

# Scheduler-Activated Dynamic Page Migration for Multiprogrammed DSM Multiprocessors<sup>1 2</sup>

Dimitrios S. Nikolopoulos and Constantine D. Polychronopoulos  
Coordinated Science Laboratory  
University of Illinois at Urbana-Champaign  
1308 West Main St., Urbana, IL, 61801-228, U.S.A.  
e-mail: dsn@csrd.uiuc.edu, cdp@csrd.uiuc.edu

and

Theodore S. Papatheodorou  
Department of Computer Engineering and Informatics  
University of Patras  
Rion 26 500, Patras, Greece  
e-mail: tsp@hpclab.ceid.upatras.gr

and

Jesús Labarta and Eduard Ayguadé  
Department d' Arquitectura de Computadors  
Universitat Politècnica de Catalunya  
c. Jordi Girona 1-3, 08034, Barcelona, Spain  
e-mail: {jesus,eduard}@ac.upc.es

Version: November 28, 2001

---

<sup>1</sup>Paper to appear in the *Journal of Parallel and Distributed Computing*, 2002.

<sup>2</sup>This work was supported by the European Commission under the TMR grant ERBFMGECT-950062 and in part by the ESPRIT IV grant No. 21907, the Greek Secretariat of Research and Technology grant No. E.D.-99-566, the Ministry of Education of Spain grants No. TIC-98-511 and TIC97-1445CE, NSF grant EIA-99-75019, a research grant from NSA and a research grant from Intel Corporation. The experiments presented in this paper were conducted with resources provided by the European Center for Parallelism of Barcelona (CEPBA).

The performance of multiprogrammed shared-memory multiprocessors suffers often from scheduler interventions that neglect data locality. On cache-coherent distributed shared-memory (DSM) multiprocessors, such scheduler interventions tend to increase the rate of remote memory accesses. This paper presents a novel dynamic page migration algorithm that remedies this problem in iterative parallel programs. The purpose of the algorithm is the early detection of pages that will most likely be accessed remotely by threads associated with them via a thread-to-memory affinity relation. The key mechanism that enables timely identification of these pages is a communication interface between the page migration engine and the operating system scheduler. The algorithm improves previously proposed competitive page migration algorithms in many aspects, including accuracy, timeliness and cost amortization. Most notably, the algorithm is not biased by obsolete memory access history that may be accumulated in the page access counters at runtime. Experiments on the SGI Origin2000 show that the algorithm outperforms by far the best static page placement algorithm and a customized page migration engine implemented in IRIX, the Origin2000 operating system. The algorithm is implemented at user-level and its functionality is orthogonal to the scheduling policy of the operating system.

*Key Words:* Distributed shared-memory multiprocessors, NUMA, multiprogramming, memory management, page migration, data locality, operating systems, runtime systems.

## Biographies

**Dimitrios S. Nikolopoulos** is a visiting Research Assistant Professor of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. He received his Ph.D. in Computer Engineering in 2000 and the Diploma in Computer Engineering in 1996, both from the University of Patras, Greece. Prior to joining the University of Illinois, Dr. Nikolopoulos was a research associate of the High Performance Information Systems Laboratory at the University of Patras, and a visiting research associate of the European Center for Parallelism of Barcelona (CEPBA). His research interests are in the areas of operating systems, runtime systems, parallel computer architectures, programming languages, and performance evaluation and modeling. He has published more than 30 technical papers in major international journals and conferences, including an award winning paper at Supercomputing'2000 and a nominated paper at Supercomputing'2001. Dr. Nikolopoulos is a member of the ACM and the IEEE Computer Society.

**Constantine D. Polychronopoulos** is a Professor of Electrical and Computer Engineering and a Research Professor of the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. At CSL, he is leading research projects that focus on creating new hardware and software approaches to speed distributed computation. Specific research projects include the design and implementation of novel compiler optimizations for ILP and multithreading, automatic program parallelization, runtime systems and operating system cores for multithreaded and multiprocessor architectures, and multithreaded micro-architecture design and performance characterization. Dr. Polychronopoulos received his Ph.D. in Computer Science at the University of Illinois at Urbana-Champaign in 1986, his M.S. in Mathematics from Vanderbilt University in 1982, and his B.Sc. in Mathematics from the University of Athens, Greece in 1980. His research work has appeared in more than 100 journal, conference and other technical publications, and has been funded by DARPA, ONR, NSA, NSF, and industrial labs. Dr. Polychronopoulos was elected on the Board of Directors of ACM SIGARCH in 1990, and is the recipient of a Fulbright scholarship, the 1989 NSF Presidential Young Investigator Award, and the 1998 Bodossaki Foundation Award.

**Theodore S. Papatheodorou** is a Professor of Computer Engineering and Informatics at the University of Patras Greece. He received his B.Sc. degree in Mathematics from the University of Athens in 1969, his M.S. degree in Mathematics from Purdue University in 1971, his Ph.D. degree in Computer Science from Purdue University in 1973, and a M.S. degree in Civil Engineering from Purdue University in 1975. Prof. Papatheodorou has held several academic positions and served in the advisory board of industries and government organizations in Greece and the United States. Currently, he is leading the High Performance Information Systems Lab at the University of Patras. His research interests include scientific computation, system and applications software for parallel and distributed computing, large-scale computations and their applications, and multimedia. Prof. Papatheodorou is a recipient of an ACM award for his contribution to the ACM International Conference on Supercomputing.

**Jesús Labarta** received the Engineering degree in Telecommunications in 1981, and the Ph.D. degree in Computer Science in 1983 from the Universitat Politècnica de Catalunya (UPC) in Barcelona, Spain. He is a Professor of the Department of

Computer Architecture at UPC since 1990. He has been lecturing on computer architecture, operating systems, computer networks and performance evaluation since 1981. His research interests include parallel computing, multiprocessor architectures, memory hierarchies, parallelizing compilers, operating systems, parallelization of numerical kernels, metacomputing tools and performance analysis and prediction tools. He has lead the technical work of UPC in 15 industrial R&D projects. Dr. Labarta is the director of CEPBA (European Center for Parallelism of Barcelona) since 1995. In CEPBA, he has been highly motivated by the promotion of parallel computing into industrial practice, especially within SMEs, and has been responsible for three clusters of projects on technology transfer, involving 28 subprojects.

**Eduard Ayguadé** received the Engineering degree in Telecommunications in 1986 and the Ph.D. degree in Computer Science in 1989, both from the Universitat Politecnica de Catalunya (UPC) in Barcelona, Spain. He has been lecturing at UPC on computer organization and architecture and optimizing compilers since 1987. He is a Professor of the Department of Computer Architecture at UPC since 1997. His research interests cover the areas of processor microarchitecture and ILP exploitation, parallelizing compilers for high-performance computing systems and tools for performance analysis and visualization. He has published more than 100 papers in these topics and participated in several multiinstitutional research projects, mostly in the framework of the European Union ESPRIT and IST programmes.

## 1. INTRODUCTION

Cache-coherent distributed shared-memory (DSM) multiprocessors are one of the dominant server architectures for industrial and academic environments. The commercial success of these systems is a synergy of three factors: Scalability, which is enabled by the DSM architecture; cost-effectiveness which is enabled by the use of off-the-shelf building components; and a simple shared-memory programming interface, which is enabled by efficient cache coherence protocols implemented in hardware. Unfortunately, the performance of these systems is adversely affected by the non-uniform memory access latency. Sustaining high performance on DSM multiprocessors requires substantial memory access locality optimizations.

In several production settings, DSM servers operate in multiprogrammed mode, in which multiple users submit jobs for execution simultaneously. Multiprogrammed DSM servers suffer from poor performance due to contention between programs for critical system resources, such as processors and memory. The adverse effects of multiprogramming on multiprocessor servers stem from the fact that different types of jobs have different and usually conflicting resource and performance requirements. To cope with the requirements of as many types of jobs as possible, the system scheduler is forced to use aggressive algorithms for job multiplexing, based on frequent preemptions and migrations of threads. These scheduler interventions, albeit necessary for load balancing and processor utilization, introduce complex performance implications, such as unnecessary idling of processors at synchronization points, high cache miss rates, and increased memory access latency due to remote memory accesses [8, 31, 32].

### 1.1. Motivating Example

A significant number of large-scale parallel systems are operated in multiprogrammed mode, allowing more than one users to share the resources of the system at the same time. From the system administration point of view, resource sharing enables higher resource utilization. From the user point of view though, resource sharing implies compromised performance due to limited resource availability. On a multiuser system, a user can not expect that a fixed number of processors and a fixed amount of memory will be available throughout the lifetime of the program. Supercomputing centers and other computational resource providers have tried to circumvent this problem by partitioning parallel systems and letting users submit jobs to isolated partitions through a front-end queuing system, after providing rough indications on the requirements of the jobs in terms of processors, memory and execution time.

The advantage of queuing systems is that they can guarantee that the user will run the jobs on dedicated resources for the reserved time slot. The major disadvantage of queuing systems is that sometimes, users experience unacceptable queuing delays and turnaround times. As a consequence, several users prefer to execute their jobs on time-shared processors, thus trading speedup for productivity. A second disadvantage is that execution on dedicated resources is charged more than execution on non-dedicated resources.

As an example, the NCSA Origin2000 cluster at the University of Illinois [19] is partitioned into 24 dedicated and 12 time-shared partitions of various sizes. Each partition is controlled by a separate submission queue. Users can submit no more than five jobs at a time at either dedicated, or time-shared queues. The experience

of the authors as users of this system<sup>3</sup> has shown that the waiting time in queues that control dedicated processor partitions is anywhere between half an hour and one week. Waiting time on time-shared queues rarely exceeds five minutes. Execution time on dedicated queues is charged with a site-specific algorithm and the amount of CPU hours used is subtracted from the user account balance. Execution time on time-shared partitions is free of charge. Time-shared partitions are used heavily by users that submit parallel jobs and personal communication with users from the academia revealed that a large fraction of these jobs is submitted for production purposes and even benchmarking [24].

Figure 1 shows a snapshot of the jobs running on a 16-processor time-shared partition of the NCSA Origin2000 at 11 a.m. on Nov. 15 2001. This partition admits jobs that use between 9 and 16 threads. Admission control is performed by the LSF queuing system. The figure shows the jobs running in the partition at the time when we submitted a job that uses 16 threads, while logged in as `cpolychr`. This job executed the OpenMP implementation of the NAS MG benchmark, using the Class A problem size. The snapshot shows that the partition is actively shared between four users, while one user is apparently idling. The four active users have submitted one parallel job each. Among the jobs submitted from other users, one uses the Gaussian computational chemistry program that models molecular structures under a variety of conditions, and another uses Abaqus, a proprietary 3-D structural finite element code. Both of them are computation-intensive programs able to exploit massive amounts of both shared-memory and distributed memory parallelism. These programs are used solely for production purposes by engineers in related fields. The submitted jobs bring the degree of multiprogramming of the partition to four. Under the most conservative assumption that the users ask for the minimum number of threads for their jobs, each processor in the partition is time-shared between two or three threads. In the worst case, each processor is time-shared between four threads. Although we can not know the real intentions of the users that submitted these jobs, judging from the characteristics and the run times of the jobs, we can safely assume that the users use the time-shared queue for production purposes and not only for testing and debugging.

The execution time of MG on 16 dedicated processors is 11.86 seconds. The execution time of MG in the job we submitted to the time-shared partition of the NCSA Origin2000 was 68.97 seconds. This is a slowdown of 5.81 compared to the execution time of the same program in a dedicated system. Since there seem to be three other users sharing the processors, we would expect our job to run with 4-fold slowdown. The mediocre performance tells us that the implications of multiprogramming on the performance of the program are beyond user expectations and need deeper investigation.

## 1.2. Problem statement

On cache-coherent DSM systems (also known as ccNUMA systems due to the non-uniform memory access latency), one of the most subtle effects of multiprogramming is the increased frequency of remote memory accesses upon cache misses [7, 32]. This problem is an artifact of the tendency of the operating system scheduler to move threads between different nodes<sup>4</sup>, or share the resources of each node between different jobs. The former is required mostly for load balancing purposes, while the latter is required to dynamically space-share the system resources among jobs in the presence of load variations [30, 33]. Both events lead to a significant

increase of memory latency and occur in essentially all practical scheduling algorithms.

In terms of data locality, the number of remote memory accesses on a multi-programmed DSM multiprocessor increases whenever a thread of a program is not located in the same node with the pages that the thread accesses more frequently. We define this set of pages as the *memory affinity set* of the thread. *Collocation* of each thread with its memory affinity set is the prerequisite for minimizing the rate of remote memory accesses.

Collocation of threads and memory affinity sets can be theoretically implemented either by a scheduling strategy that schedules each thread to the node that stores its memory affinity set, or symmetrically, by a memory management strategy that places each page on the same node with the thread that contains the page in its memory affinity set. Providing efficient memory management mechanisms for dynamic collocation of threads and memory affinity sets, in a manner which is orthogonal to the scheduling strategy of the operating system, is the goal of the work presented in this paper.

### 1.3. Contributions of the paper

This paper presents a new page migration algorithm which dynamically collocates threads and memory affinity sets, in the presence of unpredictable scheduler interventions. The algorithm operates on iterative programs, which constitute the majority of parallel codes in use today. Popular codes used for benchmarking parallel architectures, such as the NAS benchmarks [3], the SPECHPC96 and SPEC2000 benchmarks [2], and most of the application codes from the SPLASH-2 benchmark suite [34] fall into this class. Iterative parallel codes are encountered in the most significant application domains of computational science and engineering, including computational fluid dynamics, molecular chemistry, weather forecasting, ocean and climate modeling, finite element methods, and crash simulation.

Iterative codes have the advantage that their accurate memory access pattern can be detected and optimized at runtime, immediately after the execution of the first iteration [14, 21]. The idea pursued by our algorithm is to collect scheduling information from the operating system and detect events that may necessitate the migration of pages from memory affinity sets which are not aligned with the threads they belong to. The algorithm correlates scheduling information with the memory access trace of the program at runtime and attempts to migrate pages timely, in response to critical scheduler interventions. Correlation of memory accesses with scheduling information enables the algorithm to improve the accuracy and timeliness of its page migration engine. Consequently, the algorithm provides solid performance improvements compared to previously proposed competitive page migration algorithms.

The algorithm is implemented in a commercial system, the SGI Origin2000 and does not require any modifications to the operating system. It is an integral part of a runtime system that provides transparent services for data distribution and memory access locality optimizations to OpenMP programs [22]. This makes the algorithm immediately available to OpenMP codes as a pluggable module.

We present results from experiments using both synthetic and realistic workloads, in which multiple instances of parallel applications from the NAS benchmarks suite [3] are multiplexed with background load, using two different scheduling strate-

gies, time sharing and dynamic space sharing. These scheduling strategies represent the two extremes of the spectrum of algorithms used in contemporary multiprocessor schedulers. The results emphasize three facts: our algorithm achieves sizeable improvements over the best static page placement algorithm; it improves significantly the performance of competitive kernel-level page migration mechanisms; and it is immune to the scheduling strategy of the operating system, i.e. its effectiveness is uniform with different scheduling strategies.

#### 1.4. The rest of this paper

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 formulates the problem of poor memory access locality due to scheduler interventions on multiprogrammed DSM multiprocessors. Section 4 presents our algorithm and Section 5 outlines the implementation. Section 6 provides experimental results. We conclude the paper in Section 7.

## 2. RELATED WORK

We classify the techniques proposed thus far for improving the efficiency of multiprogramming on shared-memory multiprocessors into techniques enabled by the scheduler and techniques enabled by the memory manager.

### 2.1. Scheduler-enabled techniques

Most schedulers of multiprogrammed multiprocessors maintain an affinity link for each thread. This link is set to point to the processor on which the thread begins its execution and is likely to store the working set of the thread in the cache [31]. The link biases scheduling decisions, in the sense that the scheduler tends to bind each thread to the corresponding affinity link. Affinity scheduling improves marginally the performance of time sharing schedulers on lightly loaded multiprogrammed shared-memory multiprocessors [29]. The effectiveness of affinity scheduling in general depends on several dynamic characteristics of the workload. If processors are time shared among multiple threads, affinity scheduling is unlikely to be effective. This happens because with a high degree of multiprogramming, the cache of a processor is unable to store the working sets of the threads that share the processor. Moreover, the use of *hard* affinity links, that is, the binding of threads to processors may introduce load imbalance.

Dynamic processor allocation policies that take into consideration the system load and the degree of parallelism available in each program may also improve the performance of multiprogrammed multiprocessors [18, 30, 35]. These policies are generalizations of static space sharing, a processor allocation scheme that assigns a distinct subset of processors to each parallel job, so that the job executes without interference with other jobs in the system. Dynamic processor allocation implements space sharing but allows processors to move between jobs, to adapt to the dynamics of the system load. It has been shown that on multiprogrammed DSM multiprocessors, dynamic processor allocation schemes tend to outperform scheduling policies based on time sharing [7, 8, 32, 33]. Compared to time-sharing, dynamic processor allocation allocates less processors to parallel jobs, but reduces contention and achieves satisfactory data locality by isolating the resources allocated to each job [33]. In practical implementations, dynamic processor allocation

is coupled with runtime systems that enable the dynamic adaptation of parallel jobs to the number of processors available to them at runtime [25].

One of the most important problems of multiprocessor schedulers is the uncontrolled preemption of threads that execute on the critical path of a program. These preemptions are performed in response to unexpected oscillations of system load. A third class of mechanisms for efficient multiprogramming on multiprocessors is based on the idea of resuming as quickly as possible critical threads which are maliciously preempted by the operating system [1]. These mechanisms use a two-level scheduling architecture, in which the kernel scheduler assigns physical processors to programs and a user-level scheduler maps threads to virtual processors<sup>5</sup>, which are in turn mapped to physical processors by the operating system. The two schedulers communicate via shared memory to exchange scheduling information. The user-level scheduler is notified whenever a thread is preempted by the operating system and has the means to identify if this thread was executing useful computation at preemption time. With this information, the user-level scheduler is able to resume the preempted thread when one of the virtual processors assigned to the program becomes idle.

## 2.2. Memory management techniques

The need to introduce memory management schemes that take into account multiprogramming on contemporary DSM multiprocessors emerges due to the non-uniform latency of memory accesses. Memory allocation algorithms for DSM multiprocessors try to reduce the frequency of remote memory accesses. It has been shown that for parallel programs running in a dedicated set of DSM nodes, a balanced static distribution of pages between nodes achieves usually good levels of memory access locality [7, 17]. To cope with multiprogramming, memory management algorithms must enable the movement of memory resources between programs. Dynamic sharing of memory can be done either by letting the programs claim memory from other programs, or by dynamically migrating memory based on some cost-effectiveness criterion [33]. The latter approach was previously explored in real systems, via the use of dynamic page migration [28, 32, 33].

Dynamic page migration was introduced more than a decade ago on dance-hall NUMA architectures [6, 26]. The idea is to count the number of accesses from each node to each page in memory at runtime, and identify which node accesses each page more frequently. The access counters are compared with a competitive criterion to assess whether migrating a page to the node that accesses the page more frequently is likely to reduce the number of remote memory accesses in the long term [5]. Dynamic page migration was implemented in several systems without clear indications about its effectiveness [6].

The idea of using dynamic page migration was resurrected with the appearance of ccNUMA architectures, as a way to reduce the number of remote memory accesses in programs with dynamic memory access patterns. The first complete implementation of dynamic page migration appeared in a port of the IRIX operating system for the Stanford FLASH multiprocessor [32]. This implementation uses a vector of hardware counters attached to each page in memory. Additional hardware attached to the memory modules compares local to remote accesses to each page and delivers an interrupt if the number of remote accesses by some node exceeds the number of local accesses. The interrupt handler evaluates a set of resource management criteria that have to be met for the page to migrate. If these criteria are

met, the page is migrated and the operating system takes the appropriate actions to move the page and maintain the consistency of its mappings in the TLBs and internal data structures. The Stanford FLASH implementation of dynamic page migration was later adopted for the page migration engine of the SGI Origin2000 [15].

Experiments with a simulator of the IRIX page migration engine revealed that dynamic page migration reduces the number of remote memory accesses and provides substantial reductions of turnaround time in multiprogrammed workloads consisting of CPU-intensive sequential programs [28, 32]. Two implementations of dynamic page migration on real systems (the SGI Origin2000 [15] and the Sun Wildfire [9, 23]) have not demonstrated any noteworthy performance improvements [11]. The reasons for this are the limited accuracy of the page migration algorithms used in the implementations, the sensitivity to transient effects in the workloads and the inability to amortize the cost of page migrations [21]. The work presented in this paper addresses these problems.

### 3. THE IMPLICATIONS OF SCHEDULING DECISIONS ON THE LOCALITY OF MEMORY ACCESSES ON DSM MULTIPROCESSORS

The first part of this section formulates the performance implications of certain scheduling decisions of the operating system on the locality of memory accesses on DSM multiprocessors. The goal is to show how dynamic page migration can be used as the means to overcome these problems. The second part of the section explains why previously proposed competitive page migration algorithms are incapable of dealing with scheduler interventions on multiprogrammed systems.

#### 3.1. Impact of thread migrations and preemptions

Assuming that on a DSM multiprocessor the operating system uses a locality-aware page placement algorithm, each thread is expected to be initially collocated with its memory affinity set. Any scheduler intervention that moves a thread away from its memory affinity set will cause the number of remote memory accesses to increase. There are two such interventions, thread migrations and thread preemptions. We examine each of them in the following paragraphs.

##### 3.1.1. Thread migrations

Thread migrations are used by the scheduler to balance the load of the system. They occur frequently in both time sharing and space sharing schedulers, with or without affinity links, as a consequence of load oscillation.

The problem with thread migrations in DSM systems is illustrated in Figure 2. The migration of  $s$  from node  $A$  to node  $B$  has two consequences. First, the working set of  $s$  has to be reloaded from the cache of a processor on  $A$  to the cache of a processor on  $B$ . If  $A$  stores the pages with the working set of  $s$ , which is the case if the operating system uses a NUMA-aware page placement algorithm, the cache reload has to be performed from the memory of  $A$  with remote memory accesses. Second, if  $A$  contains the pages of the memory affinity set of  $s$ , these pages will be accessed frequently from  $B$  upon cache misses incurred from  $s$ .

### 3.1.2. Thread preemptions

Preemptions of threads occur frequently with both time sharing and space sharing schedulers. In time sharing schedulers, preemptions occur due to the multiplexing of threads on processors and are relatively short-term actions, in the sense that the duration of a preemption is usually as much as the duration of a scheduler time quantum, multiplied by the number of threads that share the processor. This number is expected to be low even on heavily loaded systems. Space sharing schedulers present a different scenario. A thread preemption may occur due to a spike in the load, which forces the operating system to deallocate processors from running jobs and allocate them to newly arrived jobs. In these cases, thread preemptions may be long-term actions that require reaction from the runtime system, to resume preempted threads or redistribute the computation assigned to preempted threads among the running threads. This scenario occurs for example in dynamic processor allocation strategies that use process control [30, 35].

The undesirable consequences of a thread preemption are illustrated in Figure 3. If  $s$  is preempted while running on  $A$  and the computation performed by  $s$  is taken up by  $t$ , the working set and the memory affinity set of  $s$  will be accessed remotely from a processor in node  $B$ .

## 3.2. Using dynamic page migration

Dynamic page migration can correct misalignments between computation and data that might occur due to scheduler interventions and maintain dynamic thread-to-memory affinity relations at runtime. Upon a thread migration, a page migration engine can move the pages of the memory affinity set of the migrated thread to the node that hosts the thread after the migration. Upon a thread preemption, page migration can be used to forward the pages in the memory affinity set of the preempted thread close to the active threads that take up the preempted computation, if any. In both cases, migrations of pages should be performed *timely*, that is, shortly after the scheduling events that trigger them. At the same time, the cost of page migrations should be *amortized*. This means that page migration should be initiated only if it is likely to reduce the latency of remote memory accesses by an amount that exceeds the cost of moving the pages and maintain memory consistency thereafter.

Previously proposed page migration algorithms [28, 32] are based on competitive criteria. A competitive criterion migrates a page if the number of remote accesses by some node exceeds the number of local accesses by a predefined threshold. This threshold is set to account for the cost of page migration. Competitive criteria anticipate that the memory access pattern of a program exhibits temporal locality, so that the history of accesses to each page is likely to reflect the access pattern of the page in the near future. Unfortunately, a competitive criterion is unable to migrate pages timely in response to scheduler interventions in a multiprogrammed system. This happens because the past access history of pages becomes obsolete upon certain scheduling events and biases the page migration criterion towards poor decisions.

Consider a thread migrating from node  $A$  to node  $B$  and a page in the memory affinity set of the thread which has been accessed  $nacc_A$  times from  $A$  and  $nacc_B$  times from  $B$ , until the execution point at which the thread migrates. If the migrated thread keeps running on  $B$  for a relatively long time frame, migrating the

pages of its memory affinity set to  $B$  is worthwhile. The problem is that when the thread migrates, the access history of the pages in the memory affinity set of the thread is most probably indicating that  $nacc_A \gg nacc_B$ . If  $B$  does not access the page at least  $nacc_A - nacc_B$  times, the competitive criterion will not migrate the page to  $B$ . Consequently, the rate of remote memory accesses made by this thread will increase sharply.

Consider now the case in which the thread is preempted on  $A$  and some thread running on  $B$  takes up the computation of the preempted thread. If no other thread is running on  $A$ ,  $A$  will stop accessing the page. On the other hand,  $nacc_B$  will be increased at a fast pace due to the implicit migration of computation to  $B$ . Again if  $nacc_A \gg nacc_B$  at preemption time,  $B$  is unlikely to receive the page unless it accesses it at least  $nacc_A - nacc_B$  times.

The preceding examples show that in order to perform page migrations timely in a multiprogrammed DSM system, it is necessary to discard obsolete history that may be accumulated in the access counters of the pages. In the next section we present an algorithm that achieves this goal by correlating the snapshots of the access counters with scheduling events that occur at runtime and the structure of the program. Correlation with scheduling events enables the algorithm to decide whether the past access history of each page should affect page migration decisions or not. Information from the scheduler lets the algorithm detect accurately the nodes to which pages should migrate at runtime. Correlation with the structure of the program enables the algorithm to make some safe assumptions on the expected memory access pattern at runtime.

#### 4. PAGE MIGRATION ALGORITHM

We propose a scheduler-activated dynamic page migration algorithm designed to dynamically collocate threads and memory affinity sets in multiprogrammed DSM systems. The algorithm has the following properties:

- a.* The algorithm operates on strictly iterative parallel programs, that is, programs that repeat the same parallel computation for a number of iterations. These programs represent the majority of industrial and benchmarking parallel codes in use today. The algorithm exploits the iterative structure of the programs to infer the memory access pattern at specific points of execution.
- b.* The algorithm operates in a local scope and suggests changes of the physical location of pages in the virtual address space of the program. These changes are interpreted as directives to the operating system memory manager, which may accept or reject them according to its internal resource management policy.
- c.* The algorithm intercepts critical scheduling events at runtime. For this purpose, the algorithm uses feedback from the operating system, retrieved via a lightweight communication interface with the kernel scheduler.
- d.* The algorithm uses an aggressive, speculative page migration criterion, activated by scheduler interventions that necessitate the migration of pages.
- e.* The algorithm is orthogonal to the scheduling strategy of the operating system. In particular, the algorithm responds to low-level scheduling decisions that change the state of threads under certain time constraints. These scheduling decisions may

be used by different schedulers, spanning the range between time sharing and space sharing.

The algorithm consists of two modules. The first module identifies the status of threads in the program at specific points of execution. The second module is executed at the end of outer iterations of the program and applies migration criteria to pages that belong to the resident set of the program.

#### 4.1. Detecting the scheduling state of threads

The algorithm takes snapshots of the scheduling state of each thread from the operating system and uses these snapshots to maintain medium-term scheduling information. The purpose of the algorithm is to identify events that may trigger the migration of pages in the memory affinity sets of specific threads. There are three such events: Migration, preemption and resumption of a thread. In all three cases, the algorithm maintains approximate timing information, by timestamping the points at which the scheduling state of threads changes. The purpose of timestamping is to measure approximately the time period during which each thread remains in the same scheduling state and evaluate if this period is long enough to justify the cost of page migration (cf. Section 4.2).

The code fragment in Figure 4 shows how scheduling information is maintained by the algorithm on a per-thread basis. The code assumes that at the time of invocation, the operating system provides for each thread a hint indicating whether the thread is preempted or running on some node. Figure 5 shows the state diagram maintained by the algorithm.

According to the information provided by the operating system, the algorithm identifies the state of each thread as follows:

*running.* This state implies that the thread runs on a CPU on the same node for at least as much as a predefined interval (denoted by  $t_{thr}$ ). This interval is defined by the page migration criterion (cf. Section 4.2).

*migrated.* This state implies that the thread was previously running on a CPU on some node denoted by *curr\_node* and subsequently migrated to a CPU on node  $n \neq curr\_node$ . A thread may transit to the *migrated* state from any of the other states, as long as  $n \neq curr\_node$ .

*preempted.* This state implies that the thread was preempted from the node on which it was running before.

*resumed.* This state implies that the thread was resumed on a processor on the same node it was running before (i.e. *curr\_node*), but before the resumption, the thread was preempted for at least as much as  $t_{thr}$ . Note that if the thread is resumed on a node other than *curr\_node*, its state becomes *migrated*.

Every transition between different states is timestamped by the algorithm. Therefore, the state of a thread  $i$  is interpreted as (*running*|*migrated*|*preempted*|*resumed*) since  $T_i.timestamp$ . The fields *curr\_node* and *prev\_node* track the execution history of the thread and they are used by the page migration criterion.

Out of the four states maintained by the algorithm, three (*migrated*, *preempted*, *resumed*) are treated as triggers that activate an aggressive page migration criterion. Note that the code that maintains scheduling information performs no transi-

tion to the running state. This transition is performed by the code that implements the page migration criterion, as soon as it detects that a thread keeps running on the same node for longer than  $t_{thr}$ .

## 4.2. Page migration criterion

The page migration criterion used in our algorithm is based on the observation that an iterative program repeats its memory access pattern throughout its execution lifetime. This implies that a snapshot of the page access counters taken immediately after the execution of the first iteration reflects the memory access pattern of the entire program. In fact, this snapshot is sufficient to compute the optimal page placement with respect to as far as the relation between threads and memory affinity sets is concerned. Using this snapshot, a competitive page migration criterion can place each page in the node that minimizes the maximum latency due to remote memory accesses to this page [21]. Given the fact that the snapshot is taken early in the execution of the program, it can serve as the means to approximate the best static page placement algorithm.

If the memory access pattern of the program is repeated in every iteration, each page is accessed the same number of times in every iteration, unless it is migrated from the operating system. If the reference rate by some node to this page changes in consecutive iterations, it is likely that an intervention of the operating system has changed the scheduling state of the threads running on that node. More specifically, if the access rate by some node drops, it is possible that a thread that was running on a processor on that node was preempted. With the same rationale, if the number of accesses by some node increases, it might be the case that either a thread that references the page frequently has migrated to a processor on that node, or some piece of computation that accesses the page frequently was taken up by another thread running on that node. These observations drive the page migration criterion used in our algorithm.

The code fragment in Figure 6 shows the page migration criterion. This code is invoked at the end of every outer iteration of the program and checks all the resident pages of the program. The page migration criterion is applied, if the algorithm detects that in two consecutive iterations the access rate from the home node of the page drops, while the access rate from one or more remote nodes increases. Among these nodes, there is a node  $k$  with the maximum number of accesses to the page during the last iteration. After finding this node, the algorithm examines three cases:

*Case 1:* A thread migrated from the home node of the page to node  $k$  and runs there for at least  $t_{thr}$  time units. Due to the increased access rate from  $k$ , it is likely that this thread includes the page in its memory affinity set. The algorithm optimistically moves the page to  $k$ , anticipating that the thread will continue running on  $k$  in the near future.

*Case 2:* A thread was preempted from the home node of the page for at least  $t_{thr}$  time units. Due to the increased access rate from  $k$ , it is likely that a part of the computation performed by the preempted thread was taken up by another thread running on  $k$ . The algorithm moves the page to  $k$ , speculating that the threads that used to access the page more frequently will remain preempted in the near future and that the thread running on  $k$  will adopt the page in its memory affinity set.

*Case 3:* A thread was preempted from  $k$  and was resumed on a processor on the same node after at least  $t_{thr}$  units. In that case, the algorithm speculates that the resumed thread used to contain the page in its memory affinity set before its preemption and moves the page to  $k$ .

The threshold  $t_{thr}$  ensures that a thread that claims a page runs on the same node long enough to justify the cost of migrating the page to that node. The value of  $t_{thr}$  is set equal to  $t_{mas}$ , where  $t_{mas}$  is the time needed to migrate the entire memory affinity set of a thread. The rationale behind this heuristic is the following. The three events that trigger page migrations require at most as many page migrations as the size of the memory affinity set of a thread which is migrated, preempted, or resumed. Clearly, if the thread that claims a page keeps executing on node  $k$  for less than  $t_{mas}$ , there is no time to amortize the cost of migrating the pages to  $k$ . If the thread keeps executing on  $k$  for more than  $t_{mas}$ , the pages will move to  $k$  in time to reduce some of the latency of memory accesses issued by processors in  $k$ <sup>6</sup>. Since the algorithm does not know in advance how long each thread will keep running on the same node, it chooses to competitively wait for  $t_{thr}$  before migrating pages to  $k$  [13].

A low-overhead estimation of  $t_{mas}$  can be obtained as follows: Assuming that the operating system performs a balanced distribution of pages between nodes, the memory affinity set of each thread is expected to be equal to the size of the resident set of the program, divided by the number of nodes on which the program starts executing. The value of  $t_{mas}$  is obtained by multiplying this number with the expected cost of a page migration, which can be computed easily with microbenchmarks.

Note that the algorithm changes the state of *migrated* and *preempted* threads to *running*, after completing the migration of pages in the memory affinity sets of these threads. The *running* state implies that a thread and its memory affinity set are effectively collocated and no further page migrations are required.

### 4.3. Discussion

The presented algorithm infers the memory affinity set to which each page belongs from the observed changes in the memory access pattern. Most of these speculations are correct and lead to accurate page migration decisions, if the operating system uses a locality-conscious static page placement algorithm, such as first-touch [17] and the programs have relatively regular memory access patterns. In these cases, the operating system establishes a strong correlation between threads and pages near the beginning of execution and the page migration algorithm can react smoothly to scheduler interventions.

The situation is more complicated for programs with irregular memory access patterns, in which the correlation between threads and pages may change dynamically at runtime. A solution in these cases is to apply a competitive page migration criterion in conjunction with the scheduler-activated criterion, to track potential changes in the memory access pattern. Incorporating a competitive criterion in the algorithm is straightforward. The access counters of each page that does not satisfy the default migration criterion of the algorithm can be re-evaluated with the competitive criterion to check if migration due to a change in the memory access pattern is worthwhile.

A second consideration is the applicability of the algorithm in non-iterative

programs or iterative programs with non-iterative memory access patterns. In its current form, the migration criterion employed by the algorithm uses the access rate to each page on a per-iteration basis. In case the memory access pattern of the program is not iterative, a simple solution is to use a periodic mechanism. The mechanism checks the access rate to each page during regular time intervals, by taking snapshots of the access counters periodically. The algorithm compares the access rates to each page during two consecutive intervals and subsequently applies the migration criterion. The period between consecutive invocations should be reasonably long to compensate for possible transient effects in the page access pattern or short-term scheduler interventions. Selecting the most appropriate period depends mostly on the characteristics of the program.

One technique that can enhance the performance of a periodic page migration engine is the *decaying* of counters. The intuition behind decaying is to discard or reduce the importance of memory access history which is likely to be obsolete. A access counter can be decayed either by resetting its value to 0, or by reducing its value by some fixed amount. Since the page access pattern of a non-iterative program does not re-iterate itself, progressive decaying of counters during the execution of the program makes the page migration criterion less prone to obsolete memory access history. As a consequence, the counters can not bias the page migration criterion.

In the context of our algorithm, at least two scheduler interventions can be considered as reasons for counter decaying. Upon a thread migration or preemption, the counter of the node from which the thread migrates can be decayed if there are no other threads of the program running on the same node. Since this node will not access the page in the near future, decaying the corresponding counter will enable earlier migration of the page. Results presented in [20] prove that the decaying technique works very well in practice.

## 5. IMPLEMENTATION DETAILS

The previously presented algorithm is implemented at user-level and integrated in a runtime system that provides transparent data distribution facilities and memory locality optimizations to OpenMP programs on the SGI Origin2000 [22]. In this section we describe briefly the most essential implementation details of the algorithm.

### 5.1. User-level implementation

Each page in the memory of a node on the Origin2000 is equipped with a vector of access counters. There is one access counter per node of the system for configurations of up to 64 nodes. For larger configurations, accesses from eight nodes are multiplexed on the same counter. The memory controller is equipped with additional logic that compares local and remote accesses to each page and delivers a hardware interrupt when the number of remote accesses to a page exceeds the number of local accesses by a programmable threshold.

User-level implementation of page migration algorithms on the Origin2000 is enabled by two kernel services. IRIX provides user-level access to the hardware page access counters and scheduling state maintained by the operating system, through the `/proc` filesystem. In addition to this service, IRIX has a complete

*memory management control interface* exported to the user. Although both features are platform-specific, similar services are provided in other commercial cache-coherent DSMs, such as the Sun Wildfire [9, 23]. Furthermore, both commodity and open source operating systems such as TruUNIX and implementations of Linux backed up by SGI, IBM, HP and Sun [16] are currently incorporating features to support memory management services for NUMA systems, such as locality-based thread scheduling and demand-based page placement and migration. There are also ongoing efforts to formalize a set of portable NUMA extensions for commodity operating systems [10]. The proposed interfaces provide a rich set of user-level calls for mappings threads to processors and pages to memory nodes (such as the portable `getcpu()` interface and the `CpuMemMap` and `CpuMemSet` interfaces proposed for Linux and most vendor implementations of UNIX [10]). Given these efforts, we envision a portable implementation of our algorithm for generic support of effective dynamic page migration on multiprogrammed NUMA systems.

The `/proc` filesystem provides a low-overhead interface to the internal state of the hardware and the operating system. Information on the internal state of the system can be obtained via `ioctl()` calls to memory-mapped files in the `/proc` filesystem. There are two `ioctl` commands that enable access to the hardware and the software-extended page access counters of the Origin2000. The software-extended counters are maintained by the operating system upon overflow of the corresponding hardware counters and they are also used to maintain counter values on a per-process basis upon context switches [27]. Note that our algorithm triggers migrations after making explicit comparisons between the values of the hardware counters and evaluating the scheduling state of threads. The interrupt-driven page migration engine of IRIX is disabled while the algorithm is executing.

The memory management control interface of IRIX provides a namespace for the physical memory of the system. This name space can be used to explicitly place and migrate virtual memory pages of the data segment of a program [27].

The main drawbacks of the user-level implementation is the overhead and the lack of complete control over the system resources. The overhead stems from the system calls required to access the `/proc` filesystem and migrate pages. This cost does not harm performance in our algorithm, since both services are invoked at the relatively coarse-grain frequency of once per iteration. The lack of control of system resources appears to be a more significant limitation, mainly because the runtime system is unable to control directly the physical memory resources allocated to other programs and page migrations to loaded nodes may fail under high memory pressure.

Fortunately, in our implementation, memory pressure and similar situations are handled conveniently by the operating system, which accepts or rejects requests for page migration based on the memory occupancy of each node. IRIX applies a best-effort strategy to handle user-level migration requests. If a page can not be accommodated on a node due to shortage of free memory space, the operating system attempts to forward the page to a neighboring node<sup>7</sup> and minimize the latency of remote accesses to the page.

## 5.2. Intercepting scheduling state at user-level

In our implementation, the scheduling state of each thread is communicated from the operating system to the user space via shared memory. Asynchronous communication through shared memory between user-level libraries and the kernel

scheduler can be implemented in IRIX with the `schedctl()` interface. Using this interface, a thread can find out the actual processor on which it was most recently scheduled by the operating system, after polling the `t_cpu` field of the thread-private data area maintained in the kernel [27]. In this way, the runtime system can collect per-thread scheduling information in a distributed fashion. Each thread that shares the address space of the program executes `schedctl()` and subsequently updates a private entry in a scheduling table maintained by the runtime system. In the actual implementation, each thread is assigned a unique entry in the table, indexed by the thread number. Virtual numbers are assigned to threads by the OpenMP runtime environment and lie in the range between 0 and `OMP_MAX_THREADS-1`. No mutual exclusion between threads is required to access the table and the entries of the table are padded to avoid false sharing.

Two important implementation issues are the polling frequency and the identification of thread preemptions. Since scheduling information is obtained on a per-thread basis, the scheduling state of threads in the runtime system can be updated only when the threads are actually running. In our implementation, threads poll scheduling information at the entry and exit points of parallel constructs identified by the OpenMP `!$OMP PARALLEL`, `!$OMP PARALLEL DO` and `!$OMP PARALLEL SECTIONS` directives. Polling at these execution points is convenient for two reasons: First, it does not interfere with the execution of useful computation and introduces negligible overhead. Second, this sort of polling captures medium-term scheduling information for the threads and enables the page migration algorithm to optimize locality across disjoint, coarse pieces of parallel computation. This implementation has better stability and its behaviour is more easily interpretable compared to an implementation that polls at arbitrary points of execution.

Intercepting thread preemptions is a subtle issue, because the operating system does not provide notifications of preemptions via the `schedctl()` interface. In order to overcome this problem, the implementation identifies thread preemptions implicitly. We use a virtual timestamp per thread and take advantage of the fact that the main thread which is responsible for the execution of the sequential parts of the program (denoted as the *master* in OpenMP terminology), participates always in the execution of parallel constructs and therefore polls always its scheduling status. When a thread participates in the execution of a parallel construct, it compares its timestamp with that of the master thread at the entry of the parallel construct. If the value of the timestamp is lower, the thread equalizes its timestamp with that of the master thread. At the exit point the master thread reads the timestamps of the slave threads and if a thread's timestamp is lower than that of the master's, the thread is marked as preempted, based on the observation that the thread did not participate in the execution of the preceding parallel construct. The master increments its timestamp and proceeds normally.

### 5.3. Reducing the overhead of page migrations

The final implementation issue that needs to be discussed is the overhead of page migrations. In general, page migration is an expensive operation, involving data copying, maintenance of TLB coherence, invocation of interrupt handlers and bookkeeping costs. On state-of-the-art architectures, page migrations cost roughly 0.5 to 1 ms [32]. In particular, on the Origin2000, measurements with microbenchmarks have shown that the average cost of a page migration requested at user-level is approximately 1 ms.

Our implementation tries to amortize this cost by overlapping the execution of page migrations with the execution of program code. This is possible by spawning a thread to execute page migrations. However, considering the fact that the algorithm operates in multiprogrammed environments and spawning a thread may stress the system, the implementation uses a conditional and spawns the thread that overlaps page migrations only if the system load is below a certain threshold. If the system load is high, page migrations are inlined in the application code by the master thread.

## 6. EXPERIMENTAL RESULTS

### 6.1. Setting and methodology

We conducted experiments on a 64-processor SGI Origin2000 located at the European Center for Parallelism of Barcelona (CEPBA). This system has 32 dual-processor nodes, organized in a 4-dimensional hypercube topology, with two nodes per edge of the hypercube. The processors of the system are MIPS R10000 clocked at 250 MHz with 32 Kilobytes of split primary cache and 4 Megabytes of unified secondary cache. The system has 12 Gigabytes of memory distributed uniformly across the nodes.

The experiments were conducted with multiprogrammed workloads, submitted to the system in the form of simple C shell scripts. Each workload includes multiple copies of a single parallel benchmark and a sequential program which serves as background noise. The parallel benchmarks used are the OpenMP implementations of the NAS benchmarks [12]. These benchmarks are well tuned for the Origin2000 memory hierarchy and scale almost linearly up to 32 processors. The sequential program in the workloads is a `make` utility invoked in a repetitive manner to compile the files in the `gcc` compiler source tree. This program is I/O intensive and consists of many sequential threads with short execution times. These threads interfere in a random manner with the threads of the parallel programs in the workloads.

We conducted three sets of experiments. The first set includes synthetic experiments which evaluate the impact of scheduler interventions on the performance of parallel programs. The other two sets of experiments evaluate the performance of our page migration algorithm with two scheduling strategies, namely time sharing and dynamic space sharing with process control [30]. These scheduling strategies represent the two extremes of the spectrum of policies used in contemporary shared-memory multiprocessor schedulers. In all three sets of experiments, we ran the entire NAS benchmark suite and measured the performance degradation of the benchmarks when executed in a multiprogrammed system. For the sake of brevity, we use the results obtained for two application benchmarks, BT and SP, to discuss the relevant issues. The rest of the results are given in Appendix A.

The performance achieved by the first-touch page placement algorithm serves as a baseline for comparisons. First-touch is the default page placement algorithm of IRIX. The performance of our algorithm is also compared against the performance of the IRIX page migration engine. IRIX uses a competitive page migration algorithm based on the algorithm implemented on the Stanford FLASH multiprocessor [32], with the exception that the IRIX page migration engine is operating on a per-program basis, rather than as a global system option.

## 6.2. Impact of thread migrations

The first experiment explores the impact of thread migrations on the performance of parallel programs. In this experiment, BT is executed on an idle system. The code of the benchmark is modified to execute the first half iterations on 32 idle processors and the remaining iterations on 16 out of these 32 processors. Before executing the second half of the iterations, 16 threads of the benchmark are artificially preempted and their computation is equally redistributed among the remaining 16 active threads<sup>8</sup>. The pages in the resident set of the benchmark are distributed with the first-touch page placement algorithm of IRIX.

Since the pages are placed on a first-touch basis and the distribution of computation in the parallel loops of the benchmark is static, the pages in the resident set of the benchmark are equidistributed among the nodes that participate in the parallel execution. Due to the preemption of threads on half of these nodes, approximately half of the pages of the resident set (i.e. those that belong to the memory affinity sets of the preempted threads) are accessed remotely during the second half of the benchmark.

Let  $T_1$  and  $T_2$  be the execution times of the two halves of the benchmark. Assuming that  $t_p$  is the parallel execution time of the benchmark on  $p$  idle processors, the optimal execution time of the benchmark in the specific experiment is  $T_{opt} = T_1 + T_2 = \frac{t_{32}}{2} + \frac{t_{16}}{2}$ . Due to the redistribution of the computation assigned to the preempted threads, the rate of remote memory accesses issued by the non-preempted threads is increased and  $T_2$  is increased accordingly. It is therefore expected that  $T_2 > \frac{t_{16}}{2}$ . In order to approximate the optimal performance, a page migration engine should migrate all the pages that belong to the memory affinity sets of the 16 preempted threads. These page migrations should be performed immediately after the preemption of these threads.

In order to investigate how thread preemptions may increase the number of remote memory accesses, we tracked the rate of remote memory accesses per iteration to one of the most frequently accessed arrays (`rhs`) in BT during the execution of the benchmark. The number of remote memory accesses was recorded in our runtime system, by reading the access counters of the pages that stored the array once per iteration. The per-iteration access rates were aggregated over chunks of 20 iterations to produce the chart in Figure 7.

The chart shows the rate of remote memory accesses in three settings of the experiment. The first setting uses the first-touch page placement algorithm. The second setting uses first-touch and activates the IRIX page migration engine. The third setting uses first-touch and our user-level page migration engine. Concentrating on the results without page migration, what can be observed easily from the chart is that immediately after the 100th iteration, there is a spike in the rate of remote memory accesses. More precisely, the rate of remote memory accesses increases by a factor of 3.3, from 363/iteration to 1186/iteration. Figure 8 shows the impact of this anomaly on the execution time of the benchmark. The benchmark executes 43% slower compared to the optimal execution rate (i.e.  $T_1 + T_2 = 1.43 \cdot T_{opt}$ ). The last half iterations execute 62% slower than the optimal execution rate (i.e.  $T_2 = 1.62 \cdot \frac{t_{16}}{2}$ ). Note that this slowdown is attributed solely to remote memory accesses. There is neither sharing of processors, nor any other source of contention during the execution of the benchmark.

The observed trends do not change when the IRIX page migration engine is activated in the experiment. Page migration reduces slightly the rate of remote

memory accesses during the first half (from 363/iteration to 318/iteration). This is a second-order effect and stems from suboptimal placement of some pages by the first-touch algorithm. During the critical second half, the rate of remote memory accesses increases to 892/iteration. This is 24% less compared to the rate without page migration, but still 2.8 as much as the remote memory access rate during the first half. There is a 28% slowdown compared to the optimal execution time of the whole benchmark and a 40% slowdown compared to the optimal execution time of the second half.

The problem appears to be solved by our page migration algorithm. The rate of remote memory accesses remains unaffected by the preemption of the 16 threads<sup>9</sup>. The execution time of the benchmark is only 1% slower than the optimal.

Table 1 shows the size of the resident set and the number of page migrations executed by the IRIX page migration engine and our user-level page migration engine respectively. This information was collected by instrumenting the runtime system to record page migrations. In the case of IRIX, we used the command `nstats -r` to collect postmortem statistics of page migration activity from the IRIX kernel. Ideally, half of the pages of the resident set should be migrated after the 100th iteration. Our page migration algorithm approaches this optimal point, by migrating 47% of the resident pages after the 100th iteration. The IRIX page migration engine migrates only 24% of the resident pages. It must be emphasized that IRIX detects a total of 1291 pages as candidates for migration, but 544 of these pages are not migrated because they fail to satisfy the constraints posed by the IRIX page migration policy. It seems that although IRIX detects almost as many candidates for migration as our page migration algorithm, it fails to migrate all of these pages.

Although more accurate information about the behaviour of the IRIX page migration engine could not be obtained at runtime, we believe that the observed behaviour is an artifact of the obsolete access history of pages that biases the IRIX page migration algorithm after the preemption of threads. Note that IRIX does make use of counter decaying. In particular, IRIX decays the counters of one page every 10 ms, by subtracting a fixed quantity from the corresponding access counters. At this rate however, in order to decay the counters of all the pages that have to be migrated (1528 pages in total), IRIX needs approximately 15 seconds. Even if the counters of all pages are decayed, it is not guaranteed that the pages will eventually migrate. Considering that the total execution time of the benchmark is 92 seconds, it becomes obvious that the IRIX page migration engine can not migrate pages in a timely fashion.

### 6.3. Time sharing

In the second experiment, we executed two multiprogrammed workloads using the native time sharing scheduler of IRIX. IRIX uses an earnings-based scheduler which attempts to equalize the CPU time allocated to each job in the long term [4]. The first workload includes two copies of BT and a loop enclosing invocations to `make`, which are issued for the entire duration of the execution of the workload. Similarly, the second workload includes two copies of SP and a loop with invocations to `make`. The parallel programs in the workloads use 32 threads and this number remains fixed throughout the execution of the programs. There is no binding of threads to processors or any other intervention that alters the standard behaviour of the IRIX scheduler.

The average turnaround time of BT and SP in the workloads with different memory management schemes is illustrated in Figures 9 and 10 respectively. The optimal execution time of the benchmarks in these experiments is computed as the execution time on 32 idle processors, multiplied by the degree of multiprogramming in the workload, which is equal to 2. This is true under the oversimplifying assumption that there is no interference with the sequential background load of the threads spawned by `make`.

In this experiment, the user who submits the parallel programs for execution would expect the programs to run with a graceful slowdown approximately equal to 2. Unfortunately, the results show that the slowdown of the benchmarks is much worse. Using first-touch page placement, the actual slowdown is 4.4 and 4.0 for BT and SP respectively. Using page migration in the IRIX kernel does not seem to remedy the problem and the slowdown remains high (4.1 and 3.1 for BT and SP respectively).

During the execution of the workloads, we recorded the transitions of threads between scheduling states, as retrieved from the IRIX scheduler. This information is summarized in Table 2. The recorded information indicates that time sharing incurs a remarkably high number of thread preemptions. Most of these preemptions are short-term and they most likely occur due to the interference with the background load. A very small fraction (less than 1%) of these preemptions lasts for more than  $t_{thr}$ . Note that the size of the memory affinity set of each thread in BT and SP is approximately 96 pages. Assuming an average page migration cost of 1 ms,  $t_{thr} = t_{mas} \approx 96$  ms. On the other hand, a significant amount of thread migrations is recorded and almost half of them lasts for more than  $t_{thr}$ . IRIX uses affinity links, trying to keep each thread running on the same processor as much as possible. Unfortunately, the interference with the background load forces many threads belonging to parallel programs to lose their affinity, by moving to other nodes for load balancing purposes. Many of these threads return back to their original home nodes, however their affinity links are weakened in a manner that hurts performance.

The overall picture shows that there is significant room for improvement from page migration. Indeed, our page migration algorithm is able to reduce the slowdown of the parallel programs to 2.4–2.5. Although there is still a noticeable performance drop, the page migration algorithm delivers almost as much of an improvement as possible. We justify this argument by the results of a synthetic experiment, in which we executed the same workloads without the background noise and after binding threads to processors. The slowdown in this case was 2.23 for BT and 2.27 for SP. This slowdown occurs solely due to the interference between threads in the caches, since there are no thread migrations. There is only a marginal difference between this slowdown and the slowdown measured in the experiments with the background noise and this difference is apparently due to the intervention of the background noise in the caches.

#### 6.4. Dynamic space sharing

The experiments presented in the previous section were repeated after replacing the IRIX time sharing scheduler with a dynamic space sharing scheduler. We emulated dynamic space sharing following guidelines from previous implementations of dynamic processor allocation policies at user-level [35]. We allocate a page in shared memory in which a load index is consistently updated by the programs in

the workloads. This load index reflects the actual degree of parallelism exposed by each program at any point during its execution. The programs request 32 processors at the entry points of `!$OMP PARALLEL` constructs and 1 processor otherwise. The number of processors allocated to each program is controlled by an equipartitioning algorithm. With this strategy, processors are allocated to programs cyclically and one at a time, until the requests of all programs are satisfied or the system runs out of processors [30]. Once the number of processors allocated to a parallel program is determined, the program uses the same number of threads to execute parallel constructs. In this way, the program adapts effectively to the available number of processors.

The actual mapping of threads to processors in this experiment is left to the IRIX scheduler. The affinity links used by the IRIX scheduler were enabled to bias scheduling decisions.

Figures 11 and 12 plot the results of this experiment. The optimal turnaround time is again equal to  $2 \cdot t_{32}$ . It can be easily observed that the performance implications of multiprogramming become less acute with dynamic space sharing. However, the performance trends observed with time sharing occur in space sharing as well. Static page placement without page migration has poor performance, while competitive page migration is again insufficient. Our page migration algorithm performs only 3-6% off the optimal performance.

Table 3 provides scheduling statistics to enlighten the results. Dynamic space sharing incurs less preemptions and migrations of threads compared to time sharing, which is fairly expected. Nevertheless, dynamic space sharing incurs significant scheduling activity due to the dynamics of the workload. In an ideal space sharing setting, each parallel program should receive 16 dedicated processors and run on them throughout its lifetime. There are several reasons due to which this ideal distribution is impossible in the experiments. First, the background load requires at least one processor throughout the execution of the workload. Second, there is an inherent asynchrony in the execution of parallel programs. The executions of the two copies of a program in the same workload do not proceed synchronously through the same phases. There are cases in which one parallel program executes sequentially, while the other has as many as 30 processors to use. Asynchrony is introduced by the background noise. As a consequence, thread preemptions and migrations are relatively frequent.

The notable difference between space sharing and time sharing, is that most changes of scheduling state last long enough to activate our page migration algorithm. This happens because space sharing tends to change the scheduling states of threads across coarse-grain pieces of computation, that is, parallel loops and regions. Therefore, dynamic space sharing provides a more stable scheduling environment for applying aggressive page migration strategies.

## 7. CONCLUSIONS

This paper presented a scheduler-enabled dynamic page migration algorithm for multiprogrammed DSM multiprocessors. The algorithm was motivated by the fact that on cache-coherent DSM multiprocessors, certain scheduler interventions tend to increase the frequency of remote memory accesses, by moving threads away from their memory affinity sets. The presented algorithm intercepts medium-term scheduler interventions, evaluates whether any of these interventions justifies the cost of migrating an entire memory affinity set and activates page migration accordingly.

The algorithm is customized for iterative programs in which the accurate memory access pattern can be inferred after the execution of one iteration.

The most notable advantages of the presented algorithm are timeliness, accuracy and ability to amortize the overhead of page migration. The algorithm coordinates effectively the scheduler and the memory manager, by being responsive to scheduling events which are likely to benefit significantly from page migration. Page migrations are based on accurate snapshots of the complete memory access pattern of the programs and after discarding any obsolete memory access history. The cost of page migrations is carefully balanced against the expected benefits from reducing the rate of remote memory accesses. The algorithm can be extended easily to non-iterative programs using sampling and access counter decaying.

The implementation of the algorithm lies entirely at user-level and the algorithm can be plugged to OpenMP programs without modifications of source code. The performance of the algorithm is superior to the best static page placement algorithm, even when the latter is coupled with a competitive page migration engine. The algorithm provides uniform performance improvements with the two scheduling policies that represent the extremes of the spectrum of policies in contemporary multiprocessor schedulers, namely time sharing and dynamic space sharing. Since the functionality of the algorithm is orthogonal to the scheduling policy of the operating system, we believe that the algorithm constitutes a useful performance tuning tool for multiprogrammed DSMs.

## ACKNOWLEDGMENTS

The authors would like to thank the journal referees for their insightful comments, which helped us improve the paper considerably.

## REFERENCES

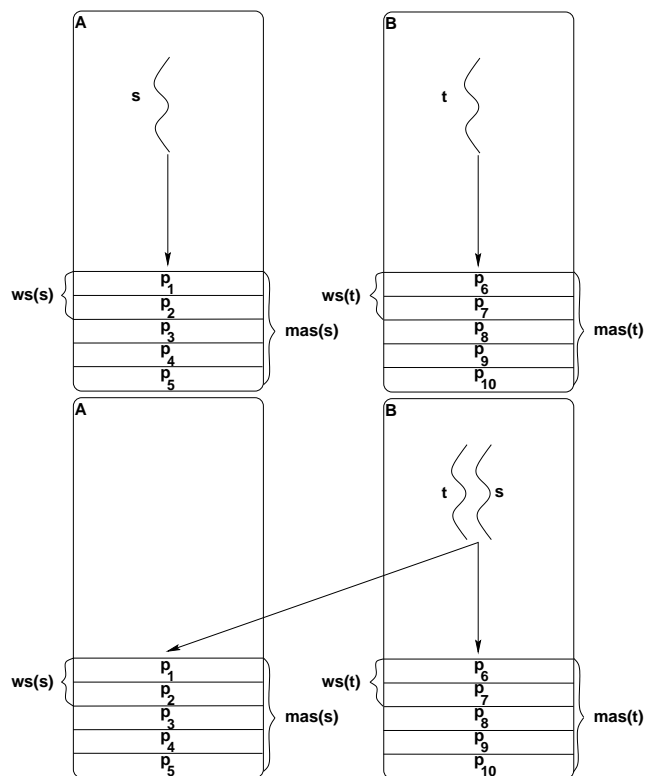
- [1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.
- [2] V. Ashlot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady. SPECmp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proc. of the Second International Workshop on OpenMP Applications and Tools (WOMPAT'01)*, pages 1–10, W. Lafayette, Indiana, July 2001.
- [3] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, December 1995.
- [4] J. Barton and N. Bitar. A Scalable Multidiscipline Multiprocessor Scheduler Framework for IRIX. In *Proc. of the First Workshop on Job Scheduling Strategies for Parallel Processing, in conjunction with IEEE IPSP'95, Lecture Notes in Computer Science vol. 949*, pages 45–69, Santa Barbara, California, April 1995.
- [5] D. Black, A. Gupta, and W. Weber. Competitive Management of Distributed Shared Memory. In *Proc. of the 34th IEEE Computer Society International Conference (COMPCON'89)*, pages 184–191, San Francisco, California, February 1989.
- [6] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but Effective Memory Techniques for NUMA Memory Management. In *Proc. of the 12th ACM Symposium on Operating System Principles (SOSP'89)*, pages 19–31, Litchfield Park, Arizona, December 1989.

- [7] R. Chandra, S. Devine, A. Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 12–24, San Jose, California, October 1994.
- [8] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proc. of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'91)*, pages 120–132, San Diego, California, June 1991.
- [9] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proc. of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, pages 171–181, Orlando, Florida, 1999.
- [10] P. Jackson. Process Scheduling and Memory Placement – Design Notes. SGI Public Document, July 2001.
- [11] D. Jiang and J. P. Singh. Scaling Application Performance on a Cache-Coherent Multiprocessor. In *Proc. of the 26th International Symposium on Computer Architecture (ISCA'99)*, pages 305–316, Atlanta, Georgia, May 1999.
- [12] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.
- [13] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proc. of the 13th ACM Symposium on Operating System Principles (SOSP'91)*, pages 41–55, Pacific Grove, California, October 1991.
- [14] P. Keleher. Tapeworm: High Level Abstractions of Shared Accesses. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 201–214, New Orleans, Louisiana, February 1999.
- [15] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA'97)*, pages 241–251, Denver, Colorado, June 1997.
- [16] Linux Scalability Effort. Linux on NUMA Project Link. [www.sourceforge.net](http://www.sourceforge.net), November 2001.
- [17] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott. Using Simple Page Placement Schemes to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In *Proc. of the 9th IEEE International Parallel Processing Symposium (IPPS'95)*, pages 380–385, Santa Barbara, California, April 1995.
- [18] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.
- [19] National Center for Supercomputing Applications. SGI Origin2000. <http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/Origin2000>, November 2001.
- [20] D. Nikolopoulos. System Software Support for Reducing Memory Latency on ccNUMA Architectures. PhD Thesis, Department of Computer Engineering and Informatics, University of Patras, Greece, December 2000.
- [21] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. A Case for User-Level Dynamic Page Migration. In *Proc. of the 14th ACM International Conference on Supercomputing (ICS'2000)*, pages 119–130, Santa Fe, New Mexico, May 2000.

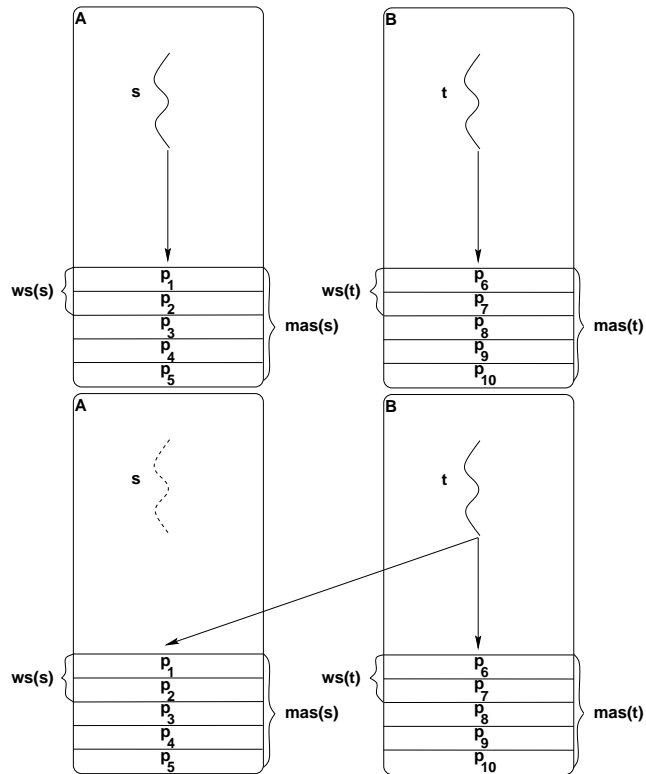
- [22] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. UPMLib: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared-Memory Multiprocessors. In *Proc. of the 5th ACM Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'2000)*, pages 85–99, Rochester, New York, May 2000.
- [23] L. Noordegraaf and R. Van der Pas. Performance Experiences on Sun's WildFire Prototype. In *Proc. of the IEEE/ACM Supercomputing'99: High Performance Networking and Computing Conference (SC'99)*, Portland, Oregon, November 1999.
- [24] A. Patil. Personal communication. Department of Computer Science, University of Houston, November 2001.
- [25] C. Polychronopoulos, N. Bitar, and S. Kleiman. Nano-Threads: A User-Level Threads Architecture. Technical Report 1297, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, March 1993.
- [26] C. Scheurich and M. Dubois. Dynamic Page Migration in Multiprocessors with Distributed Global Memory. *IEEE Transactions on Computers*, 38(8):1154–1163, August 1989.
- [27] Silicon Graphics Inc. IRIX 6.5 Man Pages. <http://techpubs.sgi.com>, January 2000.
- [28] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. Hennessy. Flexible Use of Memory for Migration/Replication in Cache-Coherent DSM Multiprocessors. In *Proc. of the 25th International Symposium on Computer Architecture (ISCA'98)*, pages 342–355, Barcelona, Spain, June 1998.
- [29] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, February 1995.
- [30] A. Tucker. *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*. PhD thesis, Department of Computer Science, Stanford University, November 1993.
- [31] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared Memory Multiprocessors. In *Proc. of the 13th ACM Symposium on Operating System Principles (SOSP'91)*, pages 26–40, Pacific Grove, California, October 1991.
- [32] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 279–289, Cambridge, Massachusetts, October 1996.
- [33] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 181–192, San Jose, California, October 1998.
- [34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95)*, pages 24–37, Santa Margherita Ligure, Italy, June 1995.
- [35] K. Yue and D. Lilja. An Effective Processor Allocation Strategy for Multiprogrammed Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1246–1258, December 1997.

S	UID	PID	SZ:RSS	STIME	TIME	CMD
S	lili	104006	40:30	11:36:30	0:00	/bin/csh /u/ac/lili/.lsbatch/100584
S	lili	104036	52:43	11:36:30	0:00	/bin/sh /u/ac/lili/.lsbatch/1005845
S	mvega	80090	52:42	08:09:36	0:00	/bin/sh -x /u/ac/mvega/.lsbatch/100
S	lili	104154	183:104	11:36:35	0:00	/usr/local/lsf3.2/mnt/sgi6/etc/res
S	lili	104166	52:43	11:36:36	0:00	/bin/sh /u/ac/lili/.lsbatch/1005845
S	lili	104169	40:30	11:36:37	0:00	/bin/csh /u/ac/lili/.lsbatch/100584
S	sbc	104196	183:104	11:36:39	0:00	/usr/local/lsf3.2/mnt/sgi6/etc/res
S	sbc	104223	39:29	11:36:40	0:00	/bin/csh /u/ac/sbc/.lsbatch/1005845
S	mvega	80232	183:104	08:09:35	0:00	/usr/local/lsf3.2/mnt/sgi6/etc/res
R	mvega	80235	7187:2229	- 08:09:42	243:54	/usr/local/apps/chemistry/g98a11/g9
S	mvega	80238	52:43	08:09:35	0:00	/bin/sh /u/ac/mvega/.lsbatch/100583
S	sbc	104251	36:26	11:36:42	0:00	abaqus -f ./sqaan_hol.com
R	sbc	104264	1362:817	11:36:47	37:14	/usr/local/apps/fe/abaqus5.8-19/bin
S	mvega	80274	3896:185	08:09:42	0:00	g98 input T121113.out
S	lp	707	126:71	20:54:08	0:00	/usr/lib/lpsched
S	ncagle	89115	39:29	09:44:41	0:00	-csh
R	lili	105901	5229:3834	- 12:08:02	6:06	regridder
R	lili	105979	5244:3866	- 12:04:23	9:44	regridder
S	cpolychr	106478	39:29	12:14:08	0:00	/bin/csh /u/ac/cpolychr/.lsbatch/10
S	cpolychr	106504	183:104	12:13:56	0:00	/usr/local/lsf3.2/mnt/sgi6/etc/res
S	cpolychr	106512	53:44	12:14:07	0:00	/bin/sh /u/ac/cpolychr/.lsbatch/100
R	cpolychr	106562	123:74	- 12:14:08	0:00	ps -elfa
S	mvega	82597	7187:2229	08:39:04	0:10	/usr/local/apps/chemistry/g98a11/g9
S	sbc	103863	52:43	11:36:40	0:00	/bin/sh /u/ac/sbc/.lsbatch/10058456
S	lili	103959	183:104	11:36:28	0:00	/usr/local/lsf3.2/mnt/sgi6/etc/res

**FIG. 1** Snapshot of jobs executing on a 16-processor time-shared partition of the NCSA Origin2000.



**FIG. 2** A thread migration on a DSM multiprocessor.  $A$  and  $B$  are nodes,  $p_i$  are pages in the resident set of the program,  $s$  and  $t$  are threads of the program,  $ws(\dots)$  denotes the working set of a thread and  $mas(\dots)$  denotes the memory affinity set of a thread. Note that  $ws(\dots) \subseteq mas(\dots)$ . In this case,  $s$  accesses the pages in  $ws(s)$  and  $mas(s)$  remotely after the migration.



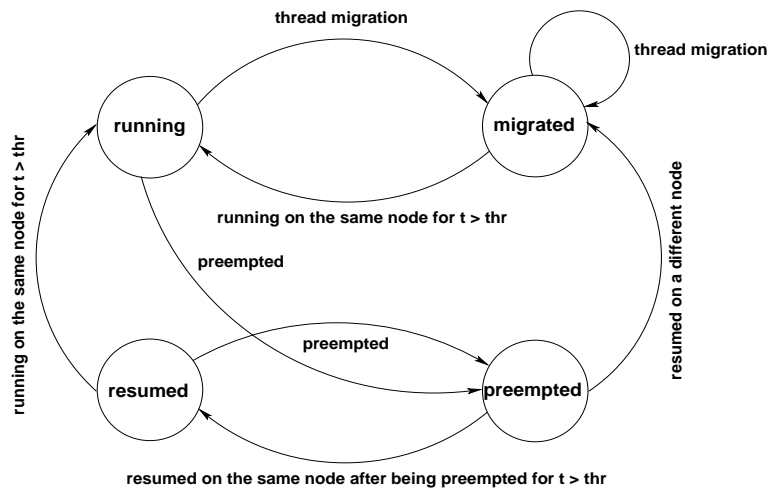
**FIG. 3** A thread preemption on a DSM multiprocessor. In this case,  $s$  is preempted and the computation performed by  $s$  is taken up by  $t$  with the aid of the runtime system. As a consequence,  $t$  accesses  $ws(s)$  and  $mas(s)$  remotely.

```

// T is a table with one tuple per thread i, i = 1 .. MAX_THREADS
// The tuple Ti = (pid, state, prev_node, curr_node, timestamp)
// describes the scheduling state of thread i.
// Pid is the identifier assigned to the thread by the operating system;
// state is the scheduling state of the thread, which can take the values
// preempted, running, migrated or resumed; curr_node is the node on which the
// thread is executed; prev_node is the previous node on which the thread was
// executing before migrating to curr_node; timestamp is the wall clock time
// at which the last snapshot of the current scheduling state of the thread
// was taken by the runtime system.
for (i = 1; i < MAX_THREADS(); i++) {
    if (i is running on a CPU on node n) {
        case (state) { // check the previous state
            running:
                if (curr_node != n) { // this is a migration
                    state = migrated;
                    prev_node = curr_node;
                    curr_node = n;
                    timestamp = clock_gettime();}
            migrated:
                if (curr_node != n) { // this is a re-migration
                    prev_node = curr_node;
                    curr_node = n;
                    timestamp = clock_gettime();}
            preempted:
                if (curr_node != n) { // this is a migration following a preemption
                    state = migrated;
                    prev_node = curr_node;
                    curr_node = n;
                    timestamp = clock_gettime();}
                else {
                    state = resumed; // the thread is resumed on the same node
                    timestamp = clock_gettime();}
            resumed:
                if (curr_node != n) { // this is a migration of a recently resumed thread
                    state = migrated;
                    prev_node = curr_node;
                    curr_node = n;
                    timestamp = clock_gettime();}
        }
    }
    else { // thread i is preempted
        case (state) {
            running : migrated : preempted : resumed :
            state = preempted;
            timestamp = clock_gettime(); }
    }
}

```

FIG. 4 Maintenance of thread scheduling information.



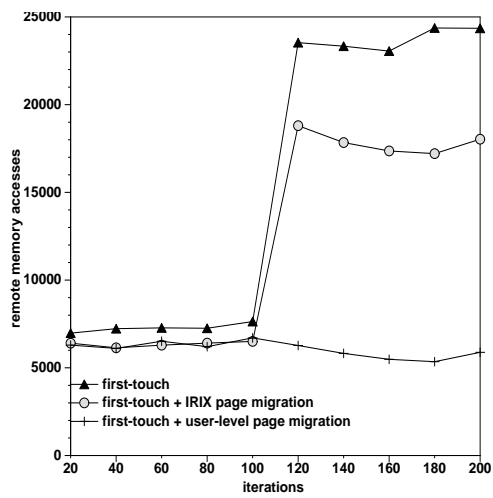
**FIG. 5** State diagram maintained by the algorithm.

```

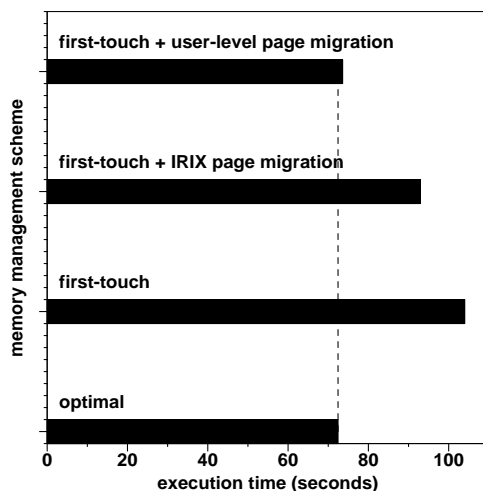
//  $P_k$ ,  $k = 1 \dots resident\_set\_size$  are the pages
// in the resident set of the program,  $nacc_k(i, it)$  is the
// number of accesses from node  $k$  to page  $i$  during
// iteration  $it$ ,  $h$  is the home node of the page and  $currit$  and
//  $previt$  the current and previous iterations of the program respectively.
//  $t_{thr}$  is the time needed to migrate the entire memory affinity set of a thread.
for ( $i = 1$ ;  $i \leq resident\_set\_size$ ;  $i++$ ) {
  read_access_counters( $P_i$ );
  if  $nacc_h(i, currit) < nacc_h(i, previt)$  {
    if  $\exists j, j \neq h, nacc_j(i, currit) > nacc_j(i, previt)$  {
      find  $k, nacc_k(i, currit) > nacc_k(i, previt) \wedge$ 
       $nacc_k(i, currit) > nacc_j(i, currit) \forall j \neq k$ 
      if ( $\exists T, T.status == migrated \ \&\& \ clock_gettime()-T.timestamp > t_{thr}$ 
       $\ \&\& \ T.prev\_node == h \ \&\& \ T.curr\_node == k$ ) {
        migrate the page to  $k$ ; } // Thread migration
      if ( $\exists T, T.status == preempted \ \&\& \ clock_gettime()-T.timestamp > t_{thr}$ 
       $\ \&\& \ T.curr\_node == h$ ) {
        migrate the page to  $k$ ; } // Preemption with implicit migration of computation to  $k$ 
      if ( $\exists T, T.status == resumed \ \&\& \ clock_gettime()-T.timestamp > t_{thr}$ 
       $\ \&\& \ T.curr\_node == k$ ) {
        migrate the page to  $k$ ; } } // Resumption after preemption
    }
  }
for ( $i = 1$ ,  $i \leq MAX\_THREADS$ ;  $i++$ ) {
  if ( $T.status == migrated || T.status == resumed$ )
   $T.status = running$ ; }

```

**FIG. 6** Page migration criterion.



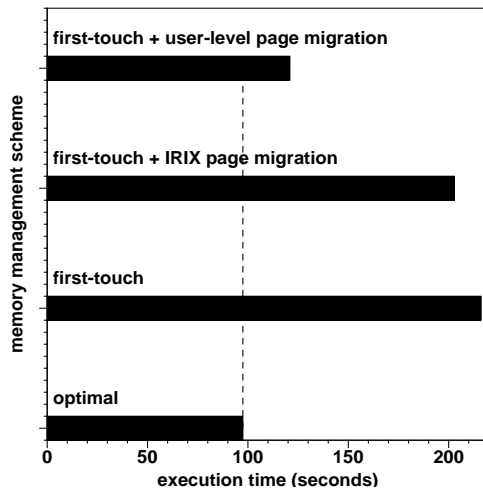
**FIG. 7** Rate of remote memory accesses to *rhs*, during the execution of NAS BT in the first experiment.



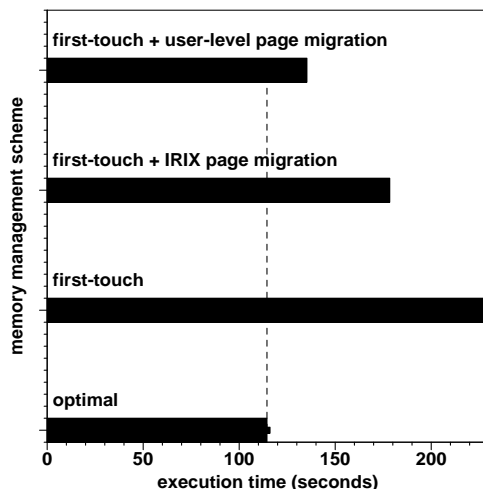
**FIG. 8** Execution time of NAS BT in the first experiment. The optimal execution time is computed as  $T_{opt} = \frac{t_{32}}{2} + \frac{t_{16}}{2}$ .

TABLE 1  
Resident set size and page migrations during the execution of BT in the first experiment.

Resident set (in pages)	3056
Page migrations-IRIX engine	747
Page migrations-user-level engine	1449



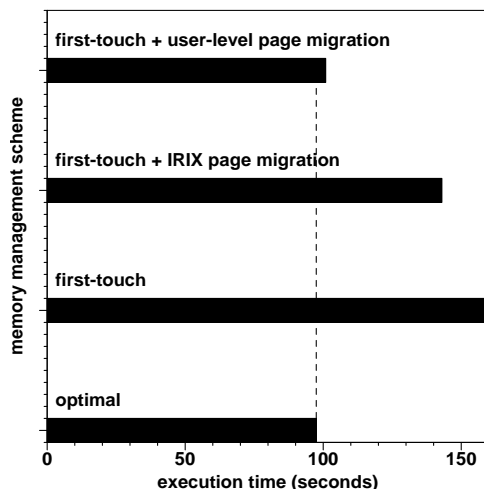
**FIG. 9** Average turnaround time of NAS BT in the multiprogrammed workload executed with time sharing. The optimal execution time is computed as  $T_{opt} = 2 \cdot t_{32}$ .



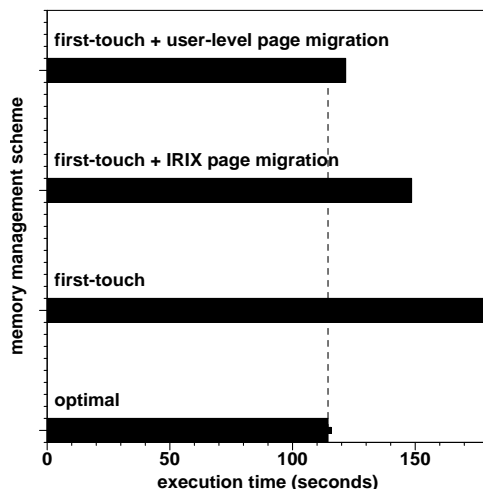
**FIG. 10** Average turnaround time of NAS SP in the multiprogrammed workload executed with time sharing. The optimal execution time is computed as  $T_{opt} = 2 \cdot t_{32}$ .

TABLE 2  
 Scheduler statistics collected during the execution of the workloads with time sharing.

<b>BT</b>		<b>SP</b>	
Preemptions	2134	Preemptions	2329
Migrations	636	Migrations	801
Preemptions with duration $> t_{thr}$	13	Preemptions with duration $> t_{thr}$	17
Migrations with duration $> t_{thr}$	168	Migrations with duration $> t_{thr}$	192



**FIG. 11** Average turnaround time of NAS BT in the multiprogrammed workload executed with dynamic space sharing. The optimal execution time is computed as  $T_{opt} = 2 \cdot t_{32}$ .

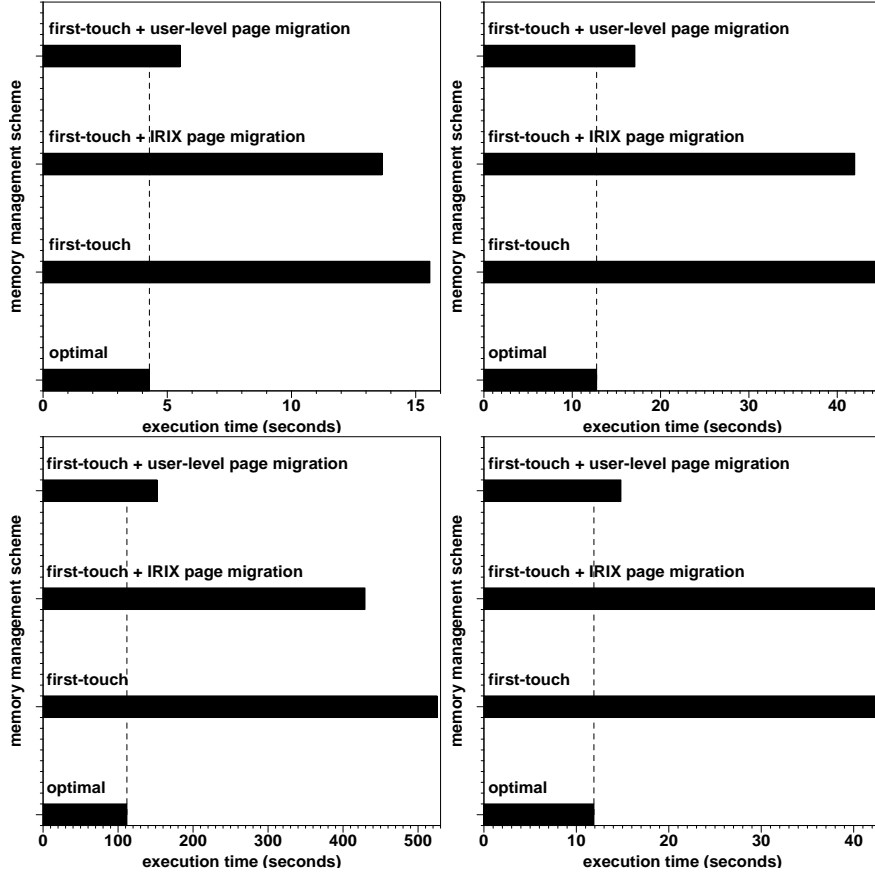


**FIG. 12** Average turnaround time of NAS SP in the multiprogrammed workload executed with dynamic space sharing. The optimal execution time is computed as  $T_{opt} = 2 \cdot t_{32}$ .

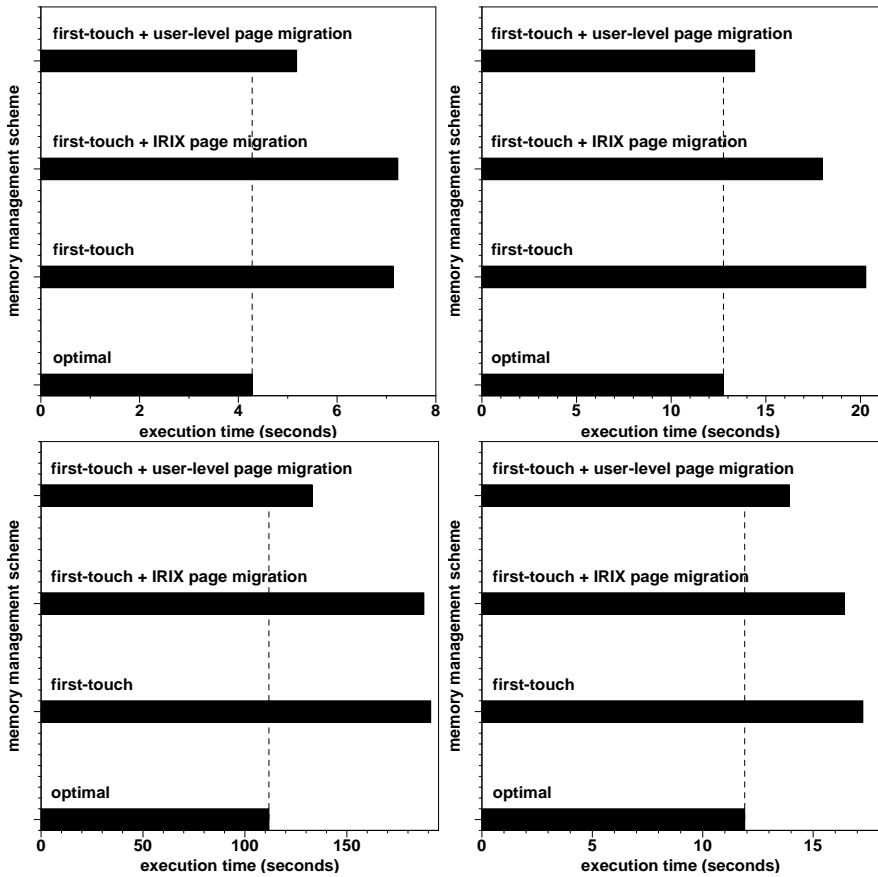
TABLE 3  
 Scheduler statistics collected during the execution of the workloads with dynamic  
 space sharing.

<b>BT</b>		<b>SP</b>	
Preemptions	336	Preemptions	461
Migrations	517	Migrations	697
Preemptions with duration $> t_{thr}$	258	Preemptions with duration $> t_{thr}$	277
Migrations with duration $> t_{thr}$	393	Migrations with duration $> t_{thr}$	404

APPENDIX A: RESULTS OF EXPERIMENTS WITH THE NAS  
BENCHMARKS OMITTED FROM SECTION 6



**FIG. 13** Average turnaround time of NAS CG (upper left), FT (upper right), LU (lower left) and MG (lower right) in the multiprogrammed workload executed with time sharing. The optimal execution time is computed as  $T_{opt} = 2 \cdot t_{32}$ .



**FIG. 14** Average turnaround time of NAS CG (upper left), FT (upper right), LU (lower left) and MG (lower right) in the multiprogrammed workload executed with dynamic space sharing. The optimal execution time is computed as  $T_{opt} = 2 \cdot t_{32}$ .

---

<sup>3</sup>We use a small allocation of 10,000 CPU hours of dedicated CPU time and unlimited time-shared CPU time. These allocations are typically given out for small-scale academic projects at the University of Illinois.

<sup>4</sup>The term node is used to denote the basic building block of a DSM multiprocessor. Typically, a node is composed of a small number of processors, memory, and a communication assist.

<sup>5</sup>The terms virtual processor, kernel thread and execution vehicle are synonyms denoting the entities provided by the operating system for multithreading a shared address space.

<sup>6</sup>We assume that page migration can be overlapped with the execution of useful computation.

<sup>7</sup>Neighboring nodes are defined according by the topology of the interconnection network.

<sup>8</sup>We used the OpenMP `OMP_SET_NUM_THREADS()` runtime intrinsic to adjust manually the number of threads in the benchmarks.

<sup>9</sup>In fact, the rate drops slightly because some remote accesses are converted to local due to the redistribution of the parallel computation.