

# Page Migration with Dynamic Space-Sharing scheduling policies: 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

*In this paper, we claim that memory migration mechanism is a useful approach to improve the execution of parallel applications in dynamic execution environments, but that their performance depends on related system components such as the processor scheduling. To show that, we evaluate the automatic memory migration mechanism provided by IRIX in Origin systems, under different dynamic processor allocation policies when executing OpenMP parallel multiprogrammed workloads. We have focused the evaluation on the effects of the page migration mechanism on the CPU time consumed by each application, the processor allocation received, and the speedup.*

*Results demonstrate that, if the processor scheduler is memory conscious, that is, it maintains as much as possible the system stable, the automatic memory page migration mechanism provided by IRIX improves the CPU time consumed by OpenMP applications.*

**Keywords:** CC-NUMA. Memory page migration. Dynamic processor allocation policy. OpenMP. Multiprogrammed workload.

## 1. Introduction

Current large shared-memory multiprocessors have a CC-NUMA (Cache-Coherent Non-Uniform Memory Access) architecture <sup>(1)</sup>. 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, a good memory placement is critical to reach a good system performance. In this work we will focus on the performance of parallel jobs when executed in a multiprogrammed environment, in particular in OpenMP parallel jobs.

From the point of view of memory accesses, parallel jobs can be classified in two groups: regular and irregular. In the first case, programmers can perform an exhaustive analysis of the job and decide a good memory placement. In the second case, the application analysis is complicated and the job must include code to perform a dynamic data re-distribution. Since application analysis is a hard work, several proposals for automatic initial memory placement have been proposed such as Random, Round-robin or First touch <sup>(2)</sup>. The First touch proposal provides good results as an automatic technique if the application is regular. It consists of allocating memory in the node that first accesses the data.

Nevertheless, the performance of any technique that applications or the system include to provide a good initial memory placements depends on the processor allocation decided by the system. If jobs run in a system with dynamic scheduling policies, where the number of processors can be dynamically modified to increase the overall system performance, a good initial memory placement will not be enough to reach a good performance.

Taking this into account, 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.

We defend that the need of detailed application analysis can be avoided by using automatic memory page migration techniques. These techniques can also be applied in the case of dynamic processor reallocation, where the static analysis is not applicable. 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 the execution of individual application in static execution environments, or compared with static hand-coded memory placement, where the migration mechanism does not introduce significant benefits.

In this work, we want to demonstrate that the use of page migration mechanisms improves the overall system performance if the processor scheduling (resource manager and parallel library) has quick adaptability to the workload and stability in the processor allocation. Moreover, we also 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, since the migration mechanism can dynamically adjust the memory mapping to the job behavior.

To demonstrate that, we have evaluated the particular 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 <sup>(3)</sup> parallel jobs. Scheduling policies different than the native IRIX scheduler have been executed in the NANOS execution environment (NANOS-EE) <sup>(4)(5)</sup>. The main goal of the NANOS-EE is to provide a dynamic and efficient execution environment to parallel applications in shared-memory multiprogrammed multiprocessor systems.

Different from other previous works, the target scenario of this work is OpenMP parallel applications executing in a multiprocessor multiprogrammed system with a dynamic space-sharing policies.

Results show that the automatic memory migration mechanism provided by IRIX is very useful if (1) the scheduling policy maintains stable the processor allocation as much as possible and (2) that responsiveness to system decisions is as fast 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 behavior, 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 3.2 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 <sup>(6)</sup>, Bretch evaluates the effect of the memory placement in the performance of parallel applications when executing without memory page migrations. It concludes that a good initial memory placement is critical for the performance of individual parallel applications if no migrations are used. Evaluation was

done with 16 processors. He assumes that a good initial memory placement in parallel applications is also critical for the overall performance.

In <sup>(2)</sup> authors work with simple page placement policies to demonstrate that the initial placement of a parallel application is important to achieve good performance. They demonstrate that a simple first-touch policy is enough to improve the execution of parallel application. They perform the evaluation in a system with 64 processors and evaluate single applications. They also evaluate page migration. However, in their case applications do not follow any regular phase that allows the page migration mechanism to improve the application execution time.

Jiang and Singh show in <sup>(7)</sup> that turning on dynamic page migration on the Origin2000 with IRIX does not improve performance for individual parallel applications. 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 <sup>(8)</sup> 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 their proposal for dynamic page migrations implemented at user-level provides a moderate performance improvement of up to 20% compared to plain first-touch page placement. 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 <sup>(9)</sup> 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 a new version of IRIX (IRIX 6.5), where these problems are already solved and the migration mechanism is much more efficient.

The main difference between this work and previous works refers to the paper claim. Most of the previous works conclude that memory migration is not a valid approach whereas we claim that memory migration is valid. Difference in the conclusions is due to several issues: Most of the works based their evaluations on individual applications or simple workloads (with 2 or 3 jobs) whereas we evaluate more complete workloads of parallel applications with different scheduling policies and system load. Regarding the execution environment, we have considered execution environments that implement dynamic space-sharing policies to improve the system utilization and the application response time. In these systems,

static manual memory placement is not applicable since job allocation can be dynamically modified. Some of the previous works were based on simulations, whereas we evaluate the memory migration with the real execution of multiprogrammed workloads. With respect the architecture, the target of this work is a CC-NUMA multiprocessor execution environment with a large enough number of processors (64) to be representative of problems that can appear in these kinds of systems, whereas some of the previous works were done in small machines.

### **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 executing the job with a static scheduling policy, or using memory page migrations.

To use a manual memory placement is only valid when the application memory accesses follow a static pattern and applications are running with a static processor allocation. This technique needs an individual analysis of each parallel application, and the processor allocation should also be static during the complete execution of the application, as in the case of pure batch schedulers. This approach requires spending a large amount of time to analyze the application and plan the data distribution. Moreover, it is not only a matter of time, analyzing the memory pattern of a parallel application and deciding a memory placement is a hard work that must be done by an expert.

In this work, we focus on demonstrating the benefits that can be automatically obtained by OpenMP applications just using automatic mechanisms of memory migration. Additionally, we also believe that the migration mechanism is orthogonal to the initial memory placement: even with an initial data distribution, processes of the application can be migrated, improving their performance using memory migrations..

The native operating system used in this work, IRIX 6.5, includes a page migration mechanism <sup>(10)(11)</sup>.

Dynamic memory 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 page migration is disabled by default. OpenMP users can transparently activate it by setting the `_DSM_MIGRATION` environment variable. When this variable is on, the runtime library executes the code that modifies the characteristics of application memory pages.

IRIX 6.5 not only provides page migration, the system includes a complete memory management that allows users to specify the page placement policy, the page replication policy, etc.

In next sections we resume the memory management functionalities, focusing in the memory page migration mechanism.

### 3.1 IRIX memory management

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**. Users are allowed to select a policy from a set of available policies for each virtual memory operation. 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. Virtual memory operations are:

- Initial allocation: **Placement policy** (Determines what physical memory node to use when memory is allocated), page size policy (Determines what virtual page size to use to map physical memory), and fallback policy (Determines the relative importance between placement and page size).
- Dynamic relocation: **migration policy** (determines the aggressiveness of page migration).

Modifying the policy associated to these operations, users can modify from the page size of their jobs to the migration policy.

In this work, we have modified the placement policy and the migration policy associated with all the code, data, and stacks of parallel applications. Modifications to activate the memory page migration mechanism have been introduced in the OpenMP runtime library, and then no application source code modifications are needed.

To activate the memory migration mechanism, the OpenMP runtime library must connect the application address space to a Policy Module with the migration mechanism activated (by default is disabled). To do that, it must **(1)** fill a policy module data structure (a policy set) with the predefined values, **(2)** create an empty policy module associated to this data structure and modify it to activate the migration mechanism, and **(3)** connect this particular policy module with the application address space.

Table I shows the specific policy module interface used in the runtime library to modify the placement policy and the migration policy. The *pm\_filldefault()* function (step 1) fills a *policy set* structure with the predefined values in the system. A *policy set* structure has fields to specify the following policies: placement policy name and arguments, fallback policy name and arguments, replication policy name and arguments, **migration policy name and arguments**, paging policy name and arguments, and page size.

(Place Table I here)

Once we have a default Policy Module we can modify it. In this work, we have modified the placement policy fields and the migration policy fields. We have set the placement policy to ***PlacementFirstTouch*** and the migration policy to ***MigrationControl***. *PlacementFirstTouch*<sup>1</sup> indicates that memory will be

---

<sup>1</sup> Data is placed in the node that accesses the first time. In 2 they demonstrate that First touch is better than round robin.

allocated in the node where creation happened. *MigrationControl* indicates that users can specify different migration parameters. The *MigrationControl* policy receives as argument a data structure that defines these user parameters.. Once modified the Policy Module, we have to create the handler that we will use to associate to the memory regions (step 2). The Policy Module creation is done using the *pm\_create()* system call, that receives as a parameter the Policy Module previously filled up.

Once created, it only remains to attach the Policy Module with the memory regions (step 3). In this case, we have created only one Policy Module because we want to apply the same policy to all the application memory regions. The association of a Policy Module with a memory region is done through the *pm\_attach()* function. This function receives as parameter the Policy Module, the address of the memory region, and the size of the memory region. The *pm\_setdefault()* function associates a policy module to the stack, text, or heap memory regions. This function has been used to modify the default policy associated to these memory regions.

## **3.2 Migration modules**

The memory migration subsystem is composed by several modules: Detection module, Migration Control module, Migration Engine module, Migration Control Periodic Operations module, and Memory Management Control Interface module (mmci). .

The *Detection module* monitors memory accesses issued by nodes to physical memory pages. This module informs, via an interrupt, the *Migration Control module* when a page is suffering an excessive number of remote accesses. The Migration Control Module decides whether a page should be migrated or not. The *Migration Engine module* carries out the effective page migration. The *Migration Control Periodic Operations module* executes periodic operations needed by the Migration Control module.

Finally, the *Memory Management Control Interface module* provides an interface for users to tune the migration policy associated with an address space.

### 3.3 Migration Control module

The Migration Control module 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 interrupt handler decides whether the page has to be migrated or not. The final decision depends on several filters that can limit the page migration.

Filters that apply the Migration Control module are based on **(1)** the distance between the source and the destination; **(2)** the memory pressure on the destination, this filter avoids an excessive number of migrations per page, and **(3)** filters quick temporary remote memory accesses.

Table II shows the memory migration parameters tunable in the SGI Origin 2000 and the values set in the runtime library (we have set the parameters to the default values). The *migration threshold* parameter defines the minimum difference between the local and any remote counter needed to generate a migration request interrupt.

(Place Table II here)

The *freeze control* enables or disables the freeze of bounced pages. The *melt control* is consulted by the Migration Control Periodic Operations module to unfreeze pages making it migratable again. The *dampening control* is used to filter quick temporary remote memory accesses. The *refcnt control* enables or disables the use of the extended reference counters.

## 4. NANOS Execution Environment

The NANOS-EE has the following main components <sup>(12)</sup>: the queuing system, the CPUManager, the profiling library (SelfAnalyzer), and the parallel library (NthLib). This execution environment has been implemented to run on SGI Origin2000 systems on the context of the NANOS project <sup>(4)</sup>.

### 4.1 Queuing System

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. The job scheduling policy implemented is FIFO. Since OpenMP jobs are malleable, fragmentation does not exist because jobs adapt their allocation to the resources available.

### 4.2 CPUManager

Once started, parallel jobs are managed by the CPUManager, which decides how many and which processors to allocate to each parallel job. When jobs start their execution, they request for processors to the CPUManager. The CPUManager applies the processor scheduling policy and distributes processors among jobs. It always tracks the number of processors used by the parallel applications to ensure that there are no more threads running than physical processors. The CPUManager includes the implementation of several scheduling policies. In this paper we have used the Equipartition <sup>(13)</sup> and the Performance-Driven Processor Allocation <sup>(14)</sup> policies. Scheduling policies evaluated in this work are fully explained in section 5.

Once informed about their current processor allocation, applications adapt their parallelism to the number of processors available through the parallel library (see section 4.3), measure their current speedup (using SelfAnalyzer, see section 4.4), and inform the CPUManager about their achieved performance. The

profiling is only performed if the CPUManager is applying a Performance-Driven policy, not in the case of Equipartition.

When a new application arrives or a running application finishes, and when all the applications inform about their speedup in the case of Performance-Driven policies, the scheduling policy is re-applied. The parallel library (NthLib) and the performance analysis library (SelfAnalyzer) are in charge of coordinating the job with the CPUManager. More information about the NANOS-EE can be found in <sup>(14)</sup> and <sup>(12)</sup>.

### **4.3 OpenMP runtime parallel library: NthLib**

The NthLib is the OpenMP runtime library used in the NANOS execution environment. It is automatically loaded with parallel applications. The NthLib has been designed and implemented to provide a powerful run-time support to execute in a multiprogrammed multiprocessor environment. It is in charge of generating and controlling the application parallelism specified by the programmer through OpenMP directives inserted in the source code. These directives are translated by the compiler into calls to the NthLib.

The NthLib library requests for processors to the CPUManager and reacts to the CPUManager decisions. That means a quick adaptation to the available processor allocation. This quick adaptability is one of the key issues to reach a good behaviour in multiprogrammed execution environments because it avoids the processor time-sharing by running threads. In section 6.1.3, we will show that this is one of the main differences with the native IRIX parallel library, which has a slower adaptability to workload changes.

### **4.4 Profiling library: SelfAnalyzer**

The SelfAnalyzer is also a runtime library automatically loaded with parallel applications. It dynamically measures the speedup of OpenMP applications and informs the CPUManager about it. It is based on the

the fact that a lot of scientific applications are iterative, and then we can use measured values for past iterations to predict (at runtime) the performance of future iterations.

SelfAnalyzer works in the following way: **(1)** It dynamically detects the iterative structure of applications using a Dynamic Periodicity Detector library <sup>(15)</sup>. This library receives as input a dynamic sequence of values and generates as output a boolean value indicating if the value corresponds to the start of a period or not. We are using as sequence of values the parallel loop identifier automatically generated by the compiler. **(2)** Once detected the iterative structure of the application, SelfAnalyzer executes some iterations of the most external sequential loop with a *baseline* number of processor to get a *baseline execution time* measure. **(3)** After computing the reference measure, the application continues executing with the number of processors allocated by the scheduling policy. **(4)** SelfAnalyzer calculates the speedup of the application as the ratio between the execution time of one loop iteration (averaged from several iterations) with P processors and the *baseline execution time*. In <sup>(12)</sup> we demonstrate that four processors is a good value to use as processor baseline.

## 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 <sup>(13)</sup>, and Performance-Driven Processor Allocation (PDPA) <sup>(14)(12)</sup>. We have also evaluated the native IRIX scheduling policy.

### 5.1 NANOS Execution Environment: CPUManager policies

#### 5.1.1 Equipartition

Equipartition <sup>(13)</sup> is a **dynamic space-sharing** policy. Equipartition is applied each time a new application starts or a running application finishes. It allocates the minimum between the number of processors requested by the application and  $P/\text{Number\_of\_jobs}$ .

We have selected this policy because we want to analyze the effect of the memory migration mechanism when the scheduling does not consider the performance of running applications. Although Equipartition does not explicitly propose it, we use a **fixed** multiprogramming level to control the system load. We have used a multiprogramming level of four jobs in all the scheduling policies evaluated, that means a maximum of four jobs running simultaneously.

### 5.1.2 Performance-Driven Processor Allocation:PDPA

PDPA<sup>(14)</sup> is also 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. This value is based on the experience resulting from previous works <sup>(14)(12)</sup>. PDPA decides the processor allocation based on the application request and the application speedup measured by SelfAnalyzer. 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. The multiprogramming (ML) level policy uses a minimum multiprogramming level and adjusts it depending on the workload characteristics. If PDPA detects that there are free processors for a period of time, it increments the ML. In that case, when any of the running jobs finishes their execution, the ML is reduced in one job and re-evaluated later to always tend to the minimum ML.

## 5.2 IRIX Execution Environment

Finally, the native IRIX Execution Environment is evaluated to have a reference value to which compare our results. The IRIX EE is composed by the queuing system, the IRIX scheduler, and the mp IRIX

parallel library. We have used the same queuing system as in the NANOS EE to control the multiprogramming level and execute jobs under the same conditions as with the NANOS EE.

The IRIX scheduler applies a time-sharing approach to kernel threads of running jobs. The IRIX scheduler tries to maintain as much as possible the process affinity; however, as we will see in section 6.1.3, the mechanism applied is sometimes harmful for parallel application since it introduces a lot of process migrations.

The mp IRIX parallel library is in charge of generating and controlling the parallelism specified by programmers through OpenMP directives. The mp library also introduces mechanisms to adapt the job parallelism to the system load. These mechanisms are controlled by the definition of environment variables such as `OMP_DYNAMIC`, that enables or disables the dynamic adjustment of the number of threads available for execution of parallel regions, or `MP_BLOCKTIME` that defines the time slave threads iterates looking for work before blocking. In this work, we have set `OMP_DYNAMIC` to `TRUE` and the `MP_BLOCKTIME` to 200000 (this value has been tuned empirically in <sup>(5)</sup>).

Nevertheless, the mp library is not coordinated with the IRIX scheduler. The mp library bases its decisions on the observation of the system, not in information directly provided by the system. This difference generates that the mp library reacts more slowly than the NthLib to changes in the load of the system, resulting in performance degradation when the number of processes is equal or greater than the number of processors.

## **6. Evaluation**

To evaluate the influence of the memory page migration mechanism in the system performance we have executed four workloads of computationally 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 <sup>(16)</sup> and bt from the NAS Parallel Benchmark Suite <sup>(17)</sup>. The specfp95 applications have been parallelized, by hand, using OpenMP. Table 3 shows the sequential execution time and the speedup characteristics of these applications. Each one has different characteristics with respect to their scalability. We use them as representative of a specific behaviour. 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.

We have generated four workloads using swim, bt, hydro2d and apsi. The idea is having different percentages of each type of application per workload. W1 is composed by applications with super-linear and good speedup. W2 is composed by applications with good and medium speedup. W3 is a mix of the four types, and W4 is composed by applications with good scalability (but not super-linear). We have not performed experiments only with no-scalable applications because we want to evaluate the memory page migration mechanism with parallel applications, and one workload with (for instance) only apsi's will be similar to sequential applications. CPU Time of applications executed on different workloads is different because application execution is really different. In the case of Equipartition and IRIX the only difference is the effect on the concurrently running applications, but in the case of PDPA also the number of received cpus. For this reason, we have normalized the results showing the ratio of the CPU Time consumed with migrations and without it (values lower than 1.0 mean that with memory migrations the application consumes less CPU time.).

(Place Table III here)

Table IV shows the composition of the workloads used. Each cell represents the percentage of the workload filled by each job type. 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%. Real systems execute at different system loads depending on the time period (daytime, night, weekends, holidays). These three system loads allow us to obtain results across all these different usage periods. 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 <sup>(18)</sup>. We also show the total number of instances of applications executed when the load is set to 100%, and the amount of 16-Kbyte pages used by all the applications in the workload. A 16 Kbytes page is the unit of memory migration in IRIX.

(Place Table IV here)

Experiments have been carried out in a real system, a SGI Origin 2000 with 64 250 Mhz. R10000 processors. Each processor has independent 32Kb. first level instruction and data caches and a shared second level 4Mb. cache. Each pair of processors in the machine come with 384 Mb. of main memory, for a total of 12 Gbytes.

## **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 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 times consumed when executing with memory page migrations with respect to those consume 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 by 33% (in average) the CPU time consumed per application.

If we analyse the results per application, the improvement introduced in swim's, bt's, and hydro2d's are 27%, 40%, and 41% 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.

(Place Figure 1 Here)

### 6.1.2 Performance-Driven Processor Allocation

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. 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 the CPU time 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 this case, the default multiprogramming level reached 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%).

(Place Figure 2 here)

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 as 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. On average, a 12% improvement is achieved.

Analysing the effects per application, the memory page migration mechanism improves the CPU time consumed by swim's, bt's, hydro's, and aphis's by 22%, 8%, 16%, and 1% respectively.

The cases of swim's and aphi'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, aphi's have very bad speedup, and PDPA allocates only 2 processors on 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 the CPU time 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 behavior of jobs under the IRIX policy, we observe that jobs suffer a lot of unneeded kernel thread migrations<sup>2</sup>. To give an insight into that, we have measured the number of process migrations, the average burst CPU time, and the number of bursts per cpu comparing IRIX, PDPA and Equipartition in the case of the workload 1<sup>3</sup>.

Table V shows the results obtained from the three policies. As we can see, the native IRIX policy generates many more process migrations than the other two policies. Moreover, IRIX generates bursts of cpu 50 times shorter than Equipartition or PDPA (243 milliseconds 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

---

<sup>2</sup> We have monitored the system activity and extracted the trace files shown in Figure 1.

<sup>3</sup> The rest of the workloads have a similar behavior

by IRIX is also significantly greater than with Equipartition or PDPA. This behavior 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.

(Place Figure 3 here)

(Place Table V here)

Figure 4 shows the behaviour of the workload 1 when executed under IRIX (plot on the top) and Equipartition (plot at the bottom). These plots correspond to the experiments with the system load set to 100%. The plots have been obtained with the Paraver tool <sup>(19)</sup>. The X axis on the plots represents time. The time scale has been set to the same value in both plots. The Y axis represents the different physical processors on which the applications are running (each line one physical cpu). The processes belonging to the same application are represented running on a processor using the same color (in gray scale).

(Place Figure 4 here)

Observing the plot at the bottom of the figure, we see that the Equipartition policy, as it is run by our CPUManager organizes the applications using contiguous physical processors, as much as possible. That way, the visualization shows an organization of blocks of different colors (one application using the same set of cpus during their complete execution). Roughly, each block corresponds to a different application and it represents its threads running on adjacent physical processors. Usually, the same physical processors are used across the complete execution of the application, although the application can be expanded or shrunk depending on the availability of processors. As the algorithm used for the application placement on the physical processors tries also to keep locality, sometimes an application can use two or more sets of consecutive processors.

Observing the plot on the top part of the figure, we observe that the IRIX scheduling policy causes a lot of changes in the assignment of threads to physical processors (bursts of colors are very small and applications do not use consecutive cpus). That way, the visualization reflects the fact that, after running for a few time (0.25 seconds as presented in Table 4, average burst time per cpu) a thread is context-switched to run on another processor. Although the applications try to adjust the number of processors used depending on the load of the system, usually there are a few more threads running than available processors and this causes the IRIX policy to behave that way. The overhead introduced by the context switches and the loose of locality inside each application are the causes of the increasing execution time of the applications and also of the full execution of the workload. As it can be observed, the same workload takes far less execution time when run on the Equipartition policy compared with the execution under IRIX.

## **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 5 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.

(Place Figure 5 here)

Applications executed under PDPA and with the memory migration mechanism activated improve their speedup by 21%, in average. In particular, swim has a remarkable 35% of improvement in its speedup, bt improves its speedup by 12%, hydro2d by 28%, 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 and we compare the increment in the processor allocation with the increment of speedup obtained.

### **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 6 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%.

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 35%. 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 12% 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.

(Place Figure 6 here)

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 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 demonstrated that automatic memory page migration is a valid approach to improve the execution of parallel application when executing in CC-NUMA multiprocessors multiprogrammed systems applying dynamic space-sharing policies. Page migration is a complementary technique to manual page placement, other automatic techniques (page replication), or automatic page placement policies (first-touch).

We have evaluated the performance of the dynamic memory page migration mechanism provided by IRIX under different system loads and three scheduling policies: the native IRIX policy, Equipartition, and Performance-Driven Processor Allocation (PDPA). Equipartition and PDPA are implemented in the NANOS execution environment. Results have been taken in a SGI Origin 2000 with 64 processors because it is quite representative of current CC-NUMA multiprocessor systems and for availability reasons. We consider 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 from processor scheduler that **(1)** jobs maintain **stable** the number and the set of processors in use, and **(2)** that job reaction to processor allocation changes be **fast**, to be able have time to detect the application pattern and migrating 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. In the case of the NANOS execution environment, jobs react quickly to processor scheduler decisions related to processor allocation, avoiding processor sharing between running threads. Moreover, in the NANOS system, the number of process migrations is avoided as much as possible.

Results show that the memory page migration mechanism provided by IRIX, in average, always improves the application performance, that is, applications 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, 33% in the case of Equipartition and 12% 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 page migration mechanism, combined with a dynamic space-sharing 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 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.
- 2 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.
- 3 OpenMP Organization. OpenMP Fortran Application Interface, v. 2.0, <http://www.openmp.org> , June 2000.
- 4 NANOS ESPRIT Project (E-21907), <http://www.ac.upc.es/nanos>
- 5 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>.
- 6 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.
- 7 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.

- 8 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.
- 9 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.
- 10 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.
- 11 Silicon Graphics, Inc. Origin2000 and Onyx2 Performance Tuning and Optimization Guide, Document number 007-3430-002, <http://techpubs.sgi.com>, 2000.
- 12 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>.
- 13 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
- 14 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.

- 15 Felix Freitag, Julita Corbalán, Jesús Labarta, “A Dynamic Periodicity Detector: Application to Speedup Computation”, Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2001), San Francisco, April 23-27, 2001.
- 16 Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. Available at <http://www.spec.org/osg/cpu95>, 1995.
- 17 H. Jin, M. Frumkin, J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report: NAS-99-011, 1999.
- 18 “The Standard Workload Format”, <http://www.cs.huji.ac.il/labs/parallel/workload/swf.html>
- 19 European Center for Parallelism of Barcelona (CEPBA), “Paraver – Parallel Program Visualization and Analysis Tool – Reference Manual”, November 2000, <http://www.cepba.upc.es/paraver>.
- 20 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.
- 21 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.

## **10. Table Captions**

**Table I: Policy Modules Function**

**Table II : Values of the memory migrations parameters used in the OpenMP runtime library**

**Table III: Parallel applications**

**Table IV: Parallel workloads**

**Table V: Process migrations per policy, workload 1 (load=100%)**

## 11. Tables

<i>Function name</i>	<i>Description</i>
pm_filldefault(...)	Fills a policy_set with predefined default values (1)
pm_create(...)	Creates a policy module (2)
pm_setdefault(...)	Selects a new default policy for stack, text, or heap. (3)
pm_attach(...)	Connects a policy module to a virtual address space range (3)

<i>Parameter</i>	<i>Value</i>
Migration enabled	On
Migration threshold	50%
Freeze enabled	Off
Melt enabled	Off
Dampening enabled	Off
Refcnt enabled	Off

<b>Application (input)</b>	<b>swim (ref)</b>	<b>Bt (Class A)</b>	<b>hydro2d (train)</b>	<b>apsi (ref)</b>
Seq. execution time (in seconds)	212	1066	223	99
Speedup with 8/16/32 processors	21.6/36.5/44.2	6.1/12/20	4.6/5.4/6.3	0.9/0.9/0.9

	<b>Swim</b>	<b>Bt</b>	<b>Hydro 2d</b>	<b>Apsi</b>	<b>Total number of applications executed (load=100%)</b>	<b>Total memory requirements (approx.), in 16Kbyte pages (load=100%)</b>
W1	50%	50%	-	-	61	90000
W2		50%	50%	-	48	50000
W3	25%	25%	25%	25%	125	75000
W4	-	100 %	-	-	16	50000

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

## 12. Figures captions

**Figure 1.** Influence of the memory page migration in the execution time under Equipartition

**Figure 2.** Influence of the memory page migration in the Execution Time under PDPA

**Figure 3.** Influence of the memory page migration in the Execution Time under IRIX

**Figure 4** Process migrations. Visualization of workload execution

**Figure 5** Influence of the memory page migration in the application speedup

**Figure 6** Influence of the memory page migration in the processor allocation under PDPA

## 13. Figures











