User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors

Dimitrios S. Nikolopoulos[†], Theodore S. Papatheodorou[†]
Constantine D. Polychronopoulos[‡], Jesús Labarta[§] and Eduard Ayguadé[§]

[†] Department of Computer Engineering and Informatics
University of Patras, Greece
{dsn,tsp}@hpclab.ceid.upatras.gr

[‡] Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
cdp@csrd.uiuc.edu

[§] Department of Computer Architecture
Technical University of Catalonia, Spain
{jesus,eduard}@ac.upc.es

Abstract

This paper presents algorithms for improving the performance of parallel programs on multiprogrammed shared-memory NUMA multiprocessors, via the use of user-level dynamic page migration. The idea that drives the algorithms is that a page migration engine can perform accurate and timely page migrations in a multiprogrammed system if it can correlate page reference information with scheduling information obtained from the operating system. The necessary page migrations can be performed as a response to scheduling events that break the implicit association between threads and their memory affinity sets. We present two algorithms that use feedback from the kernel scheduler to aggressively migrate pages upon thread migrations. The first algorithm exploits the iterative nature of parallel programs, while the second targets generic codes without making assumptions on their structure. Performance evaluation on an SGI Origin2000 shows that our page migration algorithms provide substantial improvements in throughput of up to 264% compared to the native IRIX 6.5.5 page placement and migration schemes.

1. Introduction

The commercial success of scalable cache-coherent shared memory multiprocessors and their widespread use in industrial and academic environments stimulate intensive efforts for providing system software which enables these systems to achieve high performance without sacrificing programmability. One of the exceptional characteristics of modern shared memory multiprocessors is that they are often used as multiprogrammed compute servers, in which users not only submit parallel CPU-intensive programs, but also develop and debug programs, execute long sequential simulations and commercial applications. The workloads presented to modern multiprocessors pose an ever increasing demand for throughput in parallel with the traditional requirements of speedup and scalability. It is therefore crucial to provide mechanisms that enable parallel programs to have robust performance in multiprogrammed environments.

The most prominent problem of state-of-the-art scalable shared memory systems is the non-uniformity of memory access latency (NUMA). On cache-coherent systems with distributed shared memory, each virtual memory page is initially mapped on a single node in the system and pages which are actively shared by processors in multiple nodes incur remote accesses upon secondary cache misses. Remote memory accesses cost three to eight times as much as local memory accesses on state-of-the-art systems [2, 5].

The performance implications of multiprogramming on NUMA multiprocessors make the problem of remote memory access latency harder to deal with, because the operating system may break the association between threads and pages at runtime in an unpredictable manner. When parallel programs execute on multiprogrammed systems, their threads may be arbitrarily preempted and migrated by the operating system, in order to sustain high system throughput and utilization. Thread preemptions and migrations incur the cost of cache reloads and compromise the memory performance of parallel programs. The problem on NUMA systems is more pronounced because if a preempted thread is migrated between different nodes in the system, the working set of the thread must be reloaded from a remote memory module, thus incurring a burst of expensive remote memory accesses. Furthermore, if the pages that the thread accesses more frequently reside on a node other than the node to which the thread migrates, most L2 cache misses incurred by the migrated thread will be satisfied by remote memory.

The cost of cache reloads and remote memory accesses due to thread migrations on a NUMA system can be reduced if the operating system forwards the pages which are accessed mostly by a migrated thread to the new home node of the thread. Dynamic page migration [14] is a technique which has the potential to alleviate this problem. A NUMA system with dynamic page migration manages the distributed shared memory in a competitive manner, so that each virtual memory page is effectively moved to the node that references the page more frequently. The purpose of the page movement is to minimize the maximum latency due to remote memory accesses by any node to this page. The early results from simulations of dynamic page migration engines on NUMA multiprocessors were encouraging [12, 14], however, the actual implementations of dynamic page migration in two state-of-the-art systems, the SGI Origin2000 and the Sun Wildfire, have not exhibited the expected performance gains [3, 9]. In particular, we are not aware of studies that utilized dynamic page migration to improve throughput on an actual multiprogrammed NUMA system. This fact motivated us to investigate in more depth mechanisms and algorithms for effective dynamic page migration.

In this paper, we pursue the idea of executing timely page migrations in a multiprogrammed NUMA system, by correlating reference counting information with scheduling information provided by the operating system. Our technique intercepts preemptions and thread migrations at user-level and uses these events as triggers for activating aggressive page migration algorithms that associate reference counting information with the nodes to or from which threads migrate. We present two algorithms for performing accurate page migrations upon thread migrations without waiting for the page reference counters to accumulate history that would justify page migration according to a competitive criterion. The first algorithm targets iterative parallel programs, in the case of which a page migration mechanism can make some safe assumptions on the expected memory behavior of a program and detect anomalies due to thread migrations. The second algorithm targets any kind of parallel computation, without making any assumptions on the future memory behavior of the program.

We have implemented our mechanisms in UPMlib, a runtime system that provides transparent facilities for dynamic page migration and memory performance monitoring to OpenMP programs [8]. Our current prototype is implemented on the SGI Origin2000, on top of IRIX 6.5.5. We experimented with multiprogrammed workloads composed of NAS benchmarks, using the dynamic space and time-sharing scheduler of IRIX. Our experimental results show that our page migration algorithms achieve sizeable throughput improvements of up to 264% compared to the IRIX page placement and migration schemes. The improvements are observed both with plain dynamic space-sharing and with space and time-sharing by the IRIX scheduler.

The rest of this paper is organized as follows. Section 2 establishes the problem framework. Section 3 presents our page migration algorithms. Section 4 gives some implementation details. Section 5 provides experimental results and Section 6 concludes the paper.

2. Problem Statement

This section establishes the problem framework for dynamic page migration in multiprogrammed shared-memory NUMA multiprocessors. Section 2.1 discusses briefly the performance implications of multiprogramming and job scheduling on the performance of parallel applications on NUMA multiprocessors. Section 2.2 discusses the role of dynamic page migration on reducing memory latency in multiprogrammed execution environments and overviews some related work.

2.1. Job Scheduling and Memory Performance on Multiprogrammed NUMA Multiprocessors

Sophisticated scheduling schemes for multiprogrammed shared-memory multiprocessors such as dynamic space sharing [13], require frequent changes of the scheduling state of threads, namely preemptions and migrations. These scheduling actions trigger cache reloads, which become a major source of performance degradation on a shared-memory multiprocessor. Although the performance implications of multiprogramming are to some extent similar for all types of cache-coherent shared-memory multiprocessors, it has been shown that the memory performance of parallel programs is more sensitive to multiprogramming and job scheduling strategies on scalable NUMA systems [1]. The first apparent reason is that the cost of a cache reload on a NUMA multiprocessor depends on the distance between the pages that contain the data to be reloaded and the memory modules of the system in which these pages reside. Reloading the cache from remote memory modules is significantly more expensive than reloading the cache from local memory modules. A second reason is that on a NUMA system, thread migrations break an implicit association between threads and pages for which threads have memory affinity. On a cache-coherent NUMA system, each virtual memory page is allocated on a single node which serves as the home node of the page. It is desirable to align threads and pages so that each thread is collocated with the pages that the thread accesses more frequently. We call this set of pages the *memory affinity set* of a thread. Maintaining thread-to-memory affinity is important in order to avoid forcing threads accessing remote memory upon L2 cache misses.

In practical situations, the memory affinity set of each thread is allocated on the node on which the thread is executed for the first-time, via a first-touch page placement strategy [6], or in an application-specific manner which is hardwired in the program. In both circumstances, the memory allocation scheme assumes that each thread will be bound to a specific node of the system throughout the lifetime of the program. If a thread is migrated to a node other than the node on which its memory affinity set is allocated, the thread-to-memory affinity relationship is broken and memory latency due to remote memory accesses is exacerbated. The pages in the previously established memory affinity set of the migrated thread become *orphaned*, in the sense that they no longer belong to the memory affinity set of any thread.

2.2. The Role of Dynamic Page Migration

Dynamic page migration based on information from hardware reference counters is a technique that can potentially alleviate the problem described in Section 2.1 [1, 6, 14]. The idea behind dynamic page migration is to collect per-node reference information for each page in memory and migrate a page to a remote node if: (1) the reference counters indicate that the remote node accesses the page significantly more frequently compared to the home node; (2) the benefits from migrating the page are likely to outweigh the cost of copying the page and maintaining memory coherence thereafter.

Previously proposed page migration mechanisms [12, 14] are solely based on competitive algorithms, which migrate a page if the difference between the number of local references and the number of remote references from at least one node exceeds a predefined threshold. Applying a competitive algorithm for orphaned pages in a multiprogrammed system might be a poor decision, since the algorithm must needlessly wait until the new home node of a migrated thread issues a sufficiently large amount of references to meet the competitive criterion. The timeliness of page migrations can be very poor in this case, as it depends on the past reference history of the page i.e. the number of references from the previous home node of the migrated thread. The past reference history is obsolete, since it is not correlated with the actual status of the computation which is reflected by the new mapping of threads to processors. In the worst-case, the orphaned pages might not migrate at all if the reference counters have accumulated so much obsolete history that the number of references from the new home node of the thread does not exceed the number of past references from the old home node of the thread.

The problem that has to be addressed by dynamic page migration in the aforementioned case, is how to migrate orphaned pages as soon as possible, given that page migrations are triggered with implicit information from the reference counters. This can be accomplished if the page migration engine captures the scheduling events that require page migrations and associates the information from the reference counters with these events, in order to identify orphaned pages early without being biased from obsolete page reference history. Based on this idea, in Section 3 we propose two algorithms for performing effective page migrations as a response to thread migrations on a multiprogrammed NUMA system.

3. Page Migration Algorithms

In this section we propose two dynamic page migration algorithms, designed to perform accurate and timely page migrations upon migrations of threads in multiprogrammed NUMA shared-memory multiprocessors. The algorithms associate the reference information with the nodes to or from which threads migrate and the observed reference rate from these nodes to the resident set of pages. In this way, the algorithms identify which pages belong to the memory affinity set of migrated threads and move them earlier, according to a non-competitive criterion. Both algorithms assume compiler support for identifying *hot* memory areas, that is, memory areas which are likely to concentrate excessive remote accesses and have several candidate pages for migration. Such a scheme is described in [7].

The first algorithm targets iterative parallel programs, in which the same parallel computation is repeated for a number of iterations. These programs represent the vast majority of parallel codes. Iterative parallel programs have inherent properties that enable a page migration algorithm to make some safe assumptions on the expected memory reference pattern of the program. The second algorithm is more general and does not make any assumptions on the expected memory reference pattern. Both algorithms assume that the runtime system has a mechanism to intercept the changes of the effective processor set[1] on which a parallel program runs. Changes of the effective processor set imply migra-

[1]We define as *effective processor set*, the set of processors that execute the running threads of a parallel program at any point of execution.

tion of computation and they are interpreted in the runtime system as triggers for switching the page migration policy.

3.1. Predictive Algorithm

The predictive algorithm works with iterative parallel programs, in which a page migration mechanism can be based on the assumption that the page reference pattern of one iteration will be repeated throughout the execution of the program. Under this assumption, the page migration mechanism can take snapshots of the reference counters at the end of the outer iterations of the program and use these snapshots to achieve an optimal page placement with respect to the relationship between threads and memory affinity sets. Optimal placement is attained when each page is moved to a node so that the maximum latency due to remote memory accesses by any node to this page is minimized.

In iterative codes, the page migration algorithm can easily detect anomalies in the reference pattern. If the page reference rate from the home node of a given page in one iteration appears to increase at a slower rate with respect to previous iterations, while the reference rate from some other node appears to increase at a faster rate with respect to previous iterations, the page migration algorithm can speculate that this behavior is a result of a thread migration between the two nodes. The algorithm can subsequently verify this speculation with information provided by the operating system. If the speculation is verified, the algorithm can infer that the page belongs to the memory affinity set of the migrated thread and migrate the page to the new home node of the thread, regardless of the accumulated reference history of the page.

The predictive algorithm uses by default a competitive page migration criterion to accurately migrate pages at the end of outer iterations of the parallel computation, as described in [7]. The algorithm polls the effective processor set at the beginning and the end of each parallel construct. The runtime system checks if the number of processors that execute parallel constructs changes at any point of execution and marks the processors to/from which threads migrate. Both events are recorded by the runtime system to trigger the predictive algorithm for migrating pages. In order to avoid undesirable interference from temporary thread migrations from the operating system, the algorithm checks if the thread is migrated and stays on the same node for a sufficiently long amount of time, typically at least for a couple of seconds. This check is done by time-stamping the execution points at which the page migration library polls the effective processor set. Since the cost of migrating pages is substantially high (in the order of 1ms. on state-of-the-art systems), it would be desirable to start migrating pages close to a thread, only if the thread is likely to spend a significant amount of time on the same node, in order to compensate for the overhead.

The predictive algorithm identifies a page as potentially orphaned if it detects that across two iterations of the parallel computation the number of references from the home node of the page increases at a slower rate with respect to previous iterations. The runtime system checks the effective processor set of the program to identify if any thread that was running on the home node of the page migrated to a node n, and looks also if the accesses from node n have increased at a faster rate with respect to previous iterations. If the check is verified the orphaned page is migrated to node n. The algorithm switches back to the default competitive criterion, if it detects that across two iterations of the computation there are no pages that satisfy the predictive criterion in the memory areas scanned by the page migration engine.

3.2. Aging Algorithm

We propose a second algorithm, called the aging algorithm, which does not make any assumptions on the structure of the parallel computation or the status of the computation at the time when a thread migration occurs. The aging algorithm uses a sampling-based mechanism for dynamic page migration. Assuming that there are no apparent points of execution at which the page migration engine can be activated to perform accurate page movements, as in the case of iterative programs, a sampling-based mechanism checks periodically hot memory areas and migrates the pages with excessive remote references, according to a competitive criterion.

The aging algorithm intercepts changes of the effective processor set of the program similarly to the predictive algorithm and activates counter aging when it detects a change in the processor set. When the algorithm scans the pages in a memory area and aging is activated, the algorithm ages the reference counters for the nodes from which threads migrate, if there are no other threads of the program running on these nodes. The latter condition is used to deal with cases in which a node has more than one processor and a thread migration from this node does not imply that the node will not access the page in the future. After aging the counters the algorithm uses the competitive criterion to migrate pages.

Aging a counter means simply resetting the counter to 0, or equivalently, record the contents of the counter and subtract the recorded value from the actual contents of the counter whenever the counter is accessed again in the future. The intuition behind aging is that when a thread migration occurs the reference counters for the node from which the thread migrated contain obsolete reference history, which should not bias the decisions for migrating pages. Aging does not imply immediate page migrations as the new home node of the migrated thread must still issue a sizeable number of references to the page in order to meet the competitive criterion. However, the reset of the counter of the previous home node increases the chances for migrating the page away from this node. If the access pattern has temporal locality and the migrated thread is likely to access the orphaned pages more frequently than the other threads in the near future, orphaned pages will eventually migrate to the right place.

4. Implementation Details

This section provides implementation details for our dynamic page migration algorithms. Both algorithms are implemented at user-level in UPMlib [8], a runtime system which provides transparent services for dynamic page migration and memory performance monitoring to OpenMP programs.

4.1. Intercepting Thread Migrations at User-Level

Our page migration algorithms require that the operating system provides information to the runtime system on the instantaneous mapping of threads to physical processors, in order to detect thread migrations upon their occurrence. We use shared memory and an asynchronous communication model between the programs and the operating system, based on polling for informing programs of thread migrations [10]. In this case, the kernel keeps the scheduling information updated in a pinned region of the virtual address space of the program and the program polls this information at user-level. The algorithms poll the thread mapping information before and after the execution of parallel constructs. This choice is reasonable, in the sense that it considers the macroscopic behavior of the program and optimizes page placement with respect to this behavior, rather than being biased by temporary effects, such as momentary thread migrations for one time quantum. A second reason that motivates this polling frequency is the high cost of page migrations which must be carefully amortized.

4.2. Implementation Details in IRIX

UPMlib is implemented using the IRIX 6.5.5 memory management control interface (`mmci`) [11]. The `mmci` virtualizes the physical memory of the system and allows the user to customize its page placement schemes by associating regions of the virtual address space with Memory Locality Domains (MLDs). MLDs can be associated in turn with physical nodes in the system, to enable NUMA-aware placement and migration of pages. Reference counting information used for deciding what pages should migrate is collected directly from the SGI Origin2000 hardware and extended-software reference counters via the `/proc` interface [11].

UPMlib polls the `t_cpu` field of the thread-private data area (`prda`) at the boundaries of parallel constructs, using the `schedctl(SETHINTS)` call. Each thread polls its own `prda` and records the information in a table of thread-to-CPU mappings in shared memory. In this way, UPMlib obtains information on the actual CPU on which each thread executed during the last time quantum. The table is scanned after each parallel construct in order to mark and record thread migrations. Note that this implementation detects migrations of threads with a hysteresis of at most one parallel construct.

In the implementation of the algorithms we use the master thread of each program to execute UPMlib code. In order to avoid having the working set of UPMlib erase the cache footprint of the master thread which participates in the execution of parallel code, we used stripmining to reduce the working set size of UPMlib to approximately half the size of the primary data cache. The working set of UPMlib is basically comprised of buffers into which the reference counters are read. Furthermore, we multiplexed the reference counters in one small 8-Kbyte buffer, at the cost of needing more accesses to the `/proc` filesystem for reading all hardware counters into the buffer.

5. Experimental Results

We experimented on a 32-node (64-processor) SGI Origin2000. Each node had two MIPS R10000 processors clocked at 250 MHz and 4 Mbytes of L2 cache per processor; the system had a total of 8 Gbytes of main memory equidistributed across the nodes. The operating system on which we performed the experiments was IRIX 6.5.5. We used two application benchmarks from the NAS benchmark suite, namely BT and SP, both implemented with OpenMP and customized to the Origin2000 architecture by their providers [4]. In particular, the programs were tuned to exploit the first-touch page placement strategy of IRIX on Origin2000 systems. The benchmarks are iterative in nature and this was the case for all OpenMP codes that were available to us. Although experimenting with non-iterative codes would be desirable to evaluate the aging algorithm, iterative benchmarks can also serve this purpose, by comparing the aging algorithm to the predictive algorithm. The latter is expected to perform better with iterative codes.

5.1. Impact of Thread Migrations

In the first experiment we executed BT and SP individually on a dedicated system, after modifying the code to artificially reduce the number of threads used for executing parallel loops after the first half of the iterations in each benchmark. The benchmarks were executing the first half of the iterations with 32 threads bound to 32 processors that occupied exactly 16 nodes of the system. The last half of the iterations were executed on 16 processors bound to 8 nodes of the system, which were a subset of the 16 nodes on which the benchmark started to run. This synthetic experiment forced the benchmarks to migrate the computation performed by one half of the threads to the other half. The artificial thread migrations implicitly required the migrations of the pages in half of the memory affinity sets of the programs.

The left chart in Figure 1 shows the histogram of remote references to one of the most heavily referenced shared arrays in the NAS BT benchmark (array `rhs`), as a function of the iteration number in our synthetic experiment. The numbers are averages of 10 independent experiments. The variance was less than 2%. The purpose of the histogram is to demonstrate the effect of thread migrations on memory latency. Each triplet of bars in the histogram corresponds to the accumulated number of remote references from all threads to array `rhs`, when the benchmark is executed with first-touch page placement and without page migration (`ft-IRIX`), with first-touch and the IRIX page migration engine enabled (`ft-IRIXmig`) and with first-touch and our predictive page migration algorithm (`ft-predmig`).

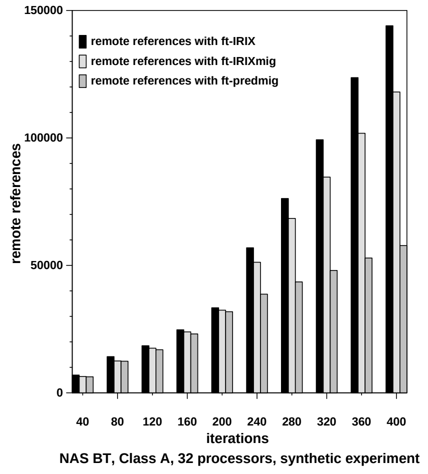
If the experiments were executed on 32 dedicated processors without preemptions, the expected behavior would be a linear increase of the number of remote references with respect to the number of iterations at a constant rate. Indeed, this is the observed behavior in the benchmark in the first 200 iterations, during which the number of processors that execute the benchmark remains unchanged. Simple linear regression shows that without page migrations the number of remote references increases at the rate of approximately 160 per iteration. The rate is somewhat lower if a page migration engine is used, because the first-touch strategy sub-optimally places several pages in the benchmark and these pages are competitively migrated in the course of execution, thus reducing the number of remote references. When half of the threads executing the benchmark are preempted and no page migration is used, the rate of the increase of remote references raises sharply to approximately 554 per iteration, 3.5 times as much as when the benchmark was executed on 32 processors. This is the consequence of having the pages in the memory affinity sets of half of the threads being accessed remotely.

The impact of thread migrations on execution time is shown in the right chart in Figure 1. The leftmost bar in the chart shows the theoretical optimal execution time of BT in the experiment, under the assumption that when the number of threads is reduced from 32 to 16, the program executes exactly twice as slowly as when executed on 32 processors. The optimal execution time was computed by measuring the execution time of the first half of the iterations and projecting the execution time of the rest of the iterations on 16 processors. The next five bars correspond to the execution times with the IRIX first-touch page placement strategy (`ft-IRIX`), with first-touch and the IRIX page migration engine enabled (`ft-IRIXmig`), with first-touch and our predictive page migration algorithm (`ft-predmig`) and first-touch and our aging page migration algorithm (`ft-agingmig`). In the case of the aging algorithm, we provide results for two different sampling frequencies used in the page migration engine, a frequency of one hot memory area (i.e. one read/write shared array in the program) per 3 seconds (`ft-agingmig/3`) and a frequency of one hot memory area per 5 seconds (`ft-agingmig/5`).

The right chart in Figure 1 shows that the burst of remote references due to the change of the number of running threads results in a slowdown of 1.44 for BT. Using page migration in the IRIX kernel reduces the slowdown factor to 1.28. The left chart shows that the remote accesses start increasing at the rate of approximately 400 per iteration, after the 200th iteration. Although page migration in the IRIX kernel provides a sizeable reduction of remote references, the observed behavior is still far from the desirable one. The rightmost bars in the left chart in Figure 1 show that our predictive page migration mechanism reduces the rate of remote references from 160 to 119 per iteration, thus exhibiting the desirable behavior. The execution time with the predictive page migration mechanism is only 2.4% higher than the theoretical optimal execution time. The aging page migration algorithm achieves performance close to the predictive algorithm, within 10% with a sampling frequency of one area per 3 seconds and 17% with a sampling frequency of one area per 5 seconds. Overall, considering the fact that the aging algorithm does not take into account the structure of the parallel program and its performance is within 17% off the performance of the more accurate predictive algorithm, we conclude that the aging algorithm constitutes a quite attractive alternative.

5.2. Impact of Dynamic Space-Sharing

In the second set of experiments, our goal was to evaluate the page migration algorithms in a dynamically space-shared environment, in which the processors of the system are distributed among the running programs. For this purpose, we executed workloads consisting of two copies of BT or two copies of SP, plus a background load of a sequential I/O-intensive C program. Each copy of the benchmarks requested 32 processors and the workload was submitted for execution on a dedicated 32-processor partition of the system. Initially we executed the workloads after setting the `OMP_SET_DYNAMIC` flag in the benchmarks, to acti-



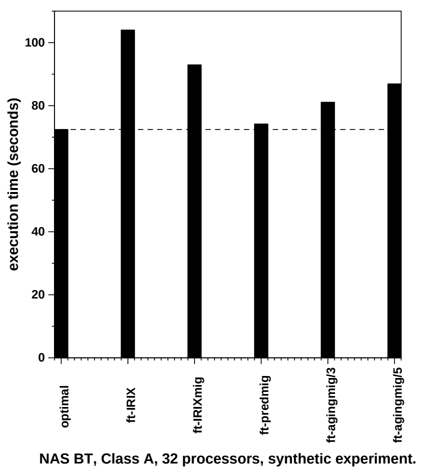


Figure 1. Results from the synthetic experiment.

vate dynamic adjustment of the number of threads that execute parallel constructs by the IRIX MP library. With this modification, we would expect the native IRIX scheduler to somehow partition the 32 processors among the two programs, according to the space-sharing scheduling scheme followed by the IRIX kernel. However, tracing of the execution of the experiments revealed that the IRIX scheduler allocated 32 threads to each program at the beginning of execution and then performed an unbalanced distribution of processors. Therefore, we modified the benchmarks by hand to adaptively self-tune their number of threads according to the total load presented to the system in the experiments. We used a page of virtual memory shared between the programs in the workload and each program updated a global system load index within this page. The programs polled the load index before entering a parallel construct and adjusted the number of threads to implement an equipartitioning processor allocation strategy, using the IRIX `mp_suggested_numthreads()` call [13]. This modification affected only the number of threads used in parallel constructs by the benchmarks. The actual processor on which each thread was scheduled was still controlled by the IRIX scheduler.

Figure 2 shows the average execution time of the benchmarks in the workloads. Since the workloads are homogeneous, the average execution time is also a direct indication of throughput. Each bar in the charts is an average of 10 independent experiments. The variance was less than 5%. The leftmost bar and the dashed line correspond to the theoretical optimal execution time of the benchmark in the workload, which is measured as the standalone execution time on 32 processors, divided by two, i.e. the degree of multiprogramming in the workload.

We instrumented UPMlib to collect statistics (not shown) on the number of thread and page migrations during the execution of the workloads with our page migration algorithms. The workloads incur a significant amount of thread migrations, primarily because of the background load, which competed for processor time with threads from the benchmarks. The thread of the sequential background load migrated continuously between processors due to the I/O operations that forced the thread to block and unblock in the kernel frequently. Due to this interference, the IRIX scheduler forced on average 554 thread migrations per experiment, of which more than 200 were *permanent*, in the sense that the migrated thread stayed on the same node for at least one second. Therefore, there was significant room for improvement from dynamically migrating orphaned pages.

The results in Figure 2 show trends similar to the results of the synthetic experiment. Plain first-touch page placement incurs a slowdown of 71% for BT and 44% for SP, compared to the theoretical optimal execution time of the benchmarks in the specific workloads. The IRIX page migration engine provides a moderate performance improvement of 13% for the BT workload and 20% for the SP workload, compared to plain first-touch page placement. The predictive page migration algorithm achieves solid improvements over first-touch, namely 63% for the BT workload and 40% for the SP workload. The aging algorithm with a sampling frequency of 3 seconds achieves also sizeable improvements of 41% and 32% respectively. The performance of the predictive algorithm is within 5% of the theoretical optimal performance in these experiments. The sensitivity of the aging algorithm to the sampling frequency is once again observed in the experiments.

5.3. Results with the IRIX scheduler

In the final set of experiments we constructed multiprogrammed workloads, in which we executed concurrently four instances of each benchmark and our sequential I/O-intensive background load. Each instance of the parallel benchmarks started with 32 threads and the workload was executed on all 64 processors of an idle sys-

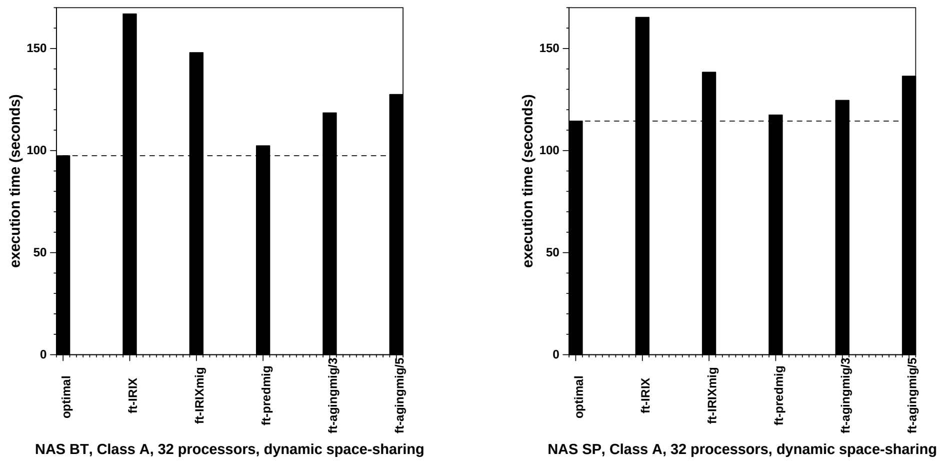


Figure 2. Average execution time of NAS BT and SP, in workloads executed with a dynamic space-sharing scheduler.

tem. The workload execution was controlled completely by the native IRIX scheduler and the processors were space and time-shared by the operating system, after setting the `OMP_SET_DYNAMIC` flag in the benchmarks.

Tracing of the experiments revealed that initially, IRIX started executing all 128 threads of the parallel programs, relying on time-sharing for distributing the system resources fairly. In the course of execution, each program was detecting via the IRIX MP runtime library that the load of the system was high and processor utilization was low and started to progressively reduce the number of active threads. Towards the end of execution of the workloads, the scheduling strategy reverted to a space-sharing scheme, in which there was little time-sharing of processors mostly due to the background load.

Figure 3 illustrates the results from these experiments. The results are averages of 10 experiments in which the variance in execution time was less than 10%. The main observation is that in this set of experiments in which the processors were space and time-shared by the IRIX scheduler, parallel programs are much more sensitive to multiprogramming and the impact of thread migrations to memory performance can be severe. Each program suffers thousands of thread migrations during its execution and the associated slowdown factor compared to the theoretical optimal execution time is as much as 2.1 for BT and 3.3 for SP. The performance improvements from using our dynamic page migration algorithms are much more pronounced in this case, ranging from 70% to 264% compared to plain first-touch page placement and from 52% to 237% compared to first-touch combined with the IRIX page migration engine. These experiments mainly show that under unpredictable scheduling conditions, a powerful page migration mechanism can be very effective in alleviating the performance implications of thread-to-memory affinity. Satisfactory performance can be sustained without necessitating a special-purpose scheduler to establish isolated processor sets, or intervention from the system administrator.

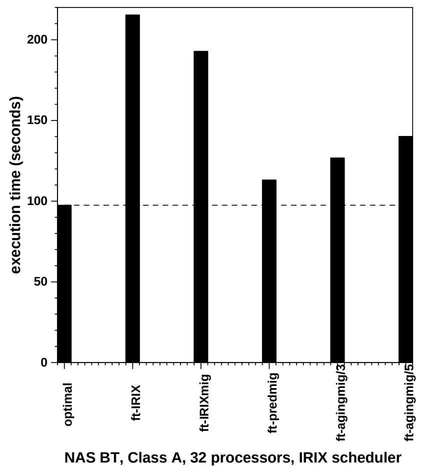
6 Conclusions

This paper presented two algorithms for effective dynamic page migration on multiprogrammed NUMA shared-memory multiprocessors. The algorithms exploited the idea of correlating the scheduling actions of the operating system with information obtained from dynamic monitoring of memory activity of a parallel program. Both algorithms exhibited significant performance improvements with multiprogrammed workloads of OpenMP programs compared to the IRIX page placement and migration mechanisms. To our knowledge, this paper is the first to present results with dynamic page migration for multiprogrammed workloads on an actual NUMA platform.

We are currently investigating ways to enhance the multiprogramming-conscious page migration algorithms in UPMlib, primarily for making them less sensitive to temporary thread migrations performed by the operating system. We are also attempting to improve the selectivity of our algorithms, in order to limit the cost of page migrations by migrating only the most critical pages. Finally, we investigate alternative strategies for identifying memory affinity sets through UPMlib and apply forms of page forwarding upon thread migrations, without relying on reference counting information.

Acknowledgements

This work was supported by the European Commission, through the TMR Contract ERBFMGECT-950062 and in



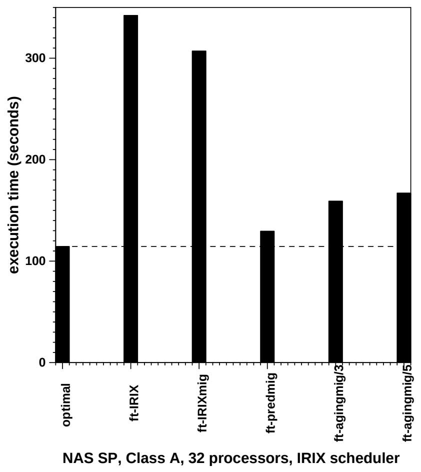


Figure 3. Average execution time of the NAS BT and SP, in workloads executed with the native IRIX scheduler.

part through the ESPRIT IV Project No. 21907 (NANOS), the Greek Secretariat of Research and technology through project No. E.D.-99-566 and the Ministry of Education of Spain through projects No. TIC-98-511 and TIC97-1445CE. The experiments presented in this paper were conducted with resources provided by the European Center for Parallelism of Barcelona (CEPBA).

References

[1] R. Chandra et. al. Scheduling and Page Migration for Multiprocessor Compute Servers. *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 12–24, San Jose, CA, October 1994.

[2] E. Hagersten and M. Koster. Wildfire: A Scalable Path for SMPs. *Proc. of the 5th International Symposium on High Performance Computer Architecture*, pp. 172–181, Orlando, FL, January 1999.

[3] D. Jiang and J. P. Singh. Scaling Application Performance on a Cache-Coherent Multiprocessor. *Proc. of the 26th International Symposium on Computer Architecture*, pp. 305–316, Atlanta, GA, May 1999.

[4] H. Jin, M. Frumkin and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[5] J. Laudon and D. Lenoski. The SGI Origin2000: A cc-NUMA Highly Scalable Server. *Proc. of the 24th International Symposium on Computer Architecture*, pp. 241–251, Denver, CO, June 1997.

[6] M. Marchetti, L. Kontothanassis, R. Bianchini and M. Scott. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. *Proc. of the 9th International Parallel Processing Symposium*, pp. 480–485, Santa Barbara, CA, April 1995.

[7] D. Nikolopoulos et.al. A Case for User-Level Dynamic Page Migration. *Proc. of the 14th ACM International Conference on Supercomputing*, pp. 119–130, Santa Fe, NM, May 2000.

[8] D. Nikolopoulos et.al. UPMlib: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Cache-Coherent NUMA Multiprocessors. *Proc. of the 5th ACM Workshop on Languages, Compilers and Runtime Systems for Scalable Computers*, Rochester, NY, May 2000.

[9] L. Noordergraaf and R. Van der Pas. Performance Experiences on Sun's Wildfire Prototype. *Proc. of Supercomputing'99*, Portland, OR, November 1999.

[10] C. Polychronopoulos, N. Bitar and S. Cleiman. Nano-Threads: A User-Level Threads Architecture. CSRD Technical Report No. 1297, University of Illinois at Urbana-Champaign, 1993.

[11] Silicon Graphics Inc. IRIX 6.5 Operating System Man Pages. http://techpubs.sgi.com, Accessed November 1999.

[12] V. Soundararajan et.al. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. *Proc. of the 25th International Symposium on Computer Architecture*, pp. 342–355, Barcelona, Spain, June 1998.

[13] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. *Proc. of the 12th ACM Symposium on Operating System Principles*, pp. 159–166, Litchfield Park, December 1989.

[14] B. Verghese, S. Devine, A. Gupta and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 279–289, Cambridge, MA, October 1996.