A REAL-TIME PARALLEL SCHEDULER FOR THE IMPRECISE COMPUTATION MODEL*

HESHAM FOUAD[†], BHAGIRATH NARAHARI[‡], AND JAMES K. HAHN[§]

Abstract. This paper considers the problem of scheduling hard real-time, periodic jobs on a multiprocessor while allowing imprecise computations. A highly dynamic job set is assumed, where limited *a priori* knowledge of a job set's behavior is available. The use of static partitioning schemes for such a job set is shown to lead to load imbalances and unecessary error. Instead, a dynamic load balancing approach is used. A set of constraints are established so that balancing the load does not invalidate established schedules or increase the error. Finally a scheduling algorithm, based on these constraints is presented.

Key words. real-time systems, multiprocessors, scheduling algorithms, imprecise computations.

AMS subject classifications. 68M20, 68N25, 68Q22

1. Introduction. The Imprecise Computation model [2] provides a framework for incorporating graceful degradation in real-time applications. The technique is based on the premise that, in some cases, producing less-that-perfect results on time is better than producing no results at all. This premise is particularly useful for interactive multimedia applications. A degraded image or sound may go unnoticed by users while a discontinuity in the media will most probably not. Devising scheduling algorithms that can effectively manage the real-time generation of synthetic images and sounds in such applications is particularly challenging. Interactive multimedia applications are compute intensive and exhibit a highly dynamic and non-deterministic behavior because of the user's interaction with the application.

Generating high quality synthetic media often requires the use of multiprocessor architectures in order to achieve interactive rates. While parallel real-time scheduling algorithms have been devised for the imprecise computation model, they are not very well suited to this task because they assume a static job set with *a priori* knowledge of the job set's behavior. Partitioning techniques are used where jobs are initially distributed among the processors and uniprocessor scheduling algorithms are then used to schedule each processor independently. Given a highly dynamic job set where the behavior of the jobs is not known *a priori*, these techniques lead to load imbalances and unnecessary degradation.

In this paper we describe a real-time parallel scheduler developed for use in a sound generation server for Virtual Environment (VE) applications. The Virtual Audio Server (VAS) provides three dimensional audio cues for VEs. One of the distinguishing features of VAS is that it supports procedurally defined synthetic sounds. A procedural representation is advantageous because it affords users the maximum flexibility in shaping the sound and, through parameterization of the procedures, sounds can be mapped to motion parameters in the VE.

VAS was developed on a Silicon Graphics Onyx^{tm} machine with eight MIPS^{tm}

*This work was supported by the Naval Research Laboratory under contract N00173-97-P-5529.

[†]The Naval Research Laboratory, Code 5707, 4400 Overlook Drive, Washington DC, 20375 (hesham@acm.org).

[‡]The George Washington University, Department of Electrical Engineering and Computer Science, Washington DC, 20052 (narahari@seas.gwu.edu)

[§]The George Washington University, Department of Electrical Engineering and Computer Science, Washington DC, 20052 (hahn@seas.gwu.edu)

R10000 processors. The Onyx^{tm} utilizes a shared memory architecture that is preferable in this application due the high data bandwidth required. Upon execution, the VAS system commandeers a user-specified number of processors that are isolated and subsequently scheduled by VAS.

2. A Real-Time Sound Server Application. Synthetic sounds are generated by evaluating their procedural representation, once for each sample produced in the output. Given that high quality audio requires sampling rates of 44100 Hz or 48000Hz, this places a significant computational burden on the system. The problem however is compounded because the sound server must effectively create a complete sonic environment composed of many sound sources. Each sound source must be generated and further processed in order to account for propagation effects from source to listener and also for localization effects in real-time.

In order to ensure the real-time response of the sound server given an arbitrary number of active sound sources, the system incorporates a real-time parallel scheduler that manages the sound generation processes. The scheduler creates an instance of a sound evaluation routine for each sound source. An execution of a sound evaluation routine entails generating a sample block of fixed size and processing the samples to simulate propagation and localization effects. A graceful degradation scheme is utilized whereby sounds in the environment are prioritized based on their characteristics and on the position and orientation of the listener. During overload conditions, less important sounds are degraded by under-sampling their representation. In [4] we present a multi-resolution representation of a synthetic sound. Using this technique, a sound generation routine evaluates the sound over a series of ten iterations. Each iteration improves the resolution and hence the quality of the resultant sound. The evaluation process can be stopped at any point past the first iteration and interpolation techniques are used to generate the missing samples.

3. A Real-Time Parallel Scheduler for Generating Sound. The sound generation process can be expressed in terms of the hard real-time, periodic workload model. The job set, $J = J_k$, consists of the set of sounds each making periodic requests for the same execution. Each task $T_{k,j}$ consists of a request for the evaluation of a block of samples. The size of this sample block is fixed across all jobs and thus occupies the same playback time in the output device. If the evaluation routine does not produce a new sample block before the current one has completed playback, silent gaps will appear in the output. The deadline for each of the requests is thus the time for a sample block to playback at the output device. Because the period of a job is defined as the time between two consecutive deadlines, all the jobs will have the same period, namely $\alpha_{k,j+1} - \alpha_{k,j}$ where $\alpha_{k,j}$ is the start time of task $T_{k,j}$.

3.1. The Imprecise Computation model. In a seminal paper, Chung et.al [2] describe a technique for evaluating monotone processes using a model for imprecise computations. A monotone process is one that is guaranteed to produce increasingly accurate results as it is allowed to execute longer. The Imprecise Computation model partitions a task into a mandatory part and an optional part. The mandatory part is that required to produce results at the minimum acceptable precision. The mandatory task set is scheduled as a hard real-time task set and a precise schedule is obtained. Any remaining time is used to schedule the optional parts of the task set. The resultant schedule is termed a feasible schedule.

The scheduling strategy for the Imprecise Computation model requires that each task be executed for a minimum time of m_k and that any remaining time in the sched-

ule be assigned to tasks such that some error metric is minimized. The characteristics of the job set determine the error metric used and thus the scheduling strategy. Jobs are classified according to their characteristics as type N or type C jobs. Type N jobs are those where the error incurred due to a task not completing its execution does not accumulate over time. For type C jobs, on the other hand, errors do accumulate and therefore require that periodically some task be allowed to execute to completion. The sound generation problem fits the characteristics of a type N job. The error incurred due to under-sampling one block of samples does not accumulate to subsequent blocks. We will therefore only consider the scheduling strategies developed for those types of jobs.

The evaluation of a sample block is a monotonic process because the sound's representation can be sampled at successively higher resolutions. We therefore define the minimum execution time m_k for each job to be the time required for the sound generation routine to perform one iteration. The optional job consists of the other nine iterations required to fully evaluate the sound. This, in effect, provides the graceful degradation that we sought to incorporate in the sound server.

Scheduling type N jobs requires two scheduling strategies, one to precisely schedule the mandatory jobs $M_k(p_k, m_k)$, and another to schedule the optional jobs $O_k(p_k, \alpha_k - m_k)$ so that some error metric is minimized. A good error metric for measuring the performance of a scheduling strategy for type N jobs is the average error of all results. An average error formulation for a type N job as described in [2] is

$$E_k = \frac{p_k}{p} \sum_{j=l-\frac{p}{p_k}+1}^{l} \epsilon_k(\alpha_k, j) \tag{3.1a}$$

where p is the least common multiple of the repetition periods of the jobs in J, p_k is the repetition period of job J_k, l is the period in which the error is being calculated, and α_k is the assigned execution time of task j. $\epsilon_k(\alpha_k, j)$ is the error in task j of job k given by

$$\epsilon_k(\alpha_k, j) = 1 - \frac{\alpha_{k,j} - m_k}{\tau_k - m_k} \tag{3.1b}$$

m_k is the minimum acceptable execution time of job J_k, and τ_k is the total execution time of that job.

The total error for the system is expressed as

$$E = \sum_{k=1}^{K} w_k E_k \tag{3.2}$$

where w_k is a normalized, nonnegative constant weight signifying the relative importance of each job. The above general expressions can be simplified to

$$E = 1 - \frac{1}{p} \sum_{k=1}^{K} \frac{w_k}{v_k} \alpha_k(O) \tag{3.3a}$$

v_k is the utilization factor of optional job O_k and is given by $v_k = \frac{(\tau_k - m_k)}{p_k}$. τ is the total processor time assigned to tasks in O_k over the time in which the error is

measured $\frac{p}{p_k}$ and is given by

$$\alpha_k(O) = \sum_{j=l-\frac{p}{p_k}+1}^{l} \alpha_{k,j} - \frac{pm_k}{p_k} \qquad (3.3b)$$

In [2] a two tiered approach is suggested for scheduling type N jobs. The mandatory job set is scheduled using the Rate Monotone algorithm. The Rate Monotone algorithm is a priority driven, preemptive algorithm that has been shown to be optimal among fixed priority algorithms. While the optional job set is scheduled using the Least Utilization (LU) algorithm. When the jobs are ordered in a non-decreasing order in k according to their weighted utilization factor $\frac{v_k}{w_k}$, it is evident from equation 3.3 that E is minimized when the maximum processing time is allocated to O_1 which corresponds to the smallest weighed utilization factor. This observation leads to the Least Utilization algorithm. This algorithm statically assigns higher priorities to the jobs with the smaller weighted utilization factors $\frac{v_k}{w_k}$. It is shown that this algorithm minimizes the average error when the error function is linear and when all the jobs have the same repetition period.

3.2. Multiprocessor Scheduling. Unfortunately, extending the above strategy to the multiprocessor case is not straightforward. According to [3] algorithms for scheduling periodic, hard real-time tasks on multiprocessors have been shown to have unacceptably poor worst-case performance. Alternatively, a partitioning approach is used where jobs are first assigned to processors, each processor is then scheduled independently using uniprocessor scheduling algorithms. The partitioning is optimized so that the number of processors used is minimized. This problem is equivalent to the bin-packing problem and hence is NP hard requiring a heuristic approach.

The Rate-Monotone Next-Fit (RMNF) (or First-Fit (RMFF)) algorithm [3] prioritizes jobs according to their periods, with shorter periods having higher priority. The jobs are assigned to processors in priority order based on a next-fit or alternatively a first-fit basis. In deciding whether or not a job fits a processor, only the mandatory job M_k of J_k is considered. The scheduling strategy for the mandatory job set is invoked to determine whether the inclusion of the newly assigned job to the existing job set on that processor can yield a feasible schedule.

A number of non-partitioning approaches to scheduling periodic hard real-time jobs have been suggested. In [9] the myopic algorithm is presented which can schedule a job set with resource constraints on a multiprocessor. The problem of finding a precise schedule is formulated as search tree and the branch and bound technique is used to search the tree for a precise schedule. The algorithm orders the set of tasks based on a heuristic function H. Possible heuristics for H include: minimum deadline first, minimum processing time first, earliest start time first, minimum laxity first, and finally some weighted combination of the above. Starting with an empty schedule the task with the smallest H value is added to the schedule and a determination is made whether or not the resultant schedule is strongly feasible. A partial schedule is strongly feasible if all the schedules obtained by extending this schedule with one of the remaining tasks are also feasible. When a partial schedule is reached which does not satisfy this constraint, backtracking is performed to traverse other possible branches.

Normally such an algorithm would have $O(n^2)$ complexity. However, as the name implies, the algorithm does not consider all the remaining tasks when evaluating the H value and when determining whether a partial schedule is strongly feasible. Instead

the tasks are ordered in increasing deadlines and only the first k tasks are considered in the evaluation, where k is fixed. The resultant algorithm exhibits $O(n)$ complexity. Surprisingly, this algorithm performs as well as the original $O(n^2)$ version.

In [7] a heuristic based algorithm is presented which assigns m tasks to n processors directly. The SA1 algorithm segments time into blocks that are the GCD of the deadlines of all the tasks. Each task is assigned its average time requirement within each block. The average requirement of each task is $T * \frac{C_i}{D_i}$, where T is the GCD of the deadlines. D_i is the task's deadline and C_i is its required execution time. Assuming processors are numbered $1, \cdots, n$, tasks are assigned their average requirement on the processors sequentially starting with processor 1. When the average requirement for a task exceeds the available time on the current processor, the task is split such that it uses the remaining schedulable time on processor 1 and the remaining execution time required to meet its average requirement is scheduled on the next processor, processor 2. In determining whether a task set m is schedulable on n processors precisely, the condition $U \leq n$ was determined to be sufficient [7], were U is the multiprocessor utilization factor

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \tag{3.4}$$

The SA1 algorithm exhibits $O(m)$ complexity where m is the number of tasks. For a dynamic system with tasks entering and leaving the system on-line, the SA1 algorithm can be run for a newly arrived task in $O(1)$ time if no tasks have left the system and time is available on one processor to schedule the task without partitioning it. Otherwise the full algorithm must be evaluated at $O(m)$ time.

In [6] the SA1 algorithm is extended to handle imprecise computations. The problem is formulated as a min-cost-max-flow network flow problem. While this approach yields a minimum error schedule, the complexity of the approach makes it impractical for any real use.

3.3. Dynamic Load Balancing. We now examine the suitability of a static partitioning scheme for the sound generation problem. Given the transient nature of sounds in an environment, the system exhibits a very dynamic behavior where jobs frequently enter and leave the scheduler. Due to the non-deterministic nature of a user's interactions within a VE, we cannot predict *a priori* the behavior of the sounds in the system. In other words, we cannot predict at any point what sounds may be active.

Utilizing a static partitioning algorithm such as RMNF for the sound evaluation problem has a number of pitfalls. The dynamic nature of the job set leads to load imbalances due to the static assignment of jobs to processors. Situations occur where some processors are idle while others are overloaded and must needlessly degrade sounds. In general, the non-deterministic nature of the job set means that any static partitioning of the job set can only consider the potential load on any processor. This will not reflect the actual load conditions at runtime.

The idea of optimizing the partitions so that the number of processors used is minimized is not realistic for this problem. We are generally assigned a fixed number of processors on which to operate. Our goal is therefore to minimize the error given a fixed number of processors. The RMNF algorithm does not consider the resultant error in the imprecise computation when assigning jobs to processors since only the mandatory job set is considered.

Finally, RMNF assigns jobs to processors based on their period. As was described earlier, all of the jobs in our workload model have the same period. These algorithms therefore lead to an arbitrary assignment of jobs to processors.

If static analysis of the job set will not lead to a good partitioning, then we must consider a dynamic approach. A dynamic load-balancing scheme is concerned with distributing jobs among the processors in a parallel or distributed system such that the load is evenly balanced. While it is generally agreed upon that maintaining a balanced load across processors is beneficial, in this model balancing the load can actually result in an increase in the average error. The following theorem relates processor load to the average error.

THEOREM 3.1. *When the total utilization factor of all jobs $U > 1$, a partitioning which assigns each of N processors a load of $\frac{U}{N}$ is optimal if and only if $U \leq N$. We refer to optimality here in the sense that no other assignment of load to processors will produce less error.*

Proof. In order to prove this theorem, we consider two cases of U:

Case $U \leq N$: Here a balanced partitioning will assign each processor a load of $\frac{U}{N} \leq 1$

It has been shown in [8] that $u \leq 1$, where u is the utilization factor of the jobs assigned to one processor, is sufficient to guarantee that the job set is precisely schedulable. Therefore under this partitioning all the optional jobs will be assigned their full execution time and there is no error. Clearly no other partitioning can produce less error.

Case $U > N$: The expressions for average error presented in equations 3.1 and 3.2 can be reformulated in terms of a job's unassigned execution time as follows

$$E = \frac{1}{p} \sum_{k=1}^{K} \frac{w_k}{v_k} \epsilon_k(O) \tag{3.5a}$$

where

$$\epsilon_k(O) = \sum_{j=l-\frac{p}{p_k}+1}^{l} (\tau_k - m_k) - (\alpha_{k,j} - m_k) \tag{3.5b}$$

and $\epsilon_k(O)$ is the portion of k's optional job that is not executed.

Suppose that we have an optimal partitioning of the job set that allocates a load of $\frac{U}{N}$ to each processor, we should expect that the average error be minimized. If this supposition is correct, then moving some job J_i from processor P_j to P_k will result in a load imbalance and hence an increase the average error. We can formulate the difference in average error resulting from moving job J_i from processor P_j to P_k as follows

$$\Delta E = \frac{1}{p} \left(\left(\frac{w_i}{v_i} \epsilon'_i(O) + \sum_{k=1}^{K} \frac{w_k}{v_k} \Delta T_k \right) - \left(\frac{w_i}{v_i} \epsilon_i(O) + \sum_{j=1}^{J} \frac{w_j}{v_j} \Delta T_j \right) \right) \tag{3.6}$$

where w_i is the weight of job J_i, v_i is the total utilization factor of O_i, $\epsilon_i(O)$ is the error due to J_i's assigned time on processor P_j (equation 3.5), $\epsilon'_i(O)$ is the error due to J_i's new assigned time on processor P_k (equation 3.5), ΔT_k is the difference in assigned time to job k after adding J_i to P_k, and finally ΔT_j is the difference in

assigned time to job j after removing J_i from P_j. We now consider the following job sets assigned to two processors P_1 and P_2 respectively:

$J_1 = (2, .5, .2), (4, .2, .3), (5, .1, .1), (6, 2, .4)$

$J_2 = (2, .2, .8), (7, .1, .1), (8, .1, .1)$

Each job is specified as a tuple (τ, m, w), consisting of the job's total execution time, its minimum acceptable execution time, and its weight respectively. The total utilization factor on each processor is 1.7 and the load is ideally balanced. According to equation 3.6, moving job $j_2(4, .2, .3)$ from P_1 to P_2 results in a decrease in average error of the jobs assigned to P_1 of .20369. The corresponding increase in average error that results from adding j_2 to P_2's job set is .056314. The net change in error due to this migration is $\Delta E = .056314 - .20369 = -.14738$. The total average error was decreased due to the job migration and hence we have a contradiction. □

3.4. The Minimum Distance Algorithm. A heuristic load-balancing algorithm was devised and incorporated into our scheduler. Due to the shared memory architecture chosen for this problem, the algorithm is able to use global load information across all processors, and use a preemptive migration policy. The following policies were defined for task allocation and task migration. The task allocation policy is based on the random allocation presented in [1]. This is based on the observation that client applications typically allocate all sounds during an initialization stage. At runtime sounds are dynamically started and stopped, and finally before exiting, sounds are deallocated. At task allocation time, when new sounds are created, no sounds are yet active and hence no actual load information exists to base a task allocation on. The static load information is of little use since it does not reflect the actual load during runtime. Finally as was shown in [1], the random allocation policy does a good job of distributing the load for a uniform grain size. While the sound evaluation routines can differ in grain size based on the complexity of the sound, experience has shown that the amount of variation they exhibit is fairly limited.

The task migration policy chosen is a sender initiated preemptory policy that affords better response to overload conditions [5]. In order to determine when a processor is to be considered overloaded, we use the total utilization factor of the mandatory and optional jobs assigned to that processor. The threshold used for initiating the send procedure is $u_p > 1$ where up the total utilization factor assigned to processor P.

We have not yet considered when load balancing should be initiated. A static invocation policy is one that is synchronous; load balancing occurs at regular predetermined intervals. A dynamic invocation policy is asynchronous such that load balancing can occur at any time. A dynamic invocation policy is more responsive and can reduce scheduling overhead since load balancing is only initiated when needed. Unfortunately, a dynamic policy can lead to serious problems for a real-time scheduler. Once each processor's scheduler has determined an allocation of time to its assigned jobs for a given period, migrating a job during that period will invalidate the schedules. A dynamic invocation policy is therefore not appropriate for a real-time system. Instead a static invocation policy is used where load balancing is initiated at the start of each period. The overhead incurred due to this policy is actually minimal because in highly dynamic sonic environments load imbalances may occur frequently enough so that a dynamic policy would invoke load balancing as often as a static policy, in which case both algorithms would impose the same overhead. In other cases, the overhead is minimized by only initiating load balancing when at least one of the processors meets the task migration threshold.

The determination of where to send a job and which job to send is made as follows: The algorithm first finds the least utilized processor P_{least}. A job is migrated from processor P_{from} to processor P_{least} if there exists a job j on P_{from} that satisfies the following two constraints: In order to insure that the migration will reduce the difference in load between the two processors we have $u_j < u_p - u_{p_least}$. u_j is the total utilization factor of job j, and u_{p_least} is the utilization factor of the least utilized processor. The second constraint is based on theorem 3.1, which shows that migrating jobs in order to balance load might increase the average error. In general, equation 3.6 gives the exact change in error, ΔE, due migrating a job between two processors. Ideally we would like to guarantee that a given job migration will not only result in a negative ΔE but also maximize $|\Delta E|$. Unfortunately an exact solution requires that for each job that is being considered for migration, the post-migration schedule on P_1 and P_2 be generated in order to evaluate ΔE. The resulting overhead makes that an unreasonable approach. Instead we must resort to using a heuristic to determine which job, if any, should be migrated.

In order to examine the effect that our choice of job has on ΔE, we reformulate equation 3.6 to the following

$$\Delta E = \frac{w_i}{v_i}(\alpha - \alpha') + \sum_{k=1}^{K} \frac{w_k}{v_k} \Delta T_k - \sum_{j=1}^{J} \frac{w_j}{v_j} \Delta T_j \tag{3.7}$$

Our choice of job will determine the value of $\frac{w_i}{v_i}$ and the value of α, the current execution time. In order to make $\frac{w_i}{v_i}(\alpha - \alpha')$ as negative as possible, we must choose a job such that $\frac{w_i}{v_i}$ is maximized, α is minimized, and α', the new execution time, is maximized. Based on how execution time is assigned to optional jobs in the LU algorithm, there exists an integer f such that: Assuming the job set is ordered by the weighted utilization factor $\frac{v_k}{w_k}$ such that it is non-decreasing in k, $J = j_1, j_2, \cdots, j_k$, an optimal assignment will allocate the first f optional jobs in O_k their full execution time. More formally, we have

$$\sum_{k=1}^{f} v_k \leq 1 - u < \sum_{k=1}^{f+1} v_k \tag{3.8}$$

where u is the utilization factor of the K mandatory jobs, and v_k is the full utilization factor of an optional job [2]. In choosing a candidate job on P_{from} to migrate, we find a job j_i such that $i > f_{from}$. In other words, we choose the job with the least weighted utilization factor that is not receiving its full execution time. In order to guarantee that job j_i will receive more execution time after migration, we must ensure that on the target processor $\frac{v_i}{w_i} < \frac{v_f}{w_f}$.

Based on these observations, our load balancing algorithm finds the first job j_i on P_{from} such that $i > f$ on the source processor and $u_i < u_{p_from} - u_{p_least}$. Job j_i is migrated if and only if $\frac{v_i}{w_i} < \frac{v_f}{w_f}$ on the target processor. Because the algorithm tries to minimize the difference in load between any two processors, we termed the algorithm the Minimum Difference (MD) algorithm. Pseudo-code for the MD algorithm is listed in Fig. 3.1.

The MD algorithm was incorporated into our scheduler and processor loads were measured during the evaluation of a test sonic environment on two processors. In Fig. 3.2 the load imposed by this test environment is plotted. In Fig. 3.3 the load assigned to each processor with the MD algorithm and a static allocation policy are

```
MD()
begin
P = {p1, p2, …, pN}                 /* P is the set of N processors */
for i:= 1 to N do
   p := P(i)
   if p.utilization > 1 then
         pmin = FindMinUtilized(P)          /* Returns the least utilized processor */
         if pmin = p then next i
         else
               u := p.utilization – pmin.utilization
               /* GetJobByUtilization finds a job with the least vj/wj */
               /* where i > f and with a total utilization ≤ u */
               j := GetJobByUtilization(p, u)
               if (j <> null) and (vj/wj < vf/wf) then
                     Migrate(j, p, pmin)    /* Migrates a job from p to pmin */
               endif
         endif
   endif
endfor
end
```

FIG. 3.1. *The Minimum Distance algorithm*

depicted. The static strategy that was used simply allocates an incoming job to the processor with the least assigned load. Also plotted on Fig. 3.3 is the optimal load assignment of $\frac{U}{N}$.

In order to measure the effectiveness of these two strategies, the mean of the difference between the load assigned to the two processors was measured. Clearly a mean value of zero implies that both processors received identical load and hence we have the optimal assignment of $\frac{U}{N}$. The mean value for the static allocation policy was found to be 0.8. The corresponding mean value for the dynamic allocation was 0.2. The dynamic policy clearly balanced the load much more effectively than the static policy. This is also evident by examination of the plots in Fig. 3.3. The processor

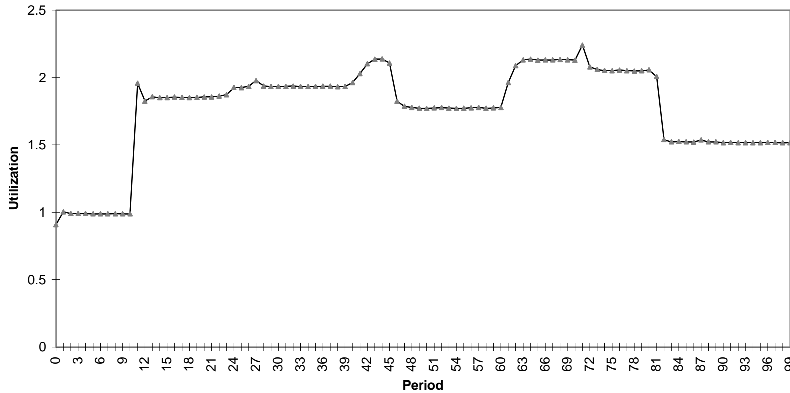

FIG. 3.2. *Processor utilization*

loads for the dynamic allocation policy were much closer to the optimal case than those for the static policy.

The effect of the load assignments on error was measured by calculating the total average error produced by both policies. The total average error was calculated simply as the sum of the average error produced on each processor over a time of 100 periods. The total average error produced by the static allocation strategy was 0.74046. The corresponding error generated using the dynamic task migration was 0.609861. As expected the MD algorithm balanced the load while significantly reducing the average error.

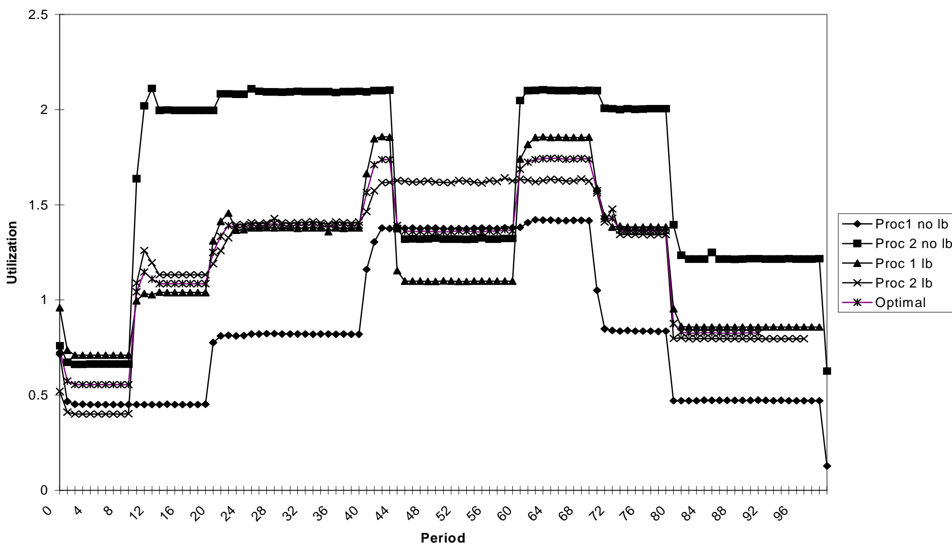

Fig. 3.3. *Processor loads with and without dynamic load balancing*

4. Conclusion. In this paper, we presented a parallel real-time scheduling algorithm for the Imprecise Computation model that utilizes dynamic load balancing. We have shown that for applications where the job set is dynamic and the behavior of jobs is not predictable, a static partitioning will lead to load imbalances and unnecessary degradation. We established a set of constraints that must be adhered to in utilizing dynamic load balancing for a real-time system such that schedules are not invalidated and the error is not increased. Finally we presented the MD algorithm which adheres to these constraints and was shown to balance the load among processors while decreasing the average error.

REFERENCES

[1] W. C. Athas and C. L. Seitz, *Multicomputers: Message-passing concurrent computers*, IEEE Computer, 8 (1988), pp. 9–24.

[2] J. Y. Chung, J. W. S Liu, and K. J. Lin, *Scheduling Real-time, Periodic Jobs Using Imprecise Results*, Proc. IEEE RTS, (1987), pp. 252–260.

[3] S. K. Dhall and C. L. Liu, *On a Real-Time Scheduling Problem*, Oper. Res., 1 (1978), pp. 127–140.

[4] H. Fouad, J. K. Hahn, and J. A. Ballas, *Perceptually Based Scheduling Algorithms for Real-time Synthesis of Complex Sonic Environments*, Proc. ICAD, (1997), pp. 77–81.

[5] D. L. Eager, E. D. Lazowska, and J. Zahorjan, *A Comparison of Receiver-Initiated and Sender- Initiated Adaptive Load Sharing*, Performance Evaluation, 1 (1986), pp. 53–68.

[6] A. Khemka, R. K. Shyamasundar, *Multiprocessor Scheduling for Imprecise Computations in a Hard Real-time Environment*, Proc. IEEE RTS, (1993), pp. 617–636.

[7] A. Khemka, R. K. Shyamasundar, *Multiprocessor Scheduling of Periodic Tasks in a Hard Real-time Environment*, Int. Journal of High Speed Computing, 4 (1993), pp. 617–636.

[8] C. L. Liu, J. W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment*, Journal of the ACM, 1 (1973), pp. 46–61.

[9] K. Ramamritham, J. A. Stankovich, and P. F. Shiah, *Efficient Scheduling Algorithms for Real-time Multiprocessor Systems*, IEEE Trans. on Parallel and Distributed Sys., 2 (1990), pp. 184–194.