

# Evaluation of the Memory Page Migration Influence in the System Performance: The case of the SGI O2000

Julita Corbalan  
DAC-UPC  
Jordi Girona 1-3  
08034 Barcelona  
+34 934015956  
juli@ac.upc.es

Xavier Martorell  
DAC-UPC  
Jordi Girona 1-3  
08034 Barcelona  
+34 934017190  
xavim@ac.upc.es

Jesus Labarta  
DAC-UPC  
Jordi Girona 1-3  
08034 Barcelona  
+34 934016987  
jesus@ac.upc.es

## ABSTRACT

Current shared-memory multiprocessor CC-NUMA architectures provide a global address space to applications by hardware. However, even though the memory is virtually shared, it is actually physically distributed. Since memory nodes are distributed across the system, the cost of the memory accesses depends on the distance between the node that accesses the data and the node that physically contains the data. To reduce the impact of a bad initial memory placement, some operating systems offer a dynamic memory migration mechanism.

In this paper, we want to demonstrate that memory migration mechanisms are a useful approach, but that their performance depends more on related issues, such as the processor scheduling, than on the mechanism itself. To show that, we evaluate the case of the automatic memory migration mechanism provided by IRIX, in Origin systems.

We have evaluated several workloads of OpenMP applications under different system conditions such as the processor scheduling policy or the system load. In particular, we have focused on the effects of the page migration mechanism on the CPU time consumed by each application, the processor allocation received, and the speedup, when applying performance-driven scheduling policies.

Results show that, if the scheduler is memory conscious, that is, it maintains as much as possible the system stable, the automatic memory page migration mechanism provided by IRIX will improve the execution time of OpenMP applications. Experiments also show that the combination of performance-driven policies and the memory migration mechanism results in a system that can be automatically self-evaluated and self-configured.

## Categories and Subject Descriptors

D.4.1 [Process Management]: Multiprocessing/multiprogramming /multitasking, Scheduling, Synchronization, Threads. D 4.8 [Performance]: Measurements. Modeling and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'03, June 23-26, 2003, San Francisco, California, USA.  
Copyright 2003 ACM 1-58113-733-8/03/0006...\$5.00.

prediction.

## General Terms

Algorithms, Measurement, Performance, Experimentation.

## Keywords

Operating systems, Memory page migrations, performance evaluation, scheduling algorithms.

## 1. INTRODUCTION

Current dominant shared-memory multiprocessors are the CC-NUMA (Cache-Coherent Non-Uniform Memory Access) architectures [8]. This kind of multiprocessors provides scalability and a transparent access to local and remote memory. However, the access to the remote memory has a major drawback: The latency to access to remote nodes depends on the distance between nodes.

Given the large remote access time, current operating systems provide support for dynamic page migration and replication in order to improve data locality. Dynamic page migration is a mechanism that provides adaptive memory locality for applications. Page replication is a mechanism that allows the existence of several copies of the same memory page in different nodes. The replication is automatically applied to read-only pages like the kernel code.

Several previous works have evaluated the usefulness of the dynamic page migration mechanism and some of them have concluded that it is not a valid approach. However, most of them were based on simulations, with a small number of processors, and evaluating individual applications. Moreover, most of the previous works that consider the execution of workloads of parallel jobs do not take into account the influence of the scheduling policy.

In this work, we want to evaluate the influence of the dynamic page migration mechanism when executing workloads of OpenMP scientific applications under different processor scheduler characteristics. We want also to find the effects of using memory page migrations in combination with a performance-driven scheduling policy.

The goal is to demonstrate that the automatic migration mechanism is a valid approach if the processor scheduling is memory conscious, that is, it maintains as much as possible the mapping between processors and processes. We want to show that

there is no need to spend large amounts of time to perform an explicit data distribution when executing parallel OpenMP applications, such as with MPI jobs, since the migration mechanism can dynamically adjust the memory mapping to the job behaviour.

To show that, we have evaluated the case of the memory migration mechanism provided by IRIX in Origin systems.

Experiments have been carried out in a real system, a SGI Origin 2000 with 64 processors. We have executed several workloads of OpenMP [14] parallel jobs. Scheduling policies different than the native IRIX scheduler have been executed in the NANOS execution environment (NANOS-EE) [12][10]. The main goal of the NANOS-EE is to provide an efficient execution environment to parallel applications in shared-memory multiprogrammed multiprocessor systems.

Results show that the automatic memory migration mechanism provided by IRIX is very useful if the scheduling policy maintains stable the processor allocation as much as possible. This enforces our conviction that different components of the system must cooperate to achieve an overall performance and that the system can be automatically self-tuned. Results also show that performance-driven scheduling policies react better to changes in the system conditions and are more robust than those policies that do not consider the job characteristics. This robustness introduces a predictable behaviour, which is a desirable property.

The rest of this paper is organized as follows: Section 2 presents the related work. Section 3 describes the dynamic memory page migration mechanism provided by IRIX 6.5. Section 4 presents the main characteristics of the NANOS-EE. Section 5 presents the scheduling policies evaluated in this work. Section 6 evaluates the effects of using the memory page migration mechanism in scientific workloads, and finally Section 7 presents the conclusions of the work.

## 2. RELATED WORK

Several works have been conducted on CC-NUMA machines to improve the memory behavior. They range from O.S. works, implementing the memory migration policies to user-level memory migration packages oriented to detect which pages should be migrated and do it periodically.

In [2], Bretch shows that memory placement becomes critical in NUMA multiprocessors to obtain overall system performance.

In [3] authors implement page migration in the DASH machine, running the IRIX operating system. They perform several experiments with workloads of parallel applications. They compare several scheduling policies (gang scheduling, processor sets and process control) with and without page migration. They conclude that without major modifications to the IRIX virtual memory system, there is no benefit of using page migration because of the excessive locking in the operating system. In our work, we are using IRIX 6.5, where these problems are already solved and the migration mechanism is much more efficient.

In [9] authors work with simple page placement policies to demonstrate that the initial placement of a parallel application is important to achieve good performance.

Jiang and Singh show in [6] that turning on dynamic page migrations on the Origin2000 with IRIX does not improve performance for individual parallel applications. As in our work,

they rely on the automatic page migration mechanism. They compare the execution with the migration mechanism activated against a manually tuned initial memory placement. In this paper, we cannot compare with a manually tuned memory placement, due to the random (and dynamic) nature of our experiments with workloads of parallel applications, where the dynamic execution conditions can make invalid any initial placement for an application. Nevertheless, one of the goals of this paper is to show that this manual tuning is really not needed, so that we do not assume any *a priori* knowledge of the jobs submitted.

Nikolopoulos in [13] also examines the influence of the IRIX migration engine on workloads of parallel applications. They use simple workloads consisting of several instances of the same application and one I/O intensive process in the background to introduce a high number of context switches. Each application is started with 32 processors and the total load is set from 2 to 4 times the number of processors of the machine. The execution environment is set to DYNAMIC to allow the applications to reduce the number of processors used and adapt the execution of each job to the system load. The high number of context switches causes also a high number of process migrations among the physical processors. A process migration occurs when the process is preempted in a processor and later resumed in another processor. With these conditions, the memory migration engine is practically unable to maintain the data used by a process near the physical processor where the process has been moved. As a result, the performance degrades a lot. Although this execution environment is so hard for the applications to obtain performance, the authors show that, in this case, their proposal for dynamic page migrations implemented from user-level provides a moderate performance improvement of up to 20% compared to plain first-touch page placement. However, their work uses the same processor scheduler in all the experiments (the IRIX 6.5 native scheduler), and compares their own page migration engine with the IRIX migration engine.

In this paper, we demonstrate that the IRIX memory migration engine performs well when it is coordinated with the processor scheduler. As we cannot modify the IRIX kernel, the coordination is established at a higher level, ensuring that the processor scheduler is not going to introduce many process migrations across processors, and that a decision taken at the processor scheduler level is maintained during a large amount of time. In this way, the IRIX migration engine can follow the decisions taken at the processor level, moving the application data accordingly to make it reside in the node nearby.

We differentiate from the previous work that evaluates the dynamic memory migration mechanism in the following points: The evaluation performed in this work has been carried out by executions in a commercial architecture, not based on simulations. The SGI Origin 2000 is a CC-NUMA machine with a large enough number of processors (64) to be representative of problems that can appear in these kinds of systems. We also emphasize in the evaluation of the effect of the scheduler on the memory page migration behavior, to confirm that the memory page migration never introduces a significant negative effect in the execution of parallel applications and that, on the other hand, it can introduce significant benefits.

### 3. Memory Page Migrations in the SGI Origin 2000

In CC-NUMA architectures, processes should be scheduled in processors near their memory pages to achieve a good system performance. There are two ways to enforce that: doing a good initial memory placement, and using memory page migrations.

The first choice is only valid when the application memory accesses follow a static pattern. This technique needs an individual analysis of each parallel application, and the processor allocation should also be as much static as possible during the complete execution of the application. This approach is the one usually followed to execute message-passing applications and requires to spend a large amount of time to analyse the application and plan the data distribution. In this work, we focus our evaluation on the gains that can be obtained from OpenMP applications, and we leave message passing applications to be considered in a future work.

We also believe that the migration mechanism is orthogonal to the programming model: It also can be useful even if a initial data distribution has been performed, since in any case processes of the application can be migrated.

The second choice, memory migrations, will allow us to work without any knowledge about the application memory behavior. The native operating system used in this work, IRIX 6.5, has a dynamic page migration mechanism [17][16]. Dynamic page migration is a mechanism that provides adaptive memory locality to applications that execute in a NUMA machine. The migration mechanism checks the memory pages and decides whether a page must be migrated, depending on a migration policy. The memory migration mechanism implements a competitive algorithm based on comparing the remote memory access counters to the local memory access counters. When the difference between remote and local accesses is greater than a tunable threshold, an interrupt is generated to inform the operating system that the physical memory page is suffering an excessive number of remote references. The threshold defines the minimum difference between the local and any remote counter needed to generate a migration request interrupt. The interrupt handler decides whether the page has to be migrated or not. The final decision depends on several filters and thresholds that can limit the page migration. Filters are oriented to avoid an excessive number of page migrations and excessive pressure in a memory node.

Any portion of a virtual address space, down to the level of a page, may be connected to a specific policy via a *Policy Module*. When the operating system needs to execute an operation to manage a section of a process address space, it uses the methods specified by the Policy Module connected (attached) to that section. The policy module specifies, among others, the migration policy and the placement policy used for each memory region.

In this work, we have modified the placement policy and the migration policy associated with all the code, data, and stacks of parallel applications. We have set the placement policy to *PlacementFirstTouch* and the migration policy to *MigrationControl*. *PlacementFirstTouch* indicates that each physical memory page will be allocated in the node where it happens the first reference to the associated virtual address. *MigrationControl* indicates that users can specify different migration parameters. In this work, the *migration threshold* has

been set to 50%, which is the default value proposed by SGI in IRIX 6.5. The *migration threshold* defines the minimum difference between the local and any remote counter needed to generate a migration request interrupt.

Modifications to activate the memory page migration mechanism have been introduced in the NthLib [10], so that no application source code modifications are needed.

### 4. NANOS Execution Environment

In this paper we have evaluated three scheduling policies: the native IRIX scheduling, Equipartition, and Performance-Driven Processor Allocation. The last two policies have been implemented in the NANOS execution environment (NANOS-EE).

The NANOS-EE has three main components: the queuing system, the CPUManager, and the OpenMP parallel jobs [5]. This execution environment has been implemented to run on SGI Origin2000 systems on the context of the NANOS project [12].

Parallel jobs are submitted to the queuing system to be executed. The queuing system used in the NANOS-EE implements the job scheduling policy and coordinates with the CPUManager. The CPUManager informs the queuing system about when it is possible to start a new application and the job scheduling policy decides which application to start. When jobs start their execution, they request for processors to the CPUManager. The CPUManager applies the processor scheduling policy and distributes processors among jobs. Applications adapt their parallelism to the number of processors available and inform the CPUManager about the achieved performance. Periodically, or when applications inform about their speedup, the scheduling policy is re-applied. The run-time parallel library (NthLib) and the Performance Analysis library (SelfAnalyzer) are in charge of communicating the job and the CPUManager. More information about the NANOS-EE can be found in [4] and [5].

### 5. Scheduling policies

The CPU Manager currently supports several processor scheduling policies to distribute processors among applications. In this paper, we have evaluated two policies implemented by the CPUManager: Equipartition [11], and Performance-Driven Processor Allocation (PDPA) [4][5], and the native IRIX scheduling policy.

Equipartition is a dynamic space-sharing policy. It decides an equal allocation among running jobs. We have selected this policy because we want to analyse the effect of the memory migration mechanism when the scheduling does not consider the performance of running applications. In Equipartition, the processor allocation is a function of the number of running applications and the application request. Re-allocations are done at each job arrival and completion. Although Equipartition does not explicitly propose that, we use a fixed multiprogramming level to control the system load.

PDPA is a dynamic space-sharing policy. PDPA implements a coordinated scheduler: a processor allocation policy and a multiprogramming level policy. The processor allocation policy tries to allocate the maximum number of processors to the running applications that reach a given *target efficiency*. The target efficiency is a parameter of the policy and it has been set to 0.7 in the experiments done for this paper. PDPA decides the processor allocation based on the application request and the application

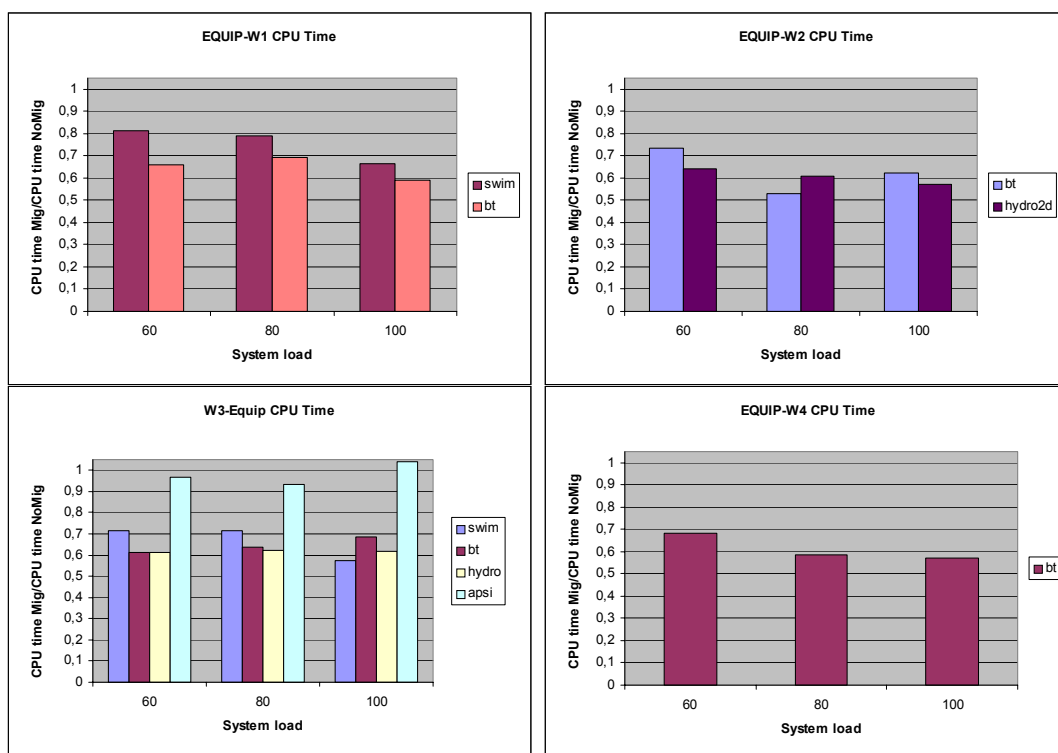


Figure 1 Influence of the migration mechanism in the CPU Time under EQUIP

performance, measured at run-time. Re-allocations are done at job arrival, job completion, and when jobs inform about their performance. The multiprogramming level policy determines when it is convenient to allow the execution of a new application.

Finally, the native IRIX scheduler is evaluated to have a reference value to which compare our results. The IRIX scheduler applies a time-sharing approach to kernel threads of running jobs. It also uses information related to the process placement to maintain as much as possible the process affinity.

## 6. Evaluation

To evaluate the influence of the memory page migration mechanism in the system performance we have executed four workloads of computational intensive parallel applications. The following experiments evaluate the impact of activating the memory migration mechanism in three relevant aspects of the execution of a job: the CPU time consumed by each application, the speedup obtained and the processor allocation received by each application. Moreover, we compare the effects of the migration mechanism when the processor scheduler is performance-driven or not.

Applications that compound the different workloads are swim, hydro2d, and apsi from the specfp95 [18] and bt from the NAS Parallel Benchmark Suite [1][7]. Each one of these applications has different characteristics with respect to their scalability. Swim reaches a super-linear speedup, hydro2d reaches a medium-low speedup, apsi is not scalable at all, and bt reaches a very good speedup, quite linear.

Table 1 shows the composition of the four workloads used to evaluate the memory page migration mechanism. Each cell represents the percentage of the workload filled with each job. In

all the experiments, swim's, bt's, and hydro2d's request for 32 processors and apsi's request for 2 processors (due to its bad scalability). We have generated three different system loads, 60%, 80%, and 100%. Each experiment represents a system where applications arrive following an exponential inter-arrival function with a different frequency. To make repetitive our experiments, we use workload trace files following the specification proposed by Feitelson in [19]. We also show the total number of instances of applications executed when the load is set to 100%.

Table 1. Parallel workloads

	Swim	Bt	Hydro2d	Apsi	Total number of applications executed (load=100%)
W1	50%	50%	-	-	61
W2	-	50%	50%	-	48
W3	25%	25%	25%	25%	125
W4	-	100%	-	-	99

Experiments have been carried out in a real system, a SGI Origin 2000 with 64 250 Mhz. R10000 processors.

### 6.1 Influence in the application execution time

The first aspect that we want to evaluate is the influence of activating the automatic memory page migration mechanism in the CPU time consumed by each application. We want to compare the results under the scheduling policies presented in previous section: Equipartition, PDPA, and IRIX. In these experiments, the

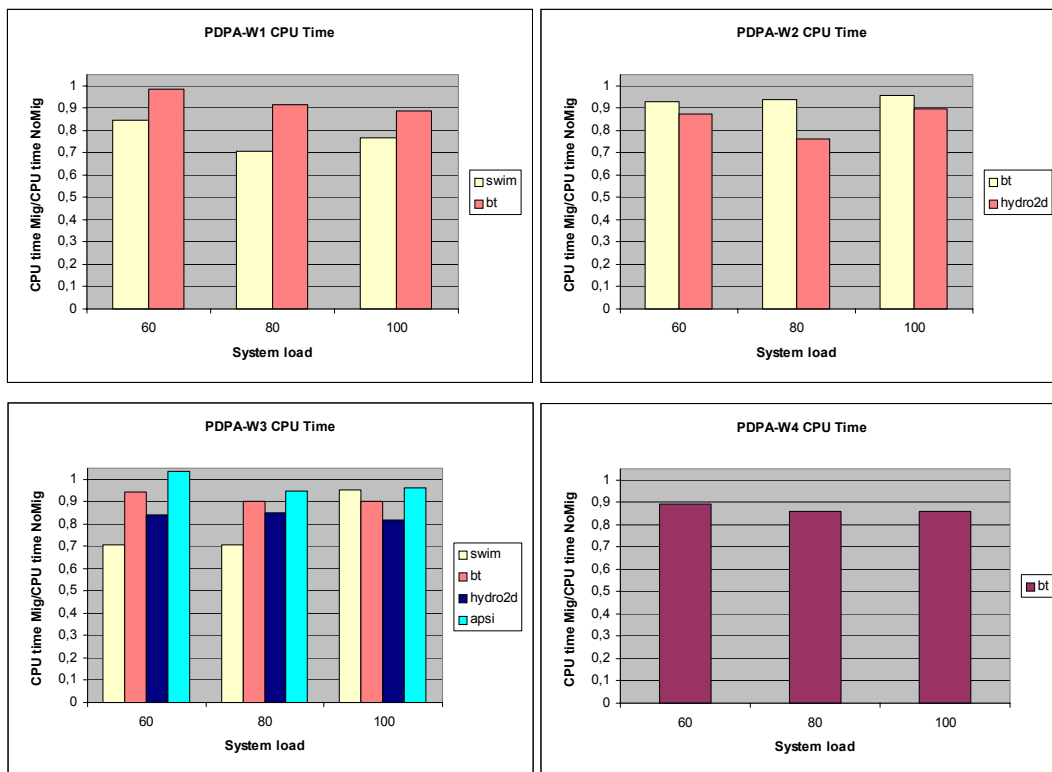


Figure 2 Influence of the migration mechanism in the CPU Time under PDPA

default multiprogramming level has been set to 4 applications in all the scheduling policies. However, PDPA adjusts the multiprogramming level dynamically. The multiprogramming level is the number of applications allowed to run concurrently.

### 6.1.1 Equipartition

Figure 1 shows the results obtained by the Equipartition policy (Equip). The plots present the ratio between the CPU time consumed when executing with memory page migrations with respect to when executing without memory page migrations. The x-axis shows the system load evaluated and the y-axis shows the benefit when using memory migrations (CPU time consumed with migrations)/(CPU time consumed without migrations), averaged per application. Values lower than 1.0 mean that with memory migrations the application consumes less CPU time.

As we can see in the figure, the activation of the memory page migration mechanism has a significant and positive influence in the CPU time spent by each application. Although the influence is different depending on the application, the memory page migration improves a 50% (in average) the CPU time consumed per application.

If we analyse the results per application, the improvement introduced in swims, bt's, and hydro2d's are 38%, the 68%, and the 69% respectively.

After this evaluation, we could conclude that the automatic memory migration mechanism provided by IRIX 6.5 introduces significant benefits to applications executed under Equipartition in the NANOS-EE. This execution environment has the characteristic that applications receive a very stable processor

allocation, with the minimum number of process migrations, but also that the number of processors assigned does not consider the application performance. We have executed the same experiments with PDPA, to evaluate the effects in the CPU time when the scheduling policy adjusts the allocation based on the application performance.

### 6.1.2 Performance-Driven Processor Allocation

Figure 2 shows the results obtained by the PDPA scheduling policy. It presents the ratio between the CPU time consumed when executing with memory page migrations with respect to when executing without memory page migrations. The x-axis shows the system load and the y-axis shows the ratio of (CPU time consumed with migrations)/(cpu time consumed without migrations), averaged per application. In that case, the default multiprogramming level has been set to four jobs, but PDPA implements a dynamic multiprogramming level. For instance, the maximum multiprogramming level has been up to 36 jobs executed in parallel in the case of w3 (load = 100%).

In this case, we can see that the influence of the memory page migration mechanism when executing under PDPA is also positive, although it is not so significant as under Equipartition. In all applications the CPU time consumed with memory page migrations is less than the CPU time consumed without memory page migrations, in average, a 13% of improvement.

Analysing the effects per application, the memory page migration mechanism improves the CPU time consumed by swim's, bt's, hydro's, and apsi's in a 29%, 9%, 19%, and 1% respectively.

The cases of swim's and apsi's are extreme. The first ones, swim's, reach super-linear speedups, and their performance is very affected by the memory performance. On the other hand, apsi's have very bad speedup, and PDPA allocates only 2

burst CPU time, and the number of bursts per cpu comparing IRIX, PDPA and Equipartition in the case of the workload 1<sup>1</sup>.

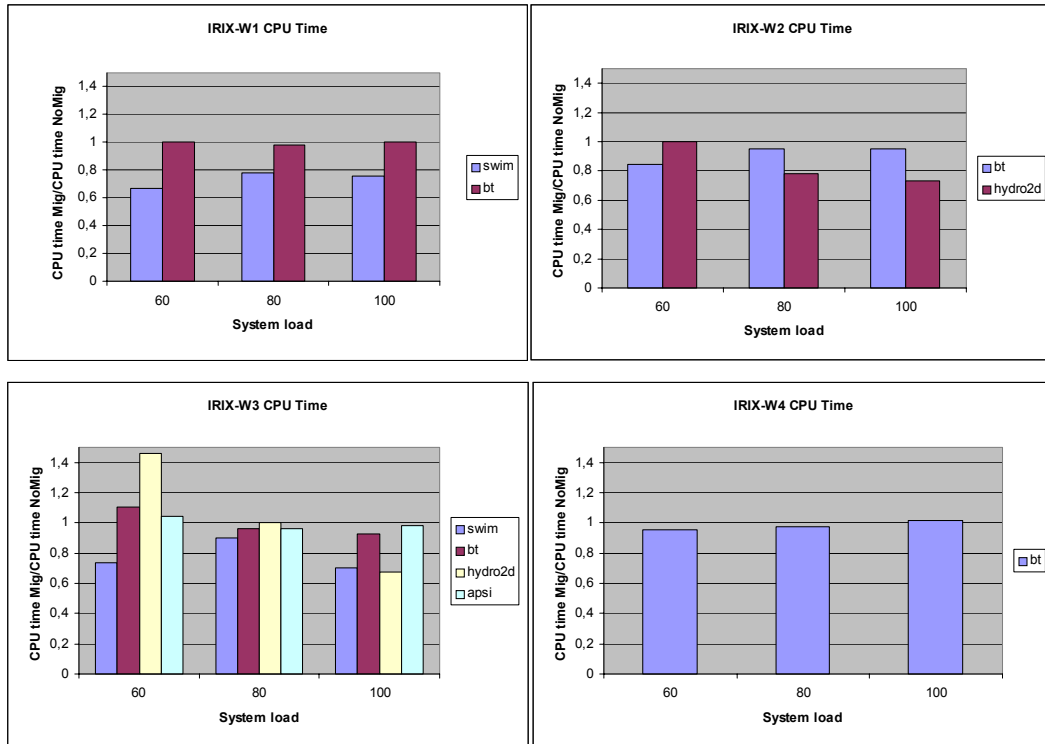


Figure 3 Influence of the migration mechanism in the CPU Time under IRIX

processors in average to them (they only request for two processors). Taking into account this processor allocation and that the CPUManager maintains the processor mapping as much as possible, the impact of the memory migration in the apsi's performance is negligible.

### 6.1.3 IRIX processor scheduling policy

Figure 3 shows the results obtained by the native IRIX scheduling policy. It presents the ratio between the CPU time consumed when executing with memory page migrations with respect to when executing without memory page migrations. The x-axis shows the system load and the y-axis shows the ratio of (CPU time spent with migrations)/(CPU time spent without migrations), averaged per application.

We can observe that, even that the migration policy improves the CPU time of applications under the IRIX policy (by 10% in average), it does not introduce as much benefits as in the case of Equipartition. Theoretically, activating the DYNAMIC flag in OpenMP, the resulting processor allocation should be equal to the Equipartition. However, if we analyse in detail the behaviour of jobs under the IRIX policy, we observe that jobs suffer a lot of unneeded kernel thread migrations. To give an insight into that, we have measured the number of process migrations, the average

	IRIX	Equip.	PDPA
Process migrations	>150000	325	66
Average burst time per cpu	0.243 sec.	>11 sec.	>10 sec.
Average number of bursts per cpu	>2000	43	41

Table 2 Process migrations per policy, workload 1 (load=100%)

<sup>1</sup> The rest of the workloads have a similar behavior

Table 2 shows the results obtained from the three policies. As we can see, the native IRIX policy generates much more process migrations than the other two policies. Moreover, IRIX generates bursts of cpu 50 times less than Equipartition or PDPA (243 miliseconds vs. more than 10 seconds). Finally, and as a logical effect of the high number of process migrations in IRIX, the number of bursts generated by IRIX is also significantly greater

average. In particular, swim has a remarkable 55% of improvement in its speedup, bt improves its speedup by 14%, hydro2d by 39%, and apsi by 2%.

The speedup obtained by one application usually depends on the number of processors assigned to it. In the next section, we evaluate the influence of the memory migration mechanism in the processor allocation decided by PDPA.

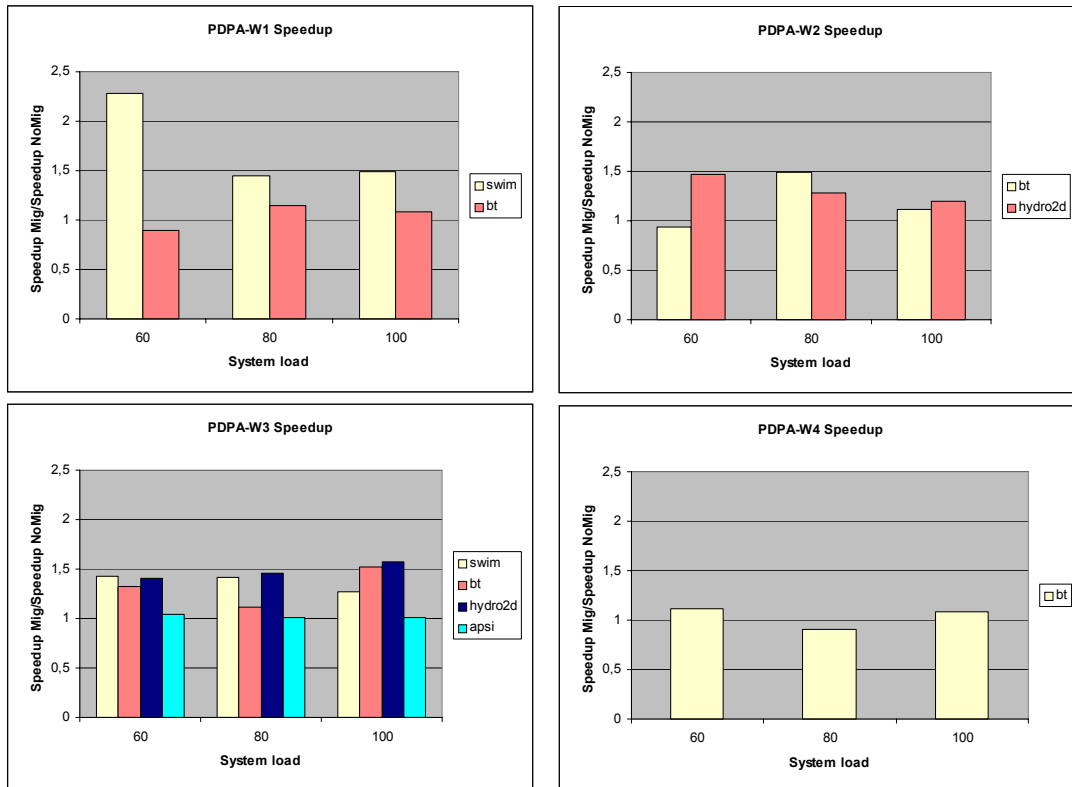


Figure 4 Influence of the migration mechanism in the application speedup

than with Equipartition or PDPA. This behaviour is not inherent to the processor allocation policy, but it depends on the process-processor mapping that it decides. As a result of it, the memory migration mechanism cannot effectively do its work.

## 6.2 Influence in the application speedup

The second aspect that we wanted to evaluate is whether the speedup attained by each application depends on the use of the memory migration mechanism. To do that, we have selected to use PDPA as the scheduling policy.

Figure 4 shows the ratio between the application speedup when executing with memory page migrations and when executing without memory page migrations when the scheduling policy is PDPA. The x-axis shows the system load and the y-axis shows the ratio of (speedup measured with migrations)/(speedup measured without migrations), averaged per application.

Applications executed under PDPA and with the memory migration mechanism activated improve their speedup by 27%, in

## 6.3 Influence in the processor allocation

The third aspect that we wanted to evaluate is whether the amount of processors received by each application depends on the use of the memory migration mechanism.

Figure 5 shows the ratio between the processor allocation decided when executing with memory page migrations and when executing without memory page migrations. The x-axis shows the system load and the y-axis shows the ratio of (processor allocation decided with migrations)/(processor allocation decided without migrations), averaged per application.

Applications executed with the memory migration mechanism activated receive 14% more processors than without the migration mechanism. In particular, swim has an increment of 10% in the number of processors assigned, bt allocation is incremented by 11%, hydro2d allocation is incremented by 31%, and apsi allocation is incremented by 2%.

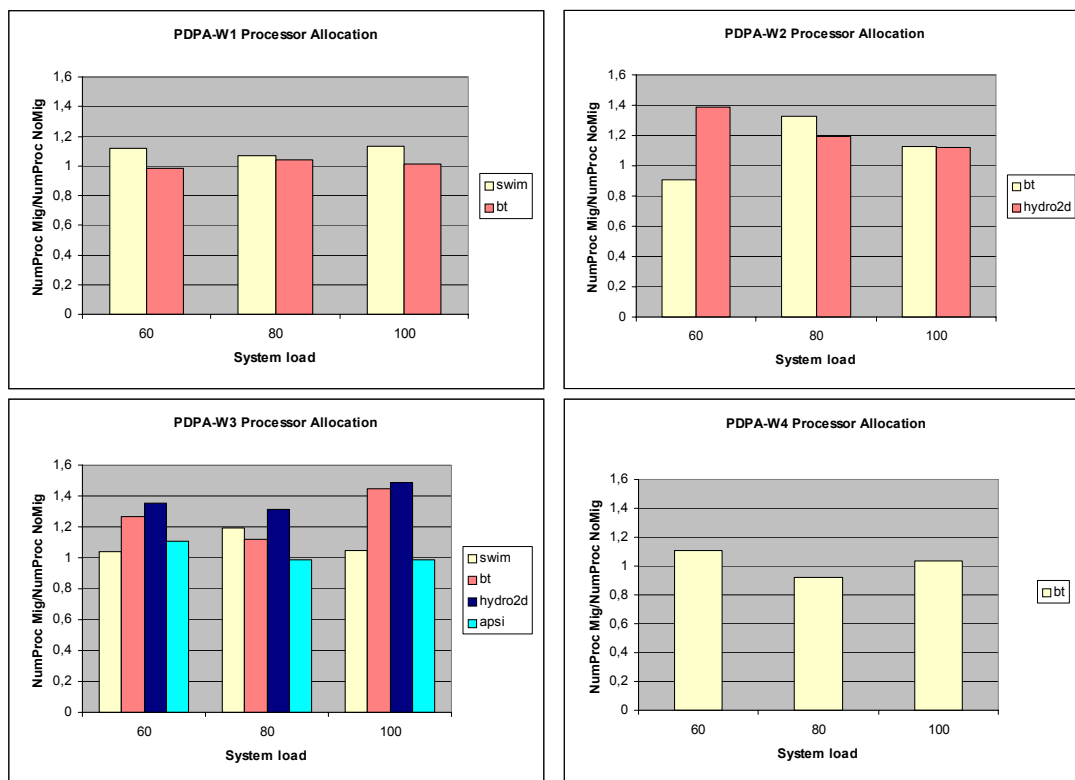


Figure 5 Influence of the migration mechanism in the processor allocation under PDPA

Correlating these values with the calculated in the previous section, we can see that in the case of swim the relationship is not proportional. While the processor allocation is increased by 10% with PDPA, the speedup is improved by 55%. This is a consequence of swim being specially memory sensitive in addition to the better ratio between local and remote memory accesses when swim is executed with the migration mechanism.

We can correlate these results with the CPU time consumed by each application. This way, we remark that applications executing under PDPA, with the migration mechanism activated, increment by 14% their processor allocation, although they use a 13% less of CPU time. This reduction in the CPU time is also a consequence of the reduction in the number of remote memory accesses.

#### 6.4 Comparing scheduling policies

If we compare how the migration mechanism benefits the three scheduling policies evaluated we can extract some conclusions. Comparing Equipartition with IRIX, we can observe the effect in the performance of other parts of the scheduler, different from the processor allocation, such as the processor mapping. Even that both policies allocate the same number of processors to applications (in average), the fact of maintain a kernel thread in the same cpu significantly improves the job and system performance, and facilitates the work of the memory migration mechanism. Analysing the absolute values, we have observed that Equipartition outperforms the native IRIX in a 40% when memory migration is enabled, where the two policies reach the best results.

Comparing Equipartition with PDPA, we have observed that PDPA is a policy more robust to changes in the system configuration because it dynamically evaluates the application performance and adapts the processor allocation and the multiprogramming level to the system and job characteristics. In this case, PDPA detects that applications executed in an execution environment with memory page migrations are more efficient and

and can take advantage of receiving more processors. This way, it increases the processor allocation to improve the application (and system) performance. Analysing the absolute values, we have observed that Equipartition outperforms PDPA in a 5% in average in those workloads that have been previously tuned (both in processor request and in multiprogramming level) and that use memory migrations. Nevertheless, PDPA outperforms Equipartition in a 30% if the memory migration mechanism is not used.

#### 7. Conclusions

In this paper, we have evaluated the performance of the dynamic memory page migration mechanism provided by IRIX. We have evaluated its performance under different system loads and three scheduling policies: the native IRIX policy, Equipartition, and PDPA. Equipartition and PDPA are implemented in the NANOS execution environment. Results have been taken in a SGI Origin 2000 with 64 processors for availability reasons. However, we consider that they are quite representative of most of the current CC-NUMA systems, and that the results obtained could be extrapolated to larger systems because the migration mechanism decisions are distributed, and from that, it should be easily scalable.

The most important conclusion is that the automatic memory migration mechanism is sensitive to the processor allocation stability. It needs that jobs maintain stable the number and the set of processors in use to be able to detect the application pattern and migrate memory pages. This consideration is not taken into account by the native IRIX scheduler. It is, in fact, a problem of poor cooperation between different parts of the system. Another important point is that in the NANOS execution environment the number of processes running is always less or equal than the number of processors, minimizing the number of process migrations.

Results show that the memory page migration mechanism provided by IRIX always improves the application performance,

that is, applications always consume less CPU time with the memory migration mechanism activated than without it. Results also show that the benefit depends on the application and on the scheduling policy (10% in the case of IRIX, 50% in the case of Equipartition and 13% in the case of PDPA). Those policies that adjust the application processor allocation as a function of their performance are less penalized by the fact of disabling the memory page migration mechanism.

The combination of PDPA with the memory migration mechanism goes on the direction of systems that are self-evaluated and self-configured. This kind of systems is robust to user activities, such as incorrect job requests, and needs less human interventions to administer them. With this combination, users do not have to spend their time tuning their applications neither in the number of processors requested nor in the memory placement.

To conclude, we could say that the best system configuration for OpenMP will be to activate the memory migration mechanism, combined with a scheduling policy that considers the job performance, and a memory conscious processor placement policy to maintain the processor affinity.

## 8. Acknowledgments

The Spanish Ministry of Education has supported this work under grant CYCIT TIC2001-0995-C02-01, and the ESPRIT Project POP (IST-2001-33071). The research described in this work has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA).

## 9. REFERENCES

- [1] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo and M. Yarrow, "The NAS Parallel Benchmarks 2.0", Technical Report NAS-95-020, NASA, December 1995.
- [2] T. Brecht, "On the Importance of Parallel Application Placement in NUMA Multiprocessors", Proceedings of the Symposium Experiences with Distributed and Multiprocessor Systems (SEDMS IV), San Diego, CA, September 1993.
- [3] R. Chandra, S. Devine, B. Verghese, A. Gupta, M. Rosenblum, "Scheduling and Page Migration for Multiprocessor Compute Servers", Proceedings of Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 1994.
- [4] J. Corbalán, X. Martorell, J. Labarta, "Performance-Driven Processor Allocation", Proc. of the 4th Operating System Design and Implementation (OSDI 2000), pp. 59-71, San Diego, California, USA, October 2000.
- [5] Julita Corbalán, "Coordinated Scheduling and Dynamic Performance Analysis in Multiprocessor Systems", Ph.D. Thesis, Universitat Politècnica de Catalunya, 2002, <http://people.ac.upc.es/juli/thesis.pdf>.
- [6] D. Jiang and J.P. Singh, "Scaling Application Performance on a Cache-Coherent Multiprocessor", Proceedings of the 26th International Symposium on Computer Architecture, pp. 305-316, Atlanta, USA, 1999.
- [7] H. Jin, M. Frumkin, J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report: NAS-99-011, 1999.
- [8] J. Laudon and D. Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server, Proc. of the 24th Int. Symp. on Computer Architecture, pp. 241-251, 1997.
- [9] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. L. Scott, "Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems", Proceedings of the 9th International Parallel Processing Symposium, pp. 480-485, Santa Barbara, CA, April 1995.
- [10] Xavier Martorell, "Dynamic Scheduling of Parallel Applications on Shared-Memory Multiprocessors", Ph.D. thesis, Universitat Politècnica de Catalunya, 1999, <http://www.ac.upc.es/homes/xavim/dynsched.pdf>.
- [11] C. McCann, R. Vaswani, J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors", ACM Transactions on Computer Systems, 11(2), pp. 146-178, May 1993
- [12] NANOS ESPRIT Project (E-21907), <http://www.ac.upc.es/nanos>
- [13] D.S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta and Eduard Ayguadé, "User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors", Proceedings of the 30th Annual International Conference on Parallel Processing (ICPP '00), pp. 95-103, Vancouver (Canada), August 2000.
- [14] OpenMP Organization. OpenMP Fortran Application Interface, v. 2.0, <http://www.openmp.org>, June 2000.
- [15] Silicon Graphics, Inc. IRIX6.5 Online Manual Pages, `cpuset`, `miser_cpus(4)`, `miser(1,4,5)` - define and manage set of CPUs and Miser resource manager, 2000.
- [16] Silicon Graphics, Inc. IRIX6.5 Online Manual Pages, `mld(3)`, `mldset(3)` and `mmci(5)` - memory locality domain operations and memory management control interface, 2000.
- [17] Silicon Graphics, Inc. Origin2000 and Onyx2 Performance Tuning and Optimization Guide, Document number 007-3430-002, <http://techpubs.sgi.com>, 2000.
- [18] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. Available at <http://www.spec.org/osg/cpu95>, 1995.
- [19] "The Standard Workload Format", <http://www.cs.huji.ac.il/labs/parallel/workload/swf.html>