\left(\frac{(c\rho)^c}{(1-\rho)c!}+\sum_{s=0}^{c-1}\frac{(c\rho)^s}{s!}\right)^{-1}+c\rho\right], \qquad \rho=\frac{\lambda}{c\mu} \tag{1}$$

Next, we approximate the delay in the system under the simple dynamic policy using the modeling steps outlined in Figure 2. For the moment we will ignore the overhead imposed by this algorithm. The first step in the modeling process corresponds to the system model which was displayed in Figure 1. The system is viewed as a network of tandem queues. Each node in the network corresponds to a task in the computation and the number of servers positioned at node i corresponds to the number of processors assigned to task i. Moving a server to a different node corresponds to reassigning a processor to a different task. We then note that under our dynamic policy a processor never goes idle when there are unattended jobs present in the system; thus, the system behaves as if all the jobs were in a common queue. We can model the system under the dynamic policy as a single feedback queue, with c servers, where each job goes through the system n times. The i^{th} pass for a job through the feedback queue corresponds to the job passing through the i^{th} queue in the tandem network. Thus, the expected service time for a job passing through the feedback queue for the i^{th} time is μ_i. Next, we observe that rather than forcing each job to go through the system exactly n times we can make a job pass through the system n times on the average using probabilistic (Bernoulli) feedback, a Markovian approximation. To model the variation in service time among the various passes of a job we use a hyperexponential service time distribution, constructed from the n original exponential service time distributions, such that each component exponential distribution is selected with equal probability. Hence, our system is now modeled as a feedback queue where the probability of feedback is $(n-1)/n$. This is the second modeling step illustrated in Figure 2. The behavior of the probabilistic feedback queue is similar to the behavior of a single M/H$_n$/c queue with the arrival rate adjusted to account for the feedback loop. Solving the balance equation associated with the feedback queue tells us that the adjusted arrival rate is $n\lambda$. The delay through the single M/H$_n$/c queue approximates the delay for one pass through the feedback queue. Finally, we will approximate the hyperexponential service time distribution by an exponential service time distribution of the same mean. An average queue in the system under our simple dynamic policy is now modeled by an M/M/c queue. This is the last modeling step illustrated in Figure 2. We can then approximate the delay through the pipeline system using equation 1. To approximate the delay through the pipeline system we simply multiply the delay given by equation 1 by n. The appropriateness of the approximation was verified by simulation

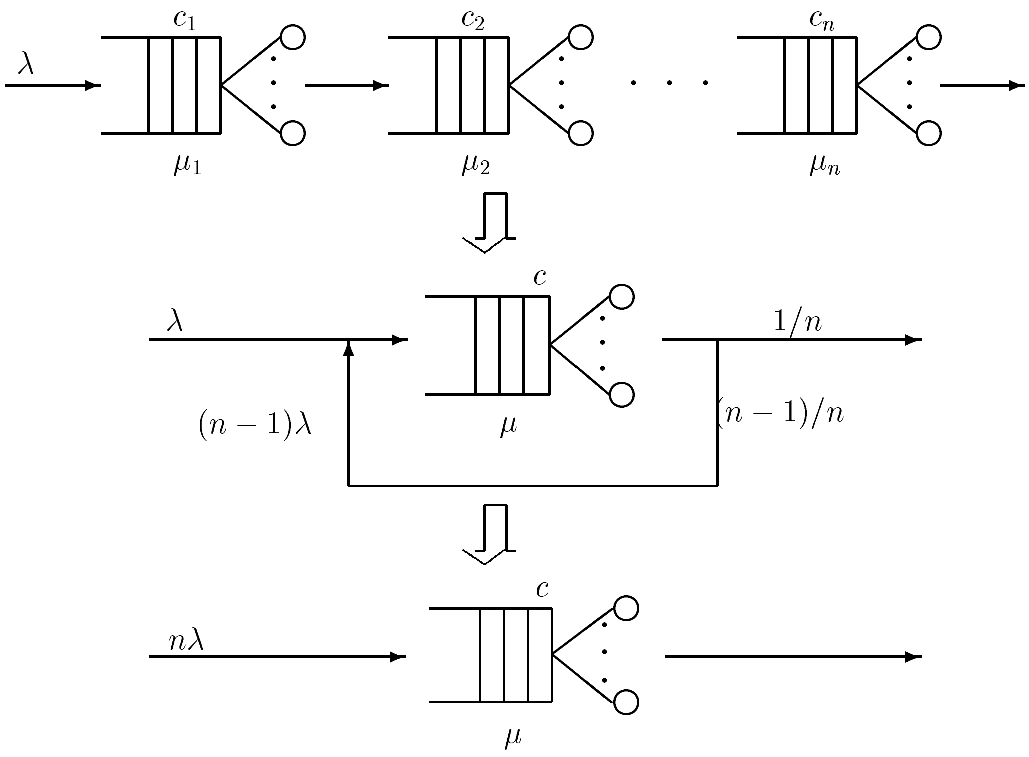

Figure 2: Modeling process

results, discussed in the next section.

In the special case of homogeneous service rates in the system the M/M/c model derived above can be shown to provide the exact mean delay through the pipeline system under the simple dynamic policy. In this case the hyperexponential service distribution, in our single queue model of the system, reduces to an exponential distribution and no approximation is necessary. The argument loosely runs as follows. On the average the single queue model and the system under our dynamic policy have the same number of jobs present. The jobs might be serviced in a different order but the rearrangement is not based on the service times of the jobs, and thus has no influence on the average delay.

When there is no overhead in the system the simple dynamic and threshold policy do not differ much in behavior. The threshold value is set to zero when there is no overhead thereby allowing frequent processor reassignments. The threshold policy still restricts reassignment of the processors somewhat by requiring that at least one processor is assigned to each task at all times, but if the number of processors is sufficiently large the threshold policy behaves similar to the simple dynamic policy. Hence, when there is no overhead the model above applies to both policies. We do not have a model for the pipeline policy and instead evaluate its performance via simulation.

4.2 Overhead

We now need to consider the effects of overhead on our system. There are two basic kinds of overhead involved in using a dynamic processor assignment algorithm. The overhead incurred in maintaining system state information and the overhead involved in reassigning processors. Each overhead is application and architecture dependent. In a shared memory machine the system state information can be kept in global memory at all times, keeping the overhead within reasonable limits. In a distributed memory machine the cost and complexity of maintaining system state information increases. The reassignment cost for a processor corresponds to the cost of a context switch. A processor is reassigned to a different task by loading a different program segment for execution. The cost will depend on the size of the tasks and on whether or not the new task already resides in main memory.

Our work is not tied to a particular application or architecture. Rather, we examine the relative performance of the optimal static policy and several dynamic processor assignment policies for various degrees of overhead. These results can then be used as guidelines when selecting an assignment algorithm for a particular architecture and application.

We expect the context switch in a processor to be the predominant source of overhead in a shared memory environment. We therefore focus on investigating the influence on performance by this type of overhead. The overhead involved in maintaining system state information is assumed to be negligible and will be ignored. Under this assumption only the jobs serviced by "reassigned" processors directly experience an overhead. A job is serviced by a reassigned processor if the processor has been reassigned since its last service completion. All jobs serviced by reassigned processors are assumed to suffer from the same amount of overhead. The overhead constant is calculated as a percentage of the average service time in the system.

4.3 Modeling Overhead

We now incorporate the overhead discussed above into our system models. The static algorithm is of course unaffected by overhead. We are assuming that only the jobs which are serviced by reassigned processors experience an overhead. Thus, we would like to get an estimate of the fraction of jobs which are serviced by such processors, for use in our approximation. Unfortunately, this is generally a hard number to characterize. As illustrated by Figure 3 the fraction of jobs serviced by a reassigned processor as a function of the arrival rate is a nonlinear function. The fraction of jobs serviced by a reassigned processor also depends on the number of tasks and the number of processors in the system as well as the processor assignment policy employed. For the simple dynamic algorithm the best bound we can provide is to assume that all jobs are serviced by a reassigned processor. An upper bound on the delay in the system, under our dynamic policy, can then be calculated by adding the overhead to the expected service time used in the M/M/c approximation derived above.

However, when there is a fair amount of overhead in the system the threshold policy performs better than the simple dynamic policy. The threshold policy only imposes the overhead of reassigning a processor when there is an expectedly larger gain to be made. Hence we will focus on modeling overhead for the threshold policy. For this policy we approximate the fraction of jobs serviced by a reassigned processor as 0.5. Our approximation is based on the following intuition. Consider the behavior of the threshold algorithm when the overhead for reassigning a processor is 50% of the average service time. Assume task i has the queue with the largest expected delay. On the average an idle processor will be reassigned to task i if there are at least $c_i + 1$ jobs in the queue, where c_i is the number of processors assigned to task i at the time of reassignment. The reassigned processor will service one job. The remaining c_i jobs will be serviced by a processor already assigned to task i. In the worst case $c_i = 1$ and each job serviced by a reassigned processor is matched by at least one job serviced by a stationary processor. This leads to our 0.5 approximation. To approximate the delay through the system we distribute the overhead over all the jobs. The previously derived M/M/c approximation can then be used by adding the expected overhead to the expected service time of a job.

In practice our 0.5 approximation turns out to be an overestimate. For non-negligible overhead the fraction of jobs processed by a reassigned processor under the threshold algorithm is often below 0.5. The values produced by the model work as an upper bound for the performance of the threshold algorithm rather than as an accurate approximation. Figure 3 shows the fraction of reassigned processors, recorded during simulation, versus arrival rate for the threshold and simple dynamic algorithms. The overhead associated with reassigning a processor was set at 25% of the average service time. The figure is for a system with 4 tasks, 16 processors, and service rate $\mu = (1.2, 0.2, 1.8, 0.7)$. Similar graphs were produced for different sets of parameter values. We can see that the graphs displaying the reassignment fractions for the two policies have similar shapes but the maximum reassignment fractions reached differ. As expected the simple dynamic algorithm has a much higher reassignment fraction than the threshold algorithm. The maximum reassignment fraction reached under the threshold policy is 20%, which is well below our 50% approximation.

5 Results

In this section we discuss results on the relative performance of the algorithms. The behavior of the algorithms under various degrees of overhead is examined. The results are obtained through simulation and through theoretical calculations. The simulation results verify that the approximations derived in the previous section are valid. A few comments on the simulation design are also included. All graphs in this section are for a system of 8 tasks and 32 processors, unless otherwise noted.

5.1 Simulation setup

Our simulations were structured as next event simulations. The two event types in the simulations are arrivals to the systems and departures from a task. Our simulations are direct extensions of the simulation of a multi-server service node described in [17]. Our simulations were modified to allow the servers (processors) to move between nodes in a network (tasks) in accordance with the particular processor assignment policy simulated. The method of batch means was used to calculate 95% confidence intervals of the average delay through the system. The size of the generated confidence intervals were generally within 1% of the value of their corresponding point estimates. For convenience the point estimates of the average delay will be displayed when presenting our results.

Overhead was handled in the simulations by adding an overhead constant to the service time of the jobs serviced by a reassigned processor. When a processor is reassigned, a job, queued for the task to which the processor is reassigned, is immediately assigned to the processor; the overhead constant is added to the service time of this job. An alternate approach would be to let the processor stay idle for the duration of the reassignment time, which is the overhead constant. If the queue for the task to which the processor was reassigned is non-empty at the end of this time interval a job would then be assigned to the processor. If the queue is empty, then the processor might be reassigned again. We have chosen the first approach which is simpler and which guarantees that a processor will get to service at least one job in between reassignments.

5.2 Homogeneous case

We will first consider the results for the case in which all tasks have a homogeneous service rate. For the graphs in figures 4, 5, and 6 the service rate was fixed at $\mu_i = 0.6$ and the overhead associated with processor reassignment was set at 0%, 25%, and 50% respectively. The performance of the static, simple dynamic, and threshold policy is displayed as well as the theoretical values obtained by using the approximation developed for the threshold

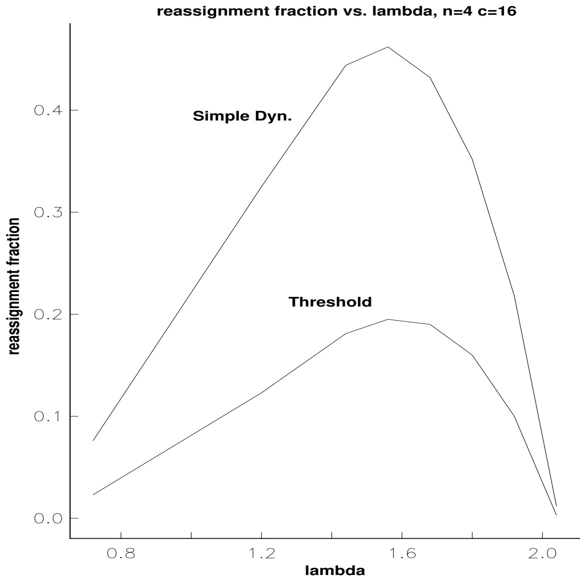

Figure 3: Reassignment fractions

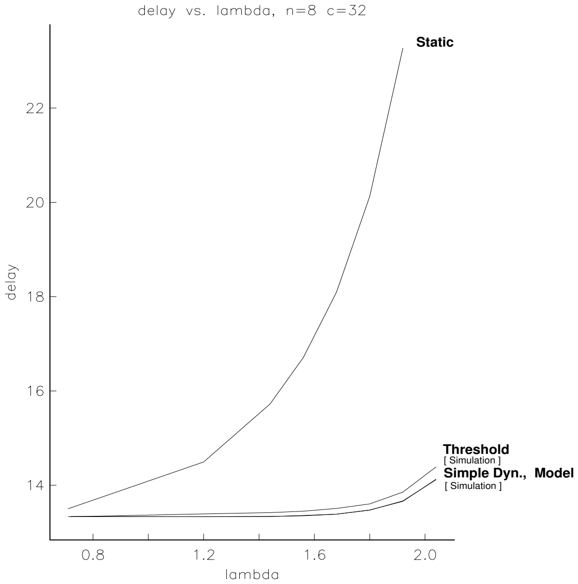

Figure 4: no overhead

policy. Similar graphs were produced for different sets of parameter values.

We can see from Figure 4 that the model derived in the previous section is correct. When there is no overhead the theoretical values produced match the values for the simple dynamic policy perfectly. Recall that we use the same model for the threshold and simple dynamic policies when there is no overhead. As expected, the threshold policy performs slightly worse but the theoretical values are still a good estimate for the threshold policy as well. As can be seen from Figures 5 and 6 the theoretical values, given by the model, provide an upper bound for the threshold policy when there is overhead in the system. The difference between the two curves appears because the processor reassignment fraction for the threshold policy is well below the 0.5 approximation.

When there is no overhead in the system the two dynamic processor assignment policies perform radically better than the static policy. The space between the static and dynamic curves represent the potential performance gain of a dynamic processor assignment algorithm when the overhead is negligible. As the overhead increases the performance of the two dynamic assignment algorithms diverges and the static algorithm starts to compare more favorably. As expected the threshold algorithm is the best dynamic assignment algorithm when there is overhead in the system. As can be seen from Figure 6 the threshold algorithm still clearly outperforms the static algorithm at 50% overhead. This shows that a dynamic load balancing scheme is profitable even at reasonably high overhead.

As we see from Figure 4 the simple dynamic policy performs very well when there is no overhead. This is as expected since the policy ensures maximum possible processor utilization. However, in a pipelined computation it seems intuitive that it should be advantageous

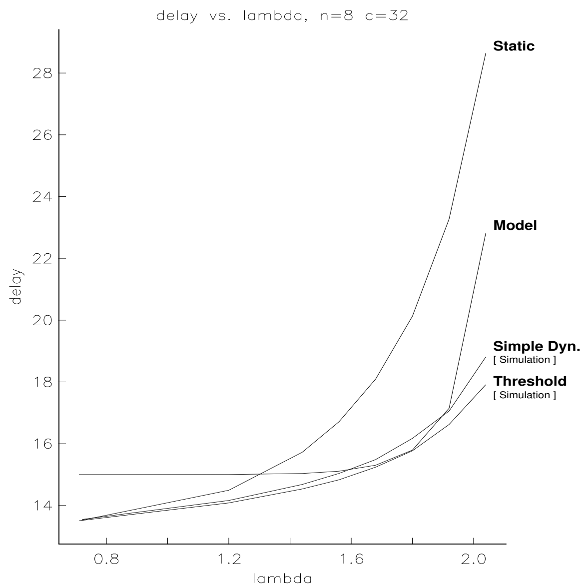

Figure 5: 25% overhead

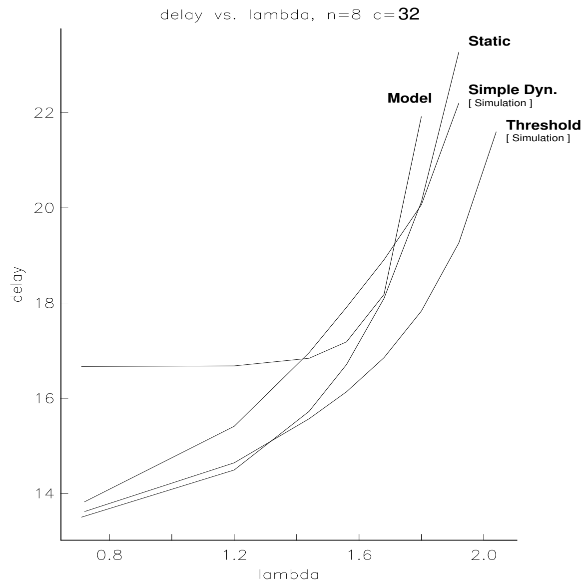

Figure 6: 50% overhead

to favor the tasks at the end of the pipeline. Since the queue for any task is equally likely to be the longest at a given time, the simple dynamic policy does not favor tasks at the end. To verify our intuition the performance of the pipeline algorithm was measured through simulation. Figure 7 compares its performance to the simple dynamic policy for a system of 4 tasks and 16 processors using a homogeneous service rate set at 0.6. As we can see the pipelined policy performs slightly better and our intuition is correct. However, the pipelined policy is only useful if there is no overhead. We will not consider it further.

5.3 Heterogeneous case

Next we will examine the case when the tasks have a heterogeneous service rate. Figures 8, 9, and 10 display the results for the static, simple dynamic, and threshold policy in addition to the values obtained from the threshold model. Service rates of (1.2, 0.2, 1.8, 0.7, 1.2, 0.2, 1.8, 0.7) were used and the overhead was set at 0%, 25%, and 50% of the average service time respectively. Graphs similar to the ones displayed in Figures 8, 9, and 10 were produced for other parameter values as well.

As in the homogeneous case we see that the model derived in the previous section is appropriate. For no overhead, Figure 8, the results for the simple dynamic policy almost match the values obtained from the model. The threshold policy performs slightly worse but its performance still corresponds well to the results given by the model. When overhead is introduced, in Figure 9 and 10, the model provides an upper bound to the performance of the threshold policy. As before the difference between the values produced by the model

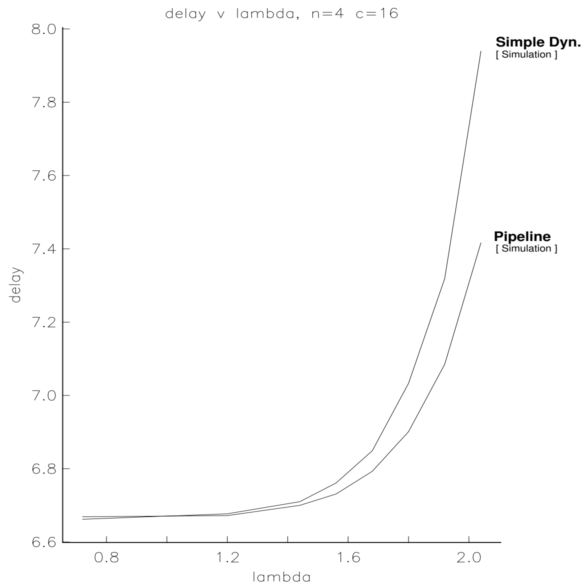

Figure 7: pipeline graph

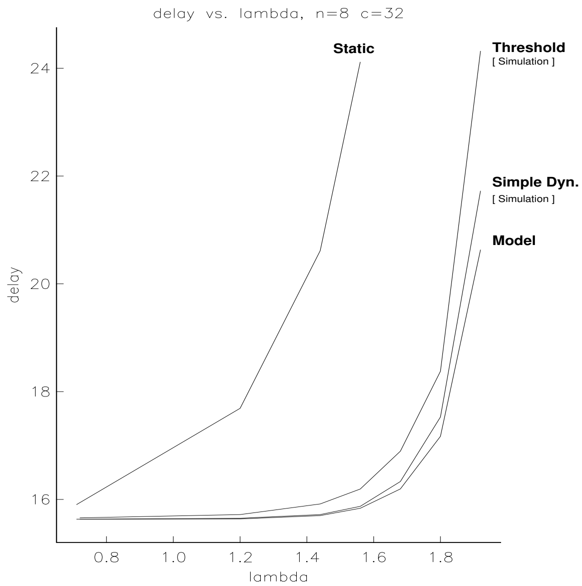

Figure 8: no overhead

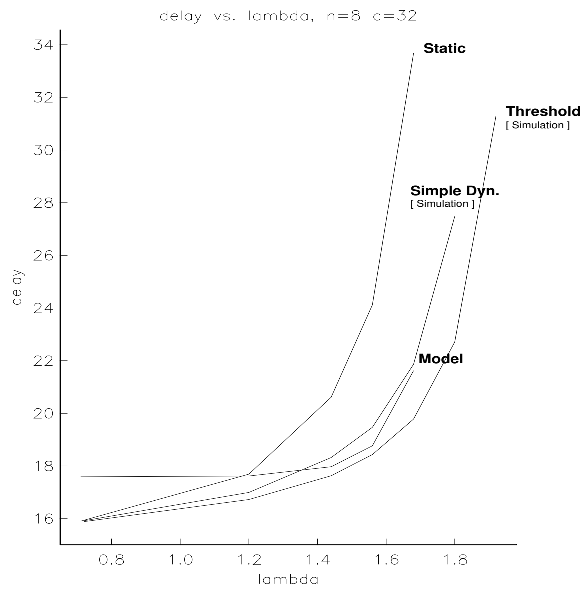

Figure 9: 25% overhead

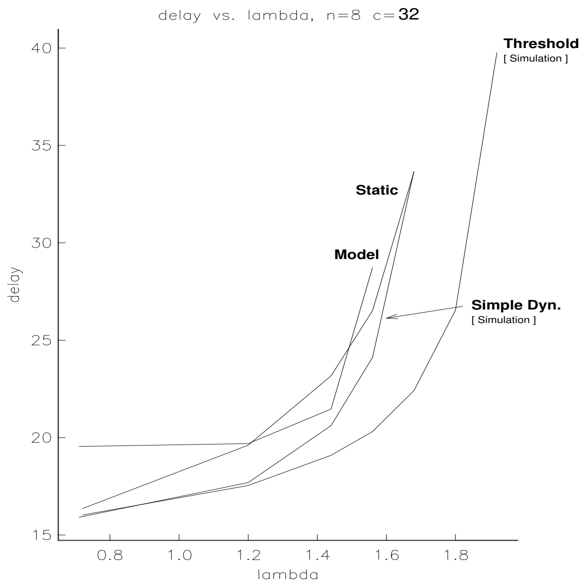

Figure 10: 50% overhead

and the simulation results for the threshold policy is explained by the actual reassignment fraction for the threshold algorithm being lower than 0.5.

Similar to the homogeneous case, we see that there is a great potential benefit in using a dynamic processor assignment algorithm when the overhead is negligible (Figure 8). As the overhead increases (Figures 9 and 10) the threshold algorithm presents itself as the superior dynamic processor assignment algorithm. The performance gap between the threshold algorithm and the static algorithm decreases. However, at 50% overhead (Figure 10) the performance of the threshold algorithm still distinctly surpasses the performance of the optimal static algorithm. Again we see evidence that an appropriate dynamic processor assignment algorithm is beneficial even when there is a fair amount of overhead involved in relocating processors.

5.4 Additional Comments

We saw above that the results for the homogeneous and heterogeneous cases were very similar. However, as a general trend we have noticed that the dynamic processor assignment algorithms compared slightly more favorably when the tasks had homogeneous service times or at least only small variations in their service times. Intuitively, the relative queue lengths will vary more in this situation and dynamic load balancing produces a higher payoff. If we have a few very time consuming tasks in the system, then it makes sense to use most of the processors for these tasks without ever reassigning them. Thus dynamic load balancing is not as effective in this situation.

Our simulation results have shown that the model derived in the previous section provides an approximation to the performance of the threshold algorithm. We can use the model to approximate how the threshold policy performs in comparison to the optimal static policy for a given overhead. Thus, given a set of system parameters and an estimate of how much overhead is involved in implementing the threshold policy we can assess the potential gain, if any, in using a dynamic processor assignment algorithm such as the threshold algorithm, via simple theoretical calculations. As an example, consider the motion estimation system described in [1], a serial pipeline consisting of 9 tasks corresponding to operations such as 2-D convolution, extraction of zero crossings and stereo matching – see [1, 2] for details. Experimental results on the average response time for each one of the tasks when executed on a shared memory machine, the Encore Multimax, were reported in [1]. The results for a single processor are reproduced in Table 1 for convenience. Assuming that the response times are approximately exponentially distributed and that load balancing is to be performed as discussed in this paper we can evaluate theoretically whether a dynamic algorithm should be considered. Let us further assume that we have 64 processors to distribute, that the expected

Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8	Task 9
352.20	16.54	0.85	51.70	352.20	16.54	0.85	212.00	25.50

Table 1: Single processor response times (sec.) for the Encore Multimax

arrival rate to the system is 0.05 and that the overhead associated with reassignment is 20%. Our theoretical calculations would then show that the expected delay through the system under the static allocation algorithm is approximately 1297 seconds and that the upper bound for the threshold algorithm is 1169 seconds. A dynamic algorithm should be considered. If the expected arrival rate is further increased to 0.055 the static algorithm yields an expected delay of 2055 seconds and the upper bound for the threshold algorithm is 1609 seconds. The potential benefit of a dynamic policy is higher for high arrival rates.

Eager et. al. [7] have studied load balancing in distributed systems. They showed that simple dynamic load balancing schemes are the most promising. The simple schemes perform vastly better than static policies and almost as well as more complex dynamic algorithms. The complex algorithms suffer from greater overhead, are more instable, and are more sensitive to inaccurate state information. Our results also show the great potential benefit of using a dynamic load balancing algorithm as opposed to a static algorithm. Since we assume the context switch in a processor to be the primary source of overhead it is reasonable to base the decision on whether to load a new task on as much information as possible. As the cost of maintaining state information increases relative to the cost of a context switch, less information-intense versions of the threshold algorithm should be considered.

6 Summary

We have considered processor assignments in a simple pipeline computation. A pool of processors were to be assigned to a series of tasks as to minimize the response time for a job passing through the system. We compared the optimal static load balancing algorithm to several dynamic load balancing schemes. The influence of overhead on their relative performance was examined. For non-negligible overhead the threshold policy showed to be the best dynamic processor assignment policy. It was shown that a dynamic policy offers great performance benefits even at relatively high overhead. Our threshold policy still clearly outperformed the optimal static policy even when the overhead associated with reassigning a processor was 50% of the average service time. We derived simple approximate theoretical formulas which allow an easy comparison of the optimal static policy and the threshold policy for various sets of parameters.

References

[1] A.N.Choudhary, B.Narahari, D.M.Nicol, and R.Simha. Optimal processor assignment for pipeline computations. *IEEE Transactions on Parallel and Distributed Systems*, 42(2):1141–1152, 1994.

[2] A. N. Choudhary and J. H. Patel. *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*. Kluwer Academic Publishers, Boston, MA, 1990.

[3] Y-C. Chow and W.H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor systems. *IEEE Transactions on Computers*, (5):662–675, May 1979.

[4] Wen-Hwa Chou Chung-Ta King and Lionel M. Ni. Pipelined data-parallel algorithms: Part II - design. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):486–499, October 1990.

[5] M.E. Dyer and L.G. Proll. On the validity of marginal analysis for allocating servers in M/M/c queues. *Management Science*, 23:1019–22, 1977.

[6] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. *ACM SIGMETRICS Perfomances Evaluation Review*, 16(1):63–72, May 1988.

[7] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[8] A. Ephremides, P. Varaiya, and J. Walrand. A simple dynamic routing problem. *IEEE Transaction on Automatic Control*, AC-25(4):690–693, August 1980.

[9] M. D. Feng and C. K. Yuen. Dynamic load balancing on a distributed system. In *Proceedings of the sixth IEEE Symposium on Parallel and Distributed Processing*, pages 318–325. IEEE Computer Society Press, 1994.

[10] B.L. Fox. Discrete optimization via marginal analysis. *Management Science*, 13:210–16, 1966.

[11] Dario Giarrizzo, Matthias Kaiserwerth, Thomas Wicki, and Robin C. Williamson. High-speed parallel protocol implementation. *Protocols for High-Speed Networks*, pages 165–180, 1989.

[12] L. Kleinrock. *Queueing Theory*. Wiley, 1975.

[13] Ramesh Krishnamurti and Eva Ma. The processor partitioning problem in special-purpose partitionable systems. In *International Conference on Parallel Processing*, volume 1, pages 434–443, 1988.

[14] L.M.Ni and K.Hwang. Optimal load balancing in a multiple processor system with many job classes. *IEEE Transactions on Software Engineering*, SE-11(5):491–496, 1985.

[15] Randolph D. Nelson and Thomas K. Philips. An approximation to the response time for shortest queue routing. *Performance Evaluation Review*, 17(1):181–189, May 1989.

[16] Stephen K. Park, Stephen Harvey, Rex K. Kincaid, and Keith Miller. Alternate server disciplines for mobile-servers on a congested network. In Osman Balci, Ramesh Sharda, and Stavros A. Zenios, editors, *Computer Science and Operations Research: New developments in their interfaces*, pages 105–116. Pergamon Press, first edition, 1992.

[17] Steve Park. Lecture notes on simulation, 1992.

[18] Thomas F. La Porta and Mischa Schwartz. Architectures, features, and implementation of high-speed transport protocols. *IEEE Network Magazine*, 4(2):14–22, May 1991.

[19] M. Schwartz. *Telecommunication Networks*. Addison-Wesley, 1987.

[20] Kenneth C. Sevcik. Characterization of parallelism in applications and their use in scheduling. *Performance Evaluation Review*, 17(1):171–180, May 1989.

[21] Moshe Sidi, Hanoch Levy, and Steve W. Fuhrmann. A queueing network with a single cyclically roving server. In *Queueing Systems 11*, pages 121–144. J.C. Baltzer AG, Scientific Publishing Company, 1992.

[22] Charles V. Stewart and Charles R. Dyer. Scheduling algorithms for PIPE (pipelined image-processing engine). *Journal of Parallel and Distributed Computing*, (5):131–153, 1988.

[23] Hideaki Takagi. Queueing analysis of polling models. *ACM Computing Surveys*, 20(1):5–28, March 1988.

[24] R. Weber. On the optimal assignment of customers to parallel servers. *Journal of Applied Probability*, 15:406–413, 1978.

[25] R. Weber. On the marginal benefit of adding servers to G/G/m queues. *Management Science*, 26:946–51, 1980.

[26] W.E.Leland and T.J.Ott. Load balancing heuristics and process behavior. *ACM SIGMETRICS*, pages 54–69, 1986.

[27] W. Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, 14:181–189, 1977.

[28] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, SE-14(9):1327–1341, 1988.