AN EXPERIMENTAL COMPARISON OF INITIAL PLACEMENT VS. PROCESS MIGRATION FOR LOAD BALANCING STRATEGIES

Kriton Kyrimis
Rafael Alonso

CS-TR-199-88

December 1988

An Experimental Comparison of Initial Placement vs. Process Migration for Load Balancing Strategies

Kriton Kyrimis
Rafael Alonso

Department of Computer Science
Princeton University

ABSTRACT

In this paper we describe an experimental comparison between initial placement and process migration to determine which is the better method to use to implement a load balancing system. A trace-based synthetic workload is described, which allows us to control the role that implementation overhead plays in such a system, and the two methods are compared for varying values of this overhead, from significant to negligible.

Index terms: process migration, load balancing, scheduling, distributed systems, local area networks, performance analysis.

1. Introduction

In a distributed computing environment it is often desirable to balance the system-wide load in such a way that we do not have a situation where some machines are busy while others are relatively idle. In a distributed system, a *load balancing* strategy is used to achieve an even distribution of the work. Such strategies may be preemptive or not. A preemptive implementation is usually said to be able to carry out *process migration*, while non-preemptive policies are referred to as *initial placement*. With initial placement, when a job arrives at a machine, the system determines which is the best machine to run the new job so that the load balance is maintained, and starts that job there. With process migration, each job is run on the machine where it is initiated and, at an arbitrary point in time, the system can decide to move that job to another machine so that the system-wide load can remain balanced.

Determining which of the two methods is the one that should be preferred for an actual implementation is a very controversial subject, as there are arguments in the literature in favour of both:

Cabrera [1] argues in favour of process migration by studying system traces. These traces indicate that most processes have a very small lifetime, and an initial placement algorithm would be of little benefit, as all these processes would be slowed down by the

This research was supported by New Jersey Governor's Commission Award No. 85-990660-6, and grants from IBM and SRI's Sarnoff Laboratory.

overhead of initial placement. On the other hand, it appears that implementing a process migration algorithm would be desirable, as the overhead would only have to be paid by the relatively few large jobs. It is therefore argued that one should implement a load balancing scheme that determines long-lived jobs and migrates them to another machine if doing so will improve the system-wide load balance.

Eager, Lazowska and Zahorjan [2] have developed an analytical model which tries to determine the maximum potential benefits of initial placement and process migration by assuming that the implementation overhead is zero. They conclude that process migration will not yield major performance improvements over initial placement, and that the latter is an inherently better strategy. Only under extreme conditions can process migration offer performance improvements, and these are expected to be modest. If implementation overhead for process migration is taken into account, the results remain the same, as it appears that it does not affect performance significantly.

By studying system traces, Leland and Ott [3] have determined that either initial placement or process migration can produce significant results and that running the two algorithms together should produce an even greater improvement in performance.

Finally, by studying mathematical curves derived from [3], Krueger and Livny [4] reach a similar conclusion, in that process migration in addition to initial placement can provide a significant performance improvement.

It is apparent from the above discussion that the answer to the question of how does process migration compare to initial placement as a load balancing strategy is not at all obvious, and that an experimental comparison of the two can be of some interest.

2. Implementation environment

We ran our experiments on seven Sun 2 workstation running version 3.3 of Sun O.S., which is a derivative of the Berkeley 4.2 BSD version of the UNIX† operating system. The machines were connected to each other and a file server by a 10 Mbit/sec Ethernet.‡

For the initial placement experiments we used the implementation described in [5]. By using a special command line processor new jobs are placed at the least loaded machine in the network. This is achieved by running two background jobs on each machine: The first one broadcasts and collects load information periodically. The command line processor can query it to determine which is the least loaded machine, so that it can start the new job there. If it is determined that the job must be run remotely, the command line processor connects to the second background process, running on the remote machine. The background process then starts the new job locally, establishing a connection between the job and the terminal on the machine where it was initiated. The load metric used is the system load as provided by UNIX, which is the average number of jobs in the run queue. For our particular experiment only load differences greater than one were considered (i.e., a job was only executed remotely if the difference between the load of the local machine and that of the least loaded machine in the network was at least one).

† UNIX is a registered trademark of AT&T.

‡ Ethernet is a trademark of XEROX Corporation.

The process migration experiments were run on a load balancing system built on the process migration implementation described in [6]. The implementation consists of a modified 3.3 Sun O.S. that can terminate a process dumping on disc all the information that is necessary to restart it, as well as restart a process from such a dump. Process migration is achieved by restarting the terminated process on another machine, the entire procedure being automated by use of user-level programs. The main limitation of this system is that it can only migrate successfully processes that do not have subprocesses, and that do not have any residual dependencies (e.g., temporary files with the process-id as part of their name). This modified kernel was only used for the experiments involving process migration.

For the load balancing experiments, we used two background processes. The first process is the process that broadcasts and collects load information that was used in the initial placement system. The second process runs once a minute, checking the system load. If the load is greater than one, it looks through the list of processes running on the local machine. If it finds a process that has consumed more than a certain amount of CPU time, it moves that process to the least loaded machine.

3. Workload

To make a load balancing experiment, a workload must be used that is as realistic as possible. The ideal situation would be to run the experiment on a working system and measure the changes in performance. Unfortunately, this was not possible in our case, because the system that was available to us is used by very few people, thus having a very low actual workload, and because of limitations in our implementations: In the initial placement system, the command line processor that does the initial placement is not interactive, and has to be invoked as a command each time a job is to be started, thus making it difficult to replace the standard command line processor and give it to the users, and the process migration mechanism cannot migrate arbitrary jobs. For these reasons, it was decided that a synthetic workload should be used.

Krueger and Livny [4], by applying weighted least squares analysis on the data collected by Leland and Ott [3], have determined that a process' service demand in CPU seconds has the following probability density function:

$$f(x) = .79(e^{-x/.31}/.31) + .192(e^{-x/2.8}/2.8) + .018(e^{-x/27}/27)$$

which, for a mean service demand $\bar{X}$ can be generalised to:

$$f(x) = (.79/.243\bar{X})e^{-x/.243\bar{X}} + (.192/.22\bar{X})e^{-x/.22\bar{X}} + (.018/21.2\bar{X})e^{-x/21.22\bar{X}}$$

with $\bar{X} = 1.27$. By integrating from 0 to $+\infty$, this gives a distribution function

$$F(x) = 1 - 0.79e^{-x/0.243\bar{X}} - 0.192e^{-x/2.2\bar{X}} - 0.018e^{-x/21.2\bar{X}}$$

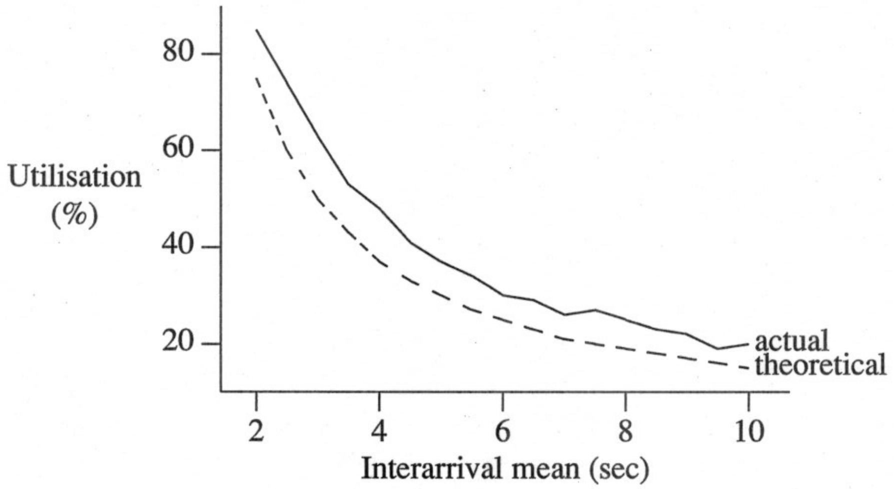
which we can use to create random values obeying this distribution, for the program that creates the synthetic workload.

By selecting different values of $\bar{X}$, it is possible to study the role that the overhead of the load balancing implementation plays. This overhead is not a quantity that can be varied under real conditions, but this approach allows us to treat it as such. A low value for $\bar{X}$ would make the relatively high overhead of our implementation comparable to the turnaround time of the average job in the system (thus making the overhead time a considerable factor in the experiment). On the other hand, a high value for $\bar{X}$ would make the overhead small compared to the running time of the average job in the system (thus, the overhead becomes negligible).

To generate the workload we also need a function that will determine the interarrival time between jobs. For this we chose a simple exponential distribution:

$$f(x) = 1/t_m \, e^{-x/t_m}, \qquad F(x) = 1 - e^{-x/t_m}$$

where t_m is is the mean interarrival time. By selecting different values for t_m we can produce different utilisations for the machine. The utilisation can either be determined by the simple formula $utilisation = \bar{X}/t_m$ or, as was done in our case, by running the workload with various values of $\bar{X}$ and t_m and measuring the results (this was done using a monitoring program that computed the CPU utilisation by reading the *cp_time* variable of the 3.3 Sun O.S. kernel, in a manner similar to that of the *vmstat* [7] utility). The latter method is somewhat more accurate, as the utilisation can be affected by system overhead that is not taken into account by the simple formula. A sample graph, illustrating the difference between the two utilisation estimates is given in Figure 1.



Theoretical and actual CPU utilisation
as a function of the interarrival mean for $\bar{X} = 1.5$.

Figure 1

Finally, to complete our workload model, we need to develop some notion of a machine being idle for some of the time because, if all the machines are busy throughout the experiment and are running a similar workload, the best load balancing policy will be to do nothing, which will not provide us with any new insights. For this experiment we simply chose to run the synthetic workload on five of our machines (with a different seed for the random number generator so that there is some variance in the workload), and to keep the other two of our seven machines idle, serving simply as cycle servers. Mutka and Livny [8], by studying traces measured on workstations, have derived distributions for the time a machine is available and not available (i.e., being used or having recently been used). As an alternative, we could have used these distributions, only generating workload during the period that a machine is not available. Since these distributions were derived on systems different from those that the workload distributions were derived (the idle time distribution was measured on workstations while the CPU time distribution was derived from the workload of a VAX mainframe), we considered that it was inappropriate to mix the two.

4. The experiments

As described in the previous section, by varying the mean service demand $\bar{X}$, it is possible to simulate the effects of varying the importance of the overhead of the load balancing implementation. We used three different values: 1.5, 5 and 5 seconds. These are arbitrary values, with the first being similar to the value of 1.27 seconds reported in [4], the second being comparable to the measured overhead of our implementation†, and the third being an order of magnitude higher. For the mean interarrival time t_m, we chose three values that produced an average CPU utilisation of 25, 50 and 75 per cent. By combining these two sets of values, we produced nine different experiments, each of which was run once without any load balancing policy, so that we could obtain the original workload, once with the initial placement policy and once with the process migration policy, for a total of 27 experiments. Each of these 27 experiments was ran a number of times, so that we could obtain statistically significant results.

As described in Section 2, in the experiments involving process migration we migrate jobs that have consumed more than a certain amount of CPU time. For $\bar{X}=1.5$, we chose to migrate jobs that have consumed more than 16 seconds of CPU time, which resulted in making only about 2 per cent of the total jobs generated candidates for process migration. For $\bar{X}=5$ and 25, this time was scaled to 53 and 266 seconds respectively. For $\bar{X}=1.5$ and 5, each experiment was run for 20 minutes and for $\bar{X}=25$, each experiment was run for one hour so that we could collect sufficient data.

The results that we obtained are given in tabular form in Table 1 as mean job response time and standard deviation as a percentage of the mean. The results are also presented in graph form in Figures 2, 3 and 4, where the standard deviation is omitted for clarity.

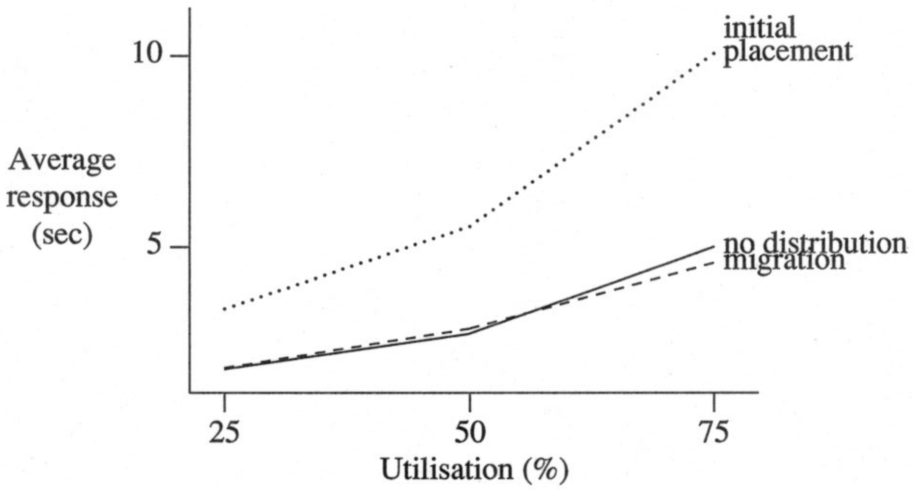
The experiments for $\bar{X}=1.5$ show that if the implementation overhead is considerable, instead of improving the average job response, initial placement actually makes it worse. On the other hand, the process migration policy produces results similar to the original workload, even producing a marginal improvement when the CPU utilisation is high. This can be explained by noticing that in the initial placement implementation, each job that arrives at the system is slowed down by the significant overhead (note that this contradicts the remark in [2] that results based on a model that assumes zero overhead are biased in favour of process migration). Jobs that are started locally are affected less, since they only have to be delayed until it is determined that they can be executed locally, but jobs that are executed remotely are further delayed by the overhead of the

† The overhead of executing a job remotely in our initial placement implementation is in the order of 3 seconds [5], while the overhead of migrating a job to a remote machine in our process migration implementation is in the order of 6 seconds [6].

High implementation overhead ($\bar{X} = 1.5$sec)			
Utilisation	No Distribution	Initial Placement	Migration
25%	1.83±0.80%	3.38±2.49%	1.86±0.22%
50%	2.75±1.66%	5.53±1.04%	2.88±0.62%
75%	5.01±1.77%	10.04±0.56%	4.59±0.09%
Medium implementation overhead ($\bar{X} = 5$sec)			
Utilisation	No Distribution	Initial Placement	Migration
25%	6.75±0.76%	7.57±0.99%	6.97±1.06%
50%	6.74±1.01%	7.76±1.23%	6.64±0.05%
75%	10.39±1.54%	10.33±3.32%	9.51±0.77%
Low implementation overhead ($\bar{X} = 25$sec)			
Utilisation	No Distribution	Initial Placement	Migration
25%	32.30±0.44%	27,79±0.72%	30.31±3.50%
50%	37.22±0.30%	31.16±1.03%	32.47±0.79%
75%	42.82±0.67%	30.36±1.69%	32.19±0.72%

Average job response times for $\bar{X} = 1.5, 5, 25$
and *utilisation* $= 25, 50, 75$ per cent

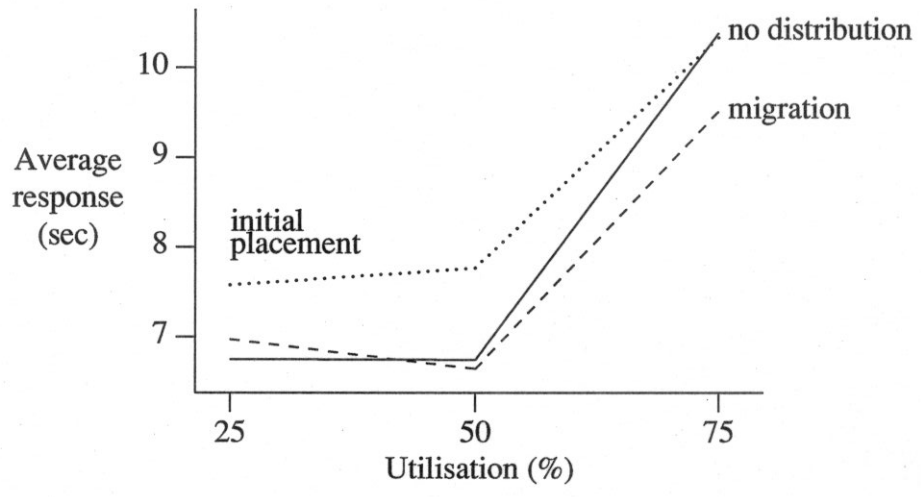
Table 1



Average response for $\bar{X} = 1.5$

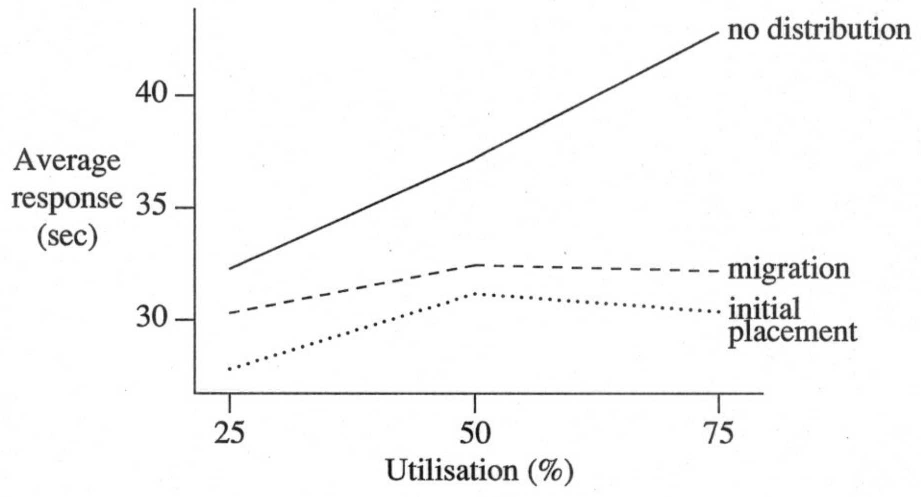
Figure 2

remote execution. Since the probability that the job will run remotely is high (once load balance has been achieved, the probability of running a job remotely is $(n-1)/n$, where n is the number of machines in the network), most jobs will have to be delayed by the remote execution overhead. On the other hand, in the process migration implementation



Average response for $\overline{X} = 5$

Figure 3



Average response for $\overline{X} = 25$

Figure 4

there are very few processes that are affected, so the overhead does not play such a significant role, and most processes run as if the load balancing mechanism did not exist. When the CPU utilisation is high, it is even possible to improve performance slightly, despite the large overhead of moving a job to another machine.

The experiments for $\overline{X} = 5$ show that if there is an implementation overhead, but it has a value comparable to the mean service demand, we get similar results as before, except for the case where CPU utilisation is high. In this case, initial placement produces similar results to doing load balancing. Process migration, however, produces a noticeable improvement.

The situation changes entirely when the implementation overhead becomes of small significance ($\bar{X}=25$). Now, both initial placement and process migration improve the average job response significantly, but this time, as suggested in [2], there is little difference between initial placement and process migration. (Actually, our results show that initial placement performs *better* than process migration, due to the high overhead of our migration implementation.)

5. Conclusions

We have conducted load balancing experiments comparing initial placement and process process migration as the load balancing strategy. These experiments took into account the role that the implementation overhead and the CPU utilisation plays in the performance of the two algorithms. We have found that Eager, Lazowska and Zahorjan [2] are correct in preferring initial placement, but only under the assumption that the implementation overhead is insignificant. In most cases there is definitely going to be some overhead, and this time our results show that process migration may be better in this case, especially if the CPU utilisation is taken into account. If the implementation overhead is too high, it is probably not worth running either of the two load balancing schemes, as the average job performance will either deteriorate or remain unaffected. On the other hand, with a reasonably good implementation it appears that process migration can help improve job performance when a machine is heavily utilised. Our experiments suggest that the recommended policy is to do nothing while the system load is low to medium, and run the load balancing algorithm that uses process migration when the system load gets high. Our results also show that even if this algorithm is run all the time (i.e., regardless of the system load), there is not going to be any significant degradation. Finally, even when the implementation overhead is very high, our results suggest that when the system load is high, there may be some small improvement in performance if load balancing is used in conjunction with process migration.

References

1. Luis-Felipe Cabrera, ''The Influence of Workload on Load Balancing Strategies,'' in *Proceedings of the Summer 1986 Usenix Conference*, pp. 446-458, Atlanta, Georgia, 1986.
2. Derek L. Eager, Edward D. Lazowska, and John Zahorjan, ''The Limited Performance Benefits of Migrating Active Processes for Load Sharing,'' Technical report 87-12-10, University of Washington, Department of Computer Science, December 1987.
3. Will E. Leland and Teunis J. Ott, ''Load-balancing Heuristics and Process Behavior,'' in *Proceedings of PERFORMANCE '86 and ACM SIGMETRICS 1986*, pp. 54-69, May 1986.
4. Phillip Krueger and Miron Livny, ''A Comparison of Preemptive and Non-Preemptive Load Distributing,'' in *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 123-130, June 1988.

5. Rafael Alonso, Phillip Goldman, and Peter Potrebic, ''A Load Balancing Implementation for a Local Area Network of Workstations,'' in *Proceedings of the IEEE Workstation Technology and Systems Conference*, pp. 118-124, March 1986.

6. Rafael Alonso and Kriton Kyrimis, ''A Process Migration Implementation for a UNIX System.,'' in *Proceedings of the Winter 1988 Usenix Conference*, pp. 365-372, Dallas, Texas, February 1988.

7. Sun Microsystems Inc., *UNIX Commands Reference Manual*, 1986.

8. Matt W. Mutka and Miron Livny, ''Profiling Workstations' Available Capacity for Remote Execution,'' Computer Sciences Technical Report #697, University of Wisconsin-Madison, May 1987.