A Case for User-Level Dynamic Page Migration

Dimitrios S. Nikolopoulos[1], Theodore S. Papatheodorou[1], Constantine D. Polychronopoulos[2], Jesús Labarta[3] and Eduard Ayguadé[3]

Department of Computer[1] Engineering and Informatics University of Patras Rion, 26 500, Patras, Greece
dsn,tsp@hpclab.ceid.upatras.gr

Department of Electrical[2] and Computer Engineering University of Illinois at Urbana-Champaign Urbana, IL 61 801
cdp@csrd.uiuc.edu

Department of Computer Architecture[3] Polytechnic University of Catalonia c/Jordi Girona 1-3, Modul D6, 08034, Barcelona, Spain
jesus,eduard@ac.upc.es

ABSTRACT

This paper presents *user-level dynamic page migration*, a runtime technique which transparently enables parallel programs to tune their memory performance on distributed shared memory multiprocessors, with feedback obtained from dynamic monitoring of memory activity. Our technique exploits the iterative nature of parallel programs and information available to the program both at compile time and at runtime in order to improve the accuracy and the timeliness of page migrations, as well as amortize better the overhead, compared to page migration engines implemented in the operating system. We present an adaptive page migration algorithm based on a competitive and a predictive criterion. The competitive criterion is used to correct poor page placement decisions of the operating system, while the predictive criterion makes the algorithm responsive to scheduling events that necessitate immediate page migrations, such as preemptions and migrations of threads. We also present a new technique for preventing page ping-pong and a mechanism for monitoring the performance of page migration algorithms at runtime and tuning their sensitive parameters accordingly. Our experimental evidence on a SGI Origin2000 shows that unmodified OpenMP codes linked with our runtime system for dynamic page migration are effectively immune to the page placement strategy of the operating system and the associated problems with data locality. Furthermore, our runtime system achieves solid performance improvements compared to the IRIX 6.5.5 page migration engine, for single parallel OpenMP codes and multiprogrammed workloads.

1. INTRODUCTION

The current trends in parallel computing indicate that shared memory multiprocessor architectures converge to a common model in which multiple single-processor or symmetric multiprocessor (SMP) nodes are interconnected via a fast network and equipped with additional hardware support to provide the abstraction of shared memory to the programmer [8]. This architectural model was adopted in earlier research prototypes such as the Stanford DASH and FLASH multiprocessors [14] and later, in high-end commercial products including the Sequent Sting [17], the HP/Convex Exemplar [4], the SGI Origin2000 [15] and the Sun Wildfire [10]. Although these architectures promise to meet the requirements of scalability and programmability, they also present the challenging problem of the Non-Uniformity of Memory Access latencies (NUMA). NUMA introduces the notion of distance between a thread of a parallel program and the data that this thread accesses. A thread running on one node of the system may access data located in the memory of another node. These memory accesses, albeit transparent to the programmer, have significantly higher latency than accesses to local memory and constitute one of the main sources of performance degradation.

Dealing with the latency of remote memory accesses in software poses a tradeoff with respect to the simplicity of parallel programming models for shared memory multiprocessors. Ideally, the main system software components, namely the compiler, the runtime system and the operating system should cope with remote memory accesses by applying program transformations and runtime techniques to collocate each thread with the data that the thread accesses more frequently. However, despite the effort spent in optimizing system software for NUMA multiprocessors, experiences with real applications make evident that significant programming effort is still required in order to tune data locality and achieve acceptable scalability on moderate and large-scale systems [12]. In particular, the programmer must be aware of the operating system's page placement policy and either modify the program to adapt its memory access pattern to the operating system's policy, or bypass the operating system and hardcode in the program a customized page placement scheme. Both approaches compromise the simplicity of the shared memory programming model, in terms of the effort needed to optimize a program and the subtleties of the programming constructs which are required to tune data placement. The problems related to page placement and locality have already introduced major considerations with respect to the widely popular OpenMP shared memory programming standard and raised the dilemma whether data distribution directives should be introduced in OpenMP or not [16].

ICS 2000 Santa Fe New Mexico USA

Previous research on NUMA memory management at the operating system level has shown that the most effective page placement is frequently achieved by a *first-touch* strategy, in which the processor that reads or writes to a page in shared memory first, maps this page to a local memory module [6; 18]. First-touch is usually sufficient for achieving reasonable amounts of data locality in applications with regular reference patterns. In order to deal with possible inconsistencies between first-touch page placement and dynamic memory reference patterns, researchers have proposed the use of dynamic page migration by the operating system [25]. Dynamic page migration uses per-node reference counters attached to each page in memory and applies a competitive algorithm to automatically migrate a page, if a node other than the home node of the page accesses the page significantly more frequently. Page migrations are performed transparently to the program with the aid of the operating system, which is responsible for maintaining TLB coherence and copying the page to its new home. Page migration can reduce memory latency by converting remote memory accesses into local ones. In particular for cache-coherent systems, page migration converts the secondary cache misses which are satisfied from remote memory modules, into misses satisfied from local memory modules. This optimization can reduce the latency of secondary cache misses by a factor of three to five [24].

Although the initial simulation results for dynamic page migration with engineering and commercial workloads were promising [24; 25], practical implementations of dynamic page migration in real operating systems have not proven to be as effective as expected yet [10; 12; 20]. This is attributed primarily to the poor accuracy and timeliness of page migrations triggered by the operating system, due to the inability of the operating system to associate the information from reference counters with the semantics of the program and at the same time amortize the cost of triggering page migrations and copying pages. Furthermore, the operating system must deal with corner cases such as ill application behavior, memory pressure and global resource management constraints that limit the flexibility of a page migration mechanism. As a consequence, the use of dynamic page migration in contemporary operating systems for scalable NUMA multiprocessors is either discouraged or limited to trivial cases, such as the replication of kernel and program code across nodes [23].

In this paper, we propose a new runtime technique that leverages runtime monitoring of memory activity and dynamic page migration at user-level, to reduce the latency of remote memory accesses on cache-coherent NUMA multiprocessors. Our method is based on the idea that with the aid of a runtime system that reads the page reference counters, a parallel program has the means to monitor its memory performance at well-defined execution points, at which the program can obtain an accurate snapshot of its complete memory reference pattern. Most parallel applications are iterative in nature, in the sense that they execute the same parallel computation for a number of iterations. We exploit the iterative nature of parallel applications to perform accurate performance monitoring at user-level. At the same time, with the aid of the compiler, we identify *hot* memory areas of the application virtual address space which are likely to contain candidate pages for migration and instrument the programs to invoke the monitoring and page migration services of the runtime system. The decisions of what memory pages should be migrated and when are taken entirely within the runtime system, while they are still transparent to the program and subject to the enforced global memory allocation strategies of the operating system.

Our user-level page migration engine is effectively integrated with the operating system scheduler, which provides scheduling notifications to the runtime system. The runtime system intercepts thread migrations, which occur frequently in multiuser/multiprogrammed systems and harm data locality, since they break the association between threads and data for which threads have affinity. The runtime system treats thread migrations as a trigger to switch the page migration policy into an aggressive mechanism that eagerly forwards the pages for which a thread has affinity to the node to which a thread is migrated, if the thread is likely to stay on this node long enough to justify the cost of forwarding pages.

User-level dynamic page migration enhances accuracy, timeliness, flexibility and responsiveness compared to a dynamic page migration engine implemented in the operating system. Accuracy is enhanced by applying page migration at execution points at which the reference counters provide an accurate snapshot of the reference pattern of the parallel program and by restricting the page migration mechanism to hot memory areas of the address space. Timeliness is improved because the runtime system has the ability to identify quickly the pages that should migrate —in most cases after executing one iteration of the parallel computation— and migrate them altogether as soon as possible. Among other advantages, this approach amortizes better the cost of page migrations, since this cost is paid cumulatively at the beginning of execution to stabilize the memory performance at good levels in the long-term. Flexibility is improved by letting the runtime system customize the page migration algorithm for each hot memory area separately, based on the observed memory activity in each area. Responsiveness finally, is improved because the runtime system responds promptly to unpredictable runtime scheduling events that undermine data locality.

We consider user-level page migration as a methodology towards implementing *adaptive* parallel programs, that is, programs that monitor their execution conditions and their performance at runtime and adapt to these conditions by themselves, without relying on the enforced systemwide resource management policies. Embedding adaptability to parallel programs is of particular importance in modern multiprogrammed multiprocessors, in which a program can not make safe assumptions on resource availability.

In the rest of this paper, we present the design, the algorithms, some implementation details and a preliminary performance evaluation of a runtime system for user-level dynamic page migration. We developed the runtime system on the SGI Origin2000, to support programs parallelized with OpenMP. Although our runtime system encompasses several novel ideas and solutions to challenging research problems related to page migration, the emphasis in this paper is placed on providing a concise overview of user-level dynamic page migration and demonstrating some of the performance advantages of our approach. Sensitivity analyses and comparative studies of relevant algorithms are left as issues for further investigation. We provide experimental results which demonstrate that OpenMP programs with user-level

dynamic page migration are effectively immune to the page placement strategy of the operating system and have robust, non-degrading performance even with the worst possible initial page placement. This result suggests that it might not be necessary to compromise the simplicity of OpenMP by introducing data distribution directives. Our runtime system achieves also sizeable performance improvements compared to the best-performing IRIX 6.5.5 memory placement and migration strategy. The observed improvements are of up to 28% for single parallel benchmarks and 49% for multiprogrammed workloads, executed with space and time-sharing by the native IRIX scheduler.

The rest of this paper is organized as follows: Section 2 outlines the motivation and the ideas behind user-level dynamic page migration. Section 3 presents our page migration algorithms. Section 4 discusses implementation details. Section 5 presents our experimental evidence. Section 6 discusses related work and Section 7 concludes the paper.

2. USER-LEVEL DYNAMIC PAGE MIGRATION

2.1 Motivation

Although results from simulations in previous works have indicated that dynamic page migration is an effective technique to reduce memory latency on cache-coherent NUMA systems [24; 25], the real implementations of dynamic page migration that we are aware of, namely the implementation in IRIX for the SGI Origin2000 [15] and the implementation in Solaris for the Sun Wildfire prototype [10] have not demonstrated analogous results. Performance evaluations of the Origin2000 from its vendors have not reported any results with parallel benchmarks using the IRIX page migration engine [15; 23]. A preliminary evaluation of the Sun Wildfire prototype with OLTP workloads reported also no results with dynamic page migration enabled in the system [10]. A more recent evaluation of the Wildfire [20] has shown with a synthetic experiment that page migration can improve computational throughput in the long-term, but suffers from poor responsiveness and performs significantly worse compared to coherent memory replication at the granularity of a cache line. A relevant study of the complete SPLASH-2 benchmark suite on a large-scale Origin2000 has shown that page migration was ineffective in dealing with the problems introduced by the operating system's page placement strategy and some programs required hand-tuned page placement to scale reasonably [12]. Several application studies with the OpenMP programming standard for shared memory multiprocessors have reported significant performance losses due to poor page placement and locality [5; 21]. Nevertheless, none of these studies leveraged dynamic page migration to alleviate these problems. On the contrary, researchers have reverted to programming models based on message-passing such as HPF and MPI to compensate for poor data locality.

The thesis of this paper is that dynamic page migration can effectively solve performance problems related to page placement. In particular, we believe that the factors that limit the effectiveness of kernel-level implementations of dynamic page migration are mainly the accuracy, the timeliness and the flexibility of the page migration policies enforced in the operating system. Dynamic page migration can be improved in these three aspects by taking into account the semantics of the program and exploiting information available to the program from the compiler and the operating system to trigger effective page migrations in response to critical external events.

2.2 Dynamic Page Migration in the Operating System

Page migration engines implemented in the operating system use either an interrupt-based or a sampling-based mechanism for triggering page migrations. IRIX uses an interrupt-based mechanism on the SGI Origin2000 [15; 22] while Solaris uses a sampling-based mechanism on the Sun Wildfire [20].

In an interrupt-based page migration mechanism, the hardware reference counters of a node track the number of local and remote accesses to each page and send an interrupt to a processor in the node if the difference between the local accesses and the accesses by some remote node exceeds a hardwired threshold. The interrupt handler decides to migrate the page or not, according to a set of criteria posed by the page migration policy. The advantage of an interrupt-based mechanism is that the operating system detects that a page has excessive remote accesses as soon as the counters accumulate this information. The disadvantage of an interrupt-based mechanism is that the instantaneous values of the reference counters that trigger page migrations cannot be associated with the semantics of the program. This makes an interrupt-based mechanism prone to transitory effects that occur in certain execution phases of a program. The operating system might perform unnecessary and inaccurate page migrations, if the counters are biased by bursts of remote references due to compulsory cache misses, cache reloads after context switches, or phase changes in the reference pattern of the program. The counters may also be biased by obsolete page reference history and therefore prevent pages from migrating, although the actual execution status of the computation might instruct otherwise. The latter can happen upon migrations of threads. These events necessitate immediate page migrations, since they move a thread away from the pages in its *memory affinity set*, that is, the pages that are accessed more frequently by this thread. A page migration algorithm based on the observed reference history of pages can collocate each thread with its memory affinity set on the same node [2]. However, if a thread migrates, the algorithm should aggressively forward the pages in the memory affinity set of the migrated thread to the new home node of the thread, without relying on the past reference history of these pages.

The aforementioned implications force the operating system to use fairly complex and overly conservative page migration policies with an interrupt-based mechanism. IRIX uses a multitude of heuristics to deal with corner cases and transitory effects [22]. The kernel uses a dampening threshold and migrates a page only if a processor receives many consecutive migration interrupts for the page. Moreover, the kernel uses a freezing threshold to limit the maximum number of migrations for a page and a melting threshold to reenable migrations for a page which is frozen for a significant amount of time. The kernel also ages the reference counters in order to cope with the possibility of reading counters with obsolete reference information. The operations that control freezing, melting and aging of counters are performed periodically, at the rate of one virtual memory page every 10 ms, in order

to compensate for the overhead. The periodic control operations and the dampening filter can prevent timely page migrations when the application reference pattern is stabilized, as well as deteriorate the responsiveness of the page migration mechanism to the operating system scheduling decisions.

A sampling-based mechanism for dynamic page migration polls periodically the reference counters of some pages and applies the page migration policy for those pages that have excessive remote references. Compared to an interrupt-based mechanism, a sampling-based mechanism has the advantage that if the sampling frequency is carefully selected, the operating system can deal more easily with transitory effects and possibly, apply more sophisticated page migration policies. The disadvantage of a sampling-based mechanism is its sensitivity to the sampling frequency and the overhead of monitoring pages for migration. Different applications may need different sampling frequencies, according to the granularity and the repeatability of the memory reference pattern. Furthermore, the number of pages that can be scanned and migrated upon each invocation of the page migration engine is limited because of the high overhead of triggering and performing a page migration, which is typically in the order of 1 ms. In cases in which applications have large resident sets the operating system might spend a significant amount of execution time before detecting the actual hot memory areas and start migrating pages.

2.3 Dynamic Page Migration from the Runtime System

We propose a framework for accurate, timely and flexible dynamic page migration, by the means of a runtime system that implements a customizable yet transparent user-level page migration engine. We designed and implemented a prototype of the runtime system for codes parallelized with OpenMP on the SGI Origin2000.

Our runtime system uses compiler-generated hints that identify hot memory areas which are likely to contain pages eligible for migration. The current prototype identifies as hot memory areas the shared arrays which are both read and written in sets of disjoint parallel constructs, delimited by OpenMP clauses. The native OpenMP code is transparently instrumented by the compiler with calls that invoke the page migration runtime system. The instrumentation exploits the iterative nature of the vast majority of parallel codes. The runtime system's page migration engine is invoked at the end of each outer iteration of the program, which encompasses the whole parallel computation. At these points, the runtime system can obtain a snapshot of the complete memory reference pattern of the program, by reading the reference counters for all the pages in the hot areas. Therefore, the runtime system is in a position to make very accurate decisions for migrating pages in a way that matches exactly the reference pattern. This approach differentiates from interrupt-based or sampling-based schemes, in which the snapshots of the counters that trigger page migrations are not correlated with the actual status of the computation, thus leading frequently to suboptimal page migration decisions. The page migration runtime system seeks the optimal page placement with respect to the reference pattern, after the execution of a single iteration of the program. Optimality is attained when each page is placed in the node with the processors that access this page more frequently and the maximum latency due to remote accesses by any other node is the minimum among all possible placements of the page. The necessary page migrations to achieve the optimal placement can be dilated only because of ping-pong of pages due to page-level false sharing. Pairwise page ping-pong can be detected in at most two iterations in our runtime system (cf. Section 3.3).

User-level page migration comes closer to the semantics of a parallel program compared to kernel-level page migration, since the reference counters are sampled at execution points at which the information in the counters is not biased by transitory effects. In addition, the runtime system batches most, if not all, page migrations in a single invocation at the beginning of the program. This strategy amortizes well the cost of page migrations over time, compared to an interrupt-based or a sampling-based strategy, in which page migrations are non-uniformly distributed throughout the execution time of the program. Furthermore, the runtime system is in a position to monitor the latency of remote accesses with feedback from the reference counters for each hot memory area separately and customize the migration policy parameters according to the observed memory activity in each area.

The runtime system intercepts dynamic changes of the effective processor set on which a program executes, due to preemptions and migrations of threads. These events are intercepted by polling variables in shared memory, which are kept up-to-date by the operating system. Thread migrations are treated by the runtime system as triggers for using an aggressive predictive page migration scheme for pages in the memory affinity sets of migrated threads, (cf. Section 3.2).

The runtime system and the operating system share the same hardware support for reading reference counters, migrating pages, and maintaining memory consistency thereafter. The runtime system pays an additional cost for using these services, since some of the required calls have to cross the kernel boundaries. However, this cost can be easily overlapped by performing page migrations in parallel with the execution of the program. In addition, user-level page migrations are and should be subject to resource management constraints of the operating system, which may reject requests to migrate pages under memory pressure. The distinctive features of user-level page migration compared to kernel-level page migration engines are accuracy, timeliness and good amortization of the overhead of page migrations.

Although the design of our runtime system is driven by the iterative nature of parallel programs, the runtime system supports also non-iterative programs, as well as programs that exhibit fine-grain phase changes in their memory reference pattern within the parallel computation. The former are handled with a sampling-based mechanism that serves the hot memory areas round-robin, while the latter are handled with a page forwarding mechanism that tries to perform the necessary page migrations before a phase change in the program's access pattern, in a record-replay fashion. Both mechanisms are in an early evaluation stage, therefore we do not examine them further in the remainder of this paper.

3. PAGE MIGRATION ALGORITHMS

The guidelines for the design of our page migration algorithm are flexibility and adaptability to the execution conditions of the program and the characteristics of the memory regions on which the algorithm operates. Figure 1 shows

the pseudocode of the algorithm. This algorithm runs every time the page migration runtime system is invoked at the end of each outer iteration of an iterative parallel program. At these points of execution, the algorithm scans all hot memory areas identified by the compiler.

The main body of the algorithm (lines 2–12) identifies candidate pages for migration and migrates the candidate pages that satisfy a set of criteria. Candidates for migration are identified using either a competitive or a predictive page migration criterion, which are explained in Sections 3.1 and 3.2 respectively. The code in lines 13–16 embeds a self-tuning discipline in the algorithm, based on feedback obtained from the page reference counters, with respect to the effectiveness of the algorithm in reducing the latency of remote memory accesses. The code in lines 17–20 improves the accuracy and reduces the overhead of the algorithm by tracking the page migration activity in each memory area separately. The algorithm is explained in detail in the following paragraphs.

3.1 Competitive Criterion

Equation 1 shows the competitive criterion used in our page migration algorithm. For each node i in the system and for each page in a hot memory area the criterion checks if the following inequality is satisfied:

$$ral_{tot}(i,h) > (r_u(i,h)/l_u) \cdot lal_{tot} + ml \quad (1)$$

We denote as h the home node of the page. The home node is the node that caches the page in a local memory frame. We estimate the total latency due to remote memory accesses from node i to the page ($ral_{tot}(i,h)$), and the total latency of local memory accesses from the processors on the home node h of the page (lal_{tot}) to the page. A page is considered as candidate for migration, if $ral_{tot}(i)$ is at least as much as lal_{tot}, multiplied by the ratio $r_u(i,h)/l_u$, in which $r_u(i,h)$ is the latency of a single uncontended memory access from node i to node h, and l_u is the latency of a single uncontended local memory access from a processor in the home node h. The values of $r_u(i,h)$ and l_u are obtained from the system's technical specifications. For the SGI Origin2000, $l_u \approx 300$ ns and $r_u(i,h) \approx 300 + 100 \cdot d(i,h)$ ns, where $d(i,h)$ is the distance in network hops between nodes i and h in the system.

We compute $ral_{tot}(i,h)$ by estimating first the contended latency for a single remote memory access from node i to node h ($r_c(i,h)$) and then multiplying this estimation by the number of remote accesses from i to h, which is obtained from the hardware reference counters. In the Origin2000, we use the following formula to estimate $r_c(i,h)$:

$$r_c(i,h) \approx r_u(i,h) + 50 \cdot c(h) \quad (2)$$

where $c(h)$ is a contention factor. We set $c(h)$ heuristically, to be equal to the number of nodes that have more accesses to the page than the home node h. This information can be obtained from the hardware reference counters. For each node that contends for the page, we add a constant factor of 50ns to the base uncontended memory latency, in order to compute the contended memory latency. This value is extracted from a previous study that computed the latencies in the Origin2000 memory hierarchy [11].

Our competitive criterion is accurate, in the sense that a page migration will always reduce the maximum latency due to remote accesses for the processors that compete for the page. Assuming that each remote memory access from the same node to the same page has constant latency, our competitive criterion is equivalent to a criterion that compares the number of references obtained by the reference counters [2]. In reality, on cache-coherent systems the remote memory access latency is variable and depends on the type of cache miss that triggers the remote access, the distance in network hops between the home node and the accessing node and contention at the memory module to which the access is issued.

If the competitive criterion is satisfied by more than one nodes, the page is migrated to the node that has the highest ratio of $ral_{tot}(i,h)/lal_{tot}$. The migration latency ml is added in Equation 1 to offset the cost of page migrations by the earnings from reducing the number of remote memory accesses. We calculate the migration latency experimentally, using microbenchmarks. The runtime system supports also a variant of the algorithm in which the migration latency is not taken into account. This optimistic algorithm is based on the observation that if most page migrations are performed early in the execution of the program, then it is likely that their overhead will be offset in the long-term, even if this is not reflected in the estimated memory latencies.

3.2 Predictive Criterion

The predictive criterion is used in the place of the competitive criterion, when the runtime system detects migrations of threads from the operating system. The page migration algorithm switches from the competitive to the predictive criterion if the scheduling notifications from the operating system indicate that either the number of threads used by the program to execute parallel code has changed and/or some of the threads have migrated to other nodes.

The starting point for the predictive criterion is the assumption that the competitive algorithm achieves an optimal page placement, in which the threads in the home node of a page are the threads that access the page more frequently and the maximum latency due to remote memory accesses to each page is minimized. At this point the program has a stable repetitive reference pattern and the number of memory references from each node is expected to increase linearly to the number of iterations of the program at a constant rate.

The critical observation that motivates the predictive criterion is that if across two iterations of the program the number of local accesses from the home node of a page increases at a lower rate with respect to previous iterations, while the number of accesses from some remote node to the same page increases at a faster rate with respect to previous iterations, then it is likely that the information in the reference counters is obsolete due to one of the following reasons: the threads that were running on the home node of the page have migrated; the threads that were running on the home node of the page were preempted by the operating system and their computation was taken up by threads running on remote nodes in subsequent iterations of the program; or some preempted threads that used to access the page more frequently in the past were resumed in remote nodes and access the page again[1].

[1]The second and third cases occur if the program adjusts

```
(1) for each hot memory area {
(2)   identify candidate pages for migration with either the competitive
(3)   or the predictive criterion;
(4)   if the predictive criterion is used {
(5)     apply ping-pong prevention algorithm;
(6)   }
(7)   for each candidate page {
(8)     if the page must not be frozen and is not likely to ping-pong {
(9)       migrate it;
(10)      update page bookkeeping data;
(11)    }
(12)  }
(13)  estimate the maximum memory latency due to remote accesses in this area;
(14)  if this latency increases compared to previous invocations {
(15)    tune the selectiveness of the migration algorithm;
(16)  }
(17)  if the area had no candidate pages for migration  {
(18)    if the area had no candidates in several previous invocations  {
(19)      mark the area as cold;
(20)    }
(21)  }
(22)}
```

Figure 1: Pseudocode for the page migration algorithm.

Equation 3 shows the predictive page migration criterion for a page:

$$\exists i, racc(i,t) > racc(i,t-1) \wedge lacc(t) < lacc(t-1) \quad (3)$$

in which $t-1$ and t correspond to two successive outer iterations of the parallel program, $racc(i,t)$ is the number of remote accesses from node i to the page during iteration t, and $lacc(t)$ is the number of local accesses from the home node during iteration t. If the criterion in Equation 3 is satisfied by some node i, and there is at least one thread that has migrated to node i, then the algorithm migrates the page to node i, speculating that the observed anomaly in the reference rates is due to the thread migration. If more than one nodes satisfy the predictive criterion, the page is migrated to the node that issued the most remote accesses during iteration t.

The predictive criterion identifies pages that belong to the memory affinity set of a migrated thread within at most two iterations of the program and migrates these pages without waiting for the counters to collect reference history that would justify migration according to the conservative competitive page migration criterion. The page migration algorithm switches back to the competitive criterion, if it detects that the predictive criterion is no longer satisfied by any page in the hot memory areas.

3.3 Dealing with Ping-Pong

Dynamic page migration algorithms suffer from the page *ping-pong* problem. Ping-pong occurs when the migration algorithm starts bouncing a page between two nodes, without deciding to freeze the page to any of the nodes. Ping-pong is an artifact of ill application behavior which may introduce page-level false sharing. The solution that we adopt to circumvent ping-pong is in the direction of preventing rather than detecting it after its occurrence. The ping-pong prevention algorithm maintains a history entry for each page selected for migration. The history tracks the current and the previous home node of the page if the page has already migrated before. If the page migration algorithm selects to migrate a page for a second time and the candidate target node was the home node of the page before its first migration the page freezes and does not migrate. This strategy prevents ping-pong under the assumption that the threads running in the two competing nodes have not migrated. Therefore, the ping-pong prevention mechanism is activated only while the competitive page migration criterion is active.

dynamically the number of threads used to execute parallel constructs as a response to changes of the system load.

3.4 Tuning the Performance of the Algorithm at Runtime

Our page migration algorithm uses two metrics for self-tuning. The first metric is the estimated maximum latency due to remote memory accesses seen by any processor, for each memory area on which the algorithm operates. The algorithm maintains different migration thresholds for each memory area and checks if between successive invocations of the algorithm on the same memory area the maximum memory latency due to remote accesses increases or decreases (Figure 1, lines 13–16). If this latency increases, the algorithm improves its selectiveness by increasing the ratio $r_u(i,h)/l_u$ in Equation 1 by a constant factor. The second metric used by the algorithm is the number of pages selected for migration upon each invocation of the algorithm on the same memory area. If no pages are selected for migration across several invocations of the algorithm, the area is marked as cold and the algorithm does not scan it further in subsequent invocations (Figure 1, lines 17–20), unless new thread migrations trigger a switch of the page migration criterion to predictive. This optimization improves further the accuracy of the algorithm and reduces drastically the runtime overhead.

4. IMPLEMENTATION DETAILS

Our runtime system requires two system services, a service to access the reference counters of memory pages and a service to migrate a range of the program's virtual address space to a specified node in the system. The Origin2000 operating system (IRIX 6.5.5), provides both services, via its memory management control interface (`mmci`).

Migration of a virtual address space range can be requested at user-level with the `migr_range_migrate()` system call [22]. User-level memory migrations in IRIX move ranges of the virtual address space between different *Memory Locality Domains* (MLDs). A MLD virtualizes the memory of a node in the system. NUMA-sensitive memory allocation, placement and migration can be implemented by associating ranges of the virtual address space to MLDs. In our runtime system, we preinitialize as many MLDs as the number of nodes in the system and bind each MLD to a different node. In this way the runtime system can use directly the id's of the MLDs as if they were the physical nodes in the system.

The SGI Origin2000 memory modules are equipped with hardware reference counters. Each hardware memory page, the size of which is 4 Kbytes, has one 11-bit hardware counter per node in the system, for configurations of up to 64 nodes (128 processors). The counters track all types of accesses to memory during the service of L2 cache misses by the cache coherence protocol. IRIX maps the 11-bit hardware counters of each page to software-extended 32-bit counters which are maintained in the kernel. When an 11-bit hardware counter of a page overflows, an interrupt is generated and the operating system adds the contents of all the hardware counters of this page to the corresponding software-extended counters. Both hardware and software-extended reference counters can be accessed via the `/proc` interface and their values can be either stored in a user-allocated buffer with an `ioctl` call, or memory mapped to the application's address space. We selected the former approach since the latter required root privileges on the system on which we experimented. In order to deal with the asynchrony between hardware and software-extended counters, our runtime system reads both sets of counters and sums their values in all cases in which the value of a hardware counter is lower than the value of the corresponding software-extended counter [22].

We explicitly manage the reference counters in the runtime system, in order to map to virtual instead of physical memory pages. Furthermore, we batch page migration requests for consecutive pages in the virtual address space whenever possible to reduce the overhead of multiple system calls. Since placing the overhead of scanning pages and triggering page migrations on the critical path of the program is undesirable, the runtime system uses a separate thread for accessing counters, managing buffers and migrating pages. This thread, called the *memory manager*, communicates with the master thread of the program. The master thread triggers the memory manager at the end of outer iterations and the memory manager applies the specified page migration policy. The execution of page migrations is overlapped with the execution of the program's threads. For this purpose, the runtime system sacrifices one processor to execute the memory manager. This limitation is not too restrictive. Previous studies on the Origin2000 have shown that dedicating one node to execute operating system code alleviates the interferences between the operating system and parallel programs and leads often to better speedups [12].

We also provide an alternative implementation in which the memory manager, instead of communicating with the master thread, wakes up periodically and scans a number of pages in the hot memory areas. This implementation targets parallel programs which are not inherently iterative and have no apparent points of execution at which the page migration runtime system should be invoked. The sampling period and the number of pages scanned upon each invocation can be tuned by the user, or automatically by the runtime system.

The runtime system identifies the effective processor set on which OpenMP programs execute at any instance during the execution of the programs, via the `schedctl(SETHINTS)` call. This call polls the `t_cpu` field of the private data area (`prda`) of IRIX threads and records the physical processor on which each thread ran during the last time quantum. Polling is performed before and after the execution of each parallel OpenMP construct. This technique records thread migrations at a relatively coarse granularity, in the sense that thread migrations that occur during the execution of parallel constructs are not captured by the runtime system. This decision is made deliberately, in order to avoid biasing our predictive page migration mechanism from the effects of temporary, short-term thread migrations. The intuition behind our approach is that due to the overhead and the aggressiveness of the page forwarding mechanism, the predictive criterion should be triggered only if a thread that was migrated to node i, is likely to stay on node i for sufficiently long time to justify the decision of moving the pages of its memory affinity set to node i. In the actual implementation, the predictive criterion is triggered if a thread migrates and stays on the same node for the execution of two consecutive parallel OpenMP constructs. This heuristic makes the predictive mechanism more robust and works well with schedulers that give relatively high weight on maintaining the affinity of threads to processors.

The pages selected for migration by the runtime system are maintained in a hash table. This allows fast retrieval of page id's for updating bookkeeping information for each page, such as the page history for ping-pong detection.

5. PERFORMANCE EVALUATION

We linked our runtime system for user-level dynamic page migration with three FORTRAN codes from the NAS benchmarks suite, parallelized with OpenMP [13]. We used the BT and SP application benchmarks and the CG kernel benchmark. BT is a simulated CFD application that solves 3-D Navier Stokes equations, using Alternating Direction Implicit (ADI) factorization. SP is similar to BT, but uses the Beam-Warming approximate factorization. CG approximates the smallest eigenvalue of a large, sparse matrix with the Conjugate-Gradient method. We ran experiments with the Class A problem size. The selected problem sizes scaled well up to 32 processors on the system on which we experimented, a 64-processor SGI Origin2000 with R10000 processors clocked at 250 MHz, 4 Mbytes of L2 cache per processor and 8 Gbytes of memory. The speedups tended to flatten beyond 32 processors.

The codes were a priori hand-tuned by their provider, to exploit the first-touch page placement strategy of IRIX [13]. This was achieved by executing one cold-start iteration of the complete computation before starting the main loop, in order to warm-up the memory and the caches. IRIX uses an interrupt-based page migration engine and a competitive algorithm that migrates a page if the difference between the local and remote accesses to a page exceeds a fixed threshold stored in a hardware register (cf. Section 2). Automatic page migration is by default disabled in the system, however, the user can enable page migration on a per-program

basis by setting the _DSM_MIGRATION environment variable. For all the experiments, we used the default values for the tunable parameters of the IRIX page migration engine, such as the migration, freezing, and dampening thresholds [22]. Changing the values of the tunable parameters of the IRIX page migration engine required root privileges, therefore we could not conduct tests to assess the impact of modifying the aggressiveness of the IRIX page migration policy.
We instrumented the OpenMP codes by hand to mark the hot arrays and use the page migration runtime system. The instrumentation excluded the first iteration of each benchmark to alleviate cold-start effects. The instrumentation pass for user-level dynamic page migration is currently being integrated in the OpenMP NanosCompiler [1].

5.1 Experiments with Single Parallel Benchmarks on an Idle System

We executed the benchmarks initially on an idle system, using 16 and 32 processors. In the first set of experiments we ran the unmodified versions of the benchmarks with the IRIX first-touch page placement and with first-touch and the dynamic page migration mechanism of the IRIX kernel enabled for each benchmark. We call these schemes `ft-IRIX` and `ft-IRIXmig` respectively. We then ran the instrumented versions of the benchmarks, utilizing our user-level page migration engine with first-touch page placement enabled and the IRIX page migration engine disabled. These versions utilized our competitive page migration algorithm. We call this scheme `ft-umig`. The three leftmost shaded bars of the charts in Figure 2 show the results from these experiments.
We repeated the same experiments on an idle system, after applying the following two modifications to the benchmarks. In the first case, instead of using the default first-touch page placement, we activated the IRIX round-robin page placement scheme, by setting the _DSM_PLACEMENT environment variable. We call these schemes, `rr-IRIX`, `rr-IRIXmig` and `rr-umig` respectively. The three middle bars in the charts of Figure 2 show the results from the experiments with round-robin page placement. In the second case, we reactivated first-touch and used the IRIX `mp_suggested_numthreads()` call to force the first iteration of each benchmark to execute sequentially. With the first-touch page placement policy, this modification implied that all pages used by the benchmarks were allocated on a single node, thus emulating an hypothetical page placement scheme which corresponds to the worst case for benchmarks tuned to use first-touch. In this worst-case scenario, all but two threads of each benchmark (i.e. the two threads that ran on the node where all data was placed), had to fetch data from remote memory upon every L2 cache miss. Since the benchmarks were executed on 8 nodes (16 processors) and 16 nodes (32 processors) of the system, 87.5% (7/8) and 93.75% (15/16) of the pages in the resident set of the benchmarks respectively were badly placed. Furthermore, all data was fetched from a single memory module, therefore contention at this memory module was exacerbated. We conducted this brute-force experiment in order to evaluate accurately the extent to which automatic kernel-level or user-level page migration can fix locality problems due to poor page placement. We call the worst-case page placement schemes `sn-IRIX`, `sn-IRIXmig`, and `sn-umig`, where the prefix `sn-` stands for single-node. The three rightmost striped bars of the charts in Figure 2 show the results from these experiments.
The results in the charts are averages of three independent experiments. The variance in execution time was negligible.

5.2 Performance with Worst-Case Page Placement

Our first observation stems from performing pairwise comparisons between shaded and stripped bars with the same color. We detect that the OpenMP codes exhibit strong sensitivity to the page placement strategy. If page migration is not employed, a worst-case initial page placement slows down the benchmarks by a factor of at least 1.26 (BT on 16 processors) and at most 2.47 (SP on 32 processors), compared to first-touch. The average slowdown factor due to the worst-case page placement strategy is 1.79. Using dynamic page migration in the IRIX kernel reduces this slowdown factor slightly, down to 1.76 (`ft-IRIXmig` compared to `sn-IRIXmig`). The user-level page migration runtime system takes the slowdown factor down to 1.12, 36% less than the IRIX page migration engine. This practically means that with the user-level page migration mechanism, the benchmarks were on average penalized by no more than 12% even if the resident memory pages of each benchmark were placed in the worst possible arrangement. We tracked this effect further by measuring the slowdown of all the benchmarks in the last half iterations of the main computational loop. The average slowdown was measured to 1.03, while in 4 out of 6 cases the slowdown was less than 1.01. Observe that in all cases the user-level page migration algorithm with the worst-case initial page placement outperforms the plain IRIX first-touch page placement.
The trends observed with round-robin page placement are identical to those observed with the worst-case page placement, with the exception that the slowdown factor with round-robin is modest, namely 1.27 on average.

5.3 Performance with First-Touch Page Placement

Using dynamic page migration in the IRIX kernel improves execution times compared to first-touch (`ft-IRIX` compared to `ft-IRIXmig`) by 6% on average and at most by 17%, for SP on 32 processors. User-level page migration with the competitive page migration algorithm improves the execution time of the benchmarks on average by 19% compared to `ft-IRIX` and by 11% compared to `ft-IRIXmig`. The maximum gain is obtained for BT on 32 processors (28%). The improvements stem from optimally placing pages with more than one sharers, with respect to the reference pattern of one iteration of each benchmark. We consider the improvements as sizeable, considering that the machine on which we experimented provided enough cache space to reduce drastically the total number of L2 cache misses in the benchmarks and the ratio of remote to local memory access latency for the system was less than 3:1, so the relative penalty of remote memory accesses was not as severe as in other realizations of NUMA architectures. We would expect more significant improvements by using larger problem sizes, larger system sizes, or systems with less efficient communication infrastructure compared to that of the Origin2000.

5.4 Detailed Analysis

Figure 3 illustrates two histograms of the number of remote accesses issued to two heavily referenced arrays (u and rhs) in the NAS BT and SP benchmarks, executed with the user-level page migration system and first-touch page placement

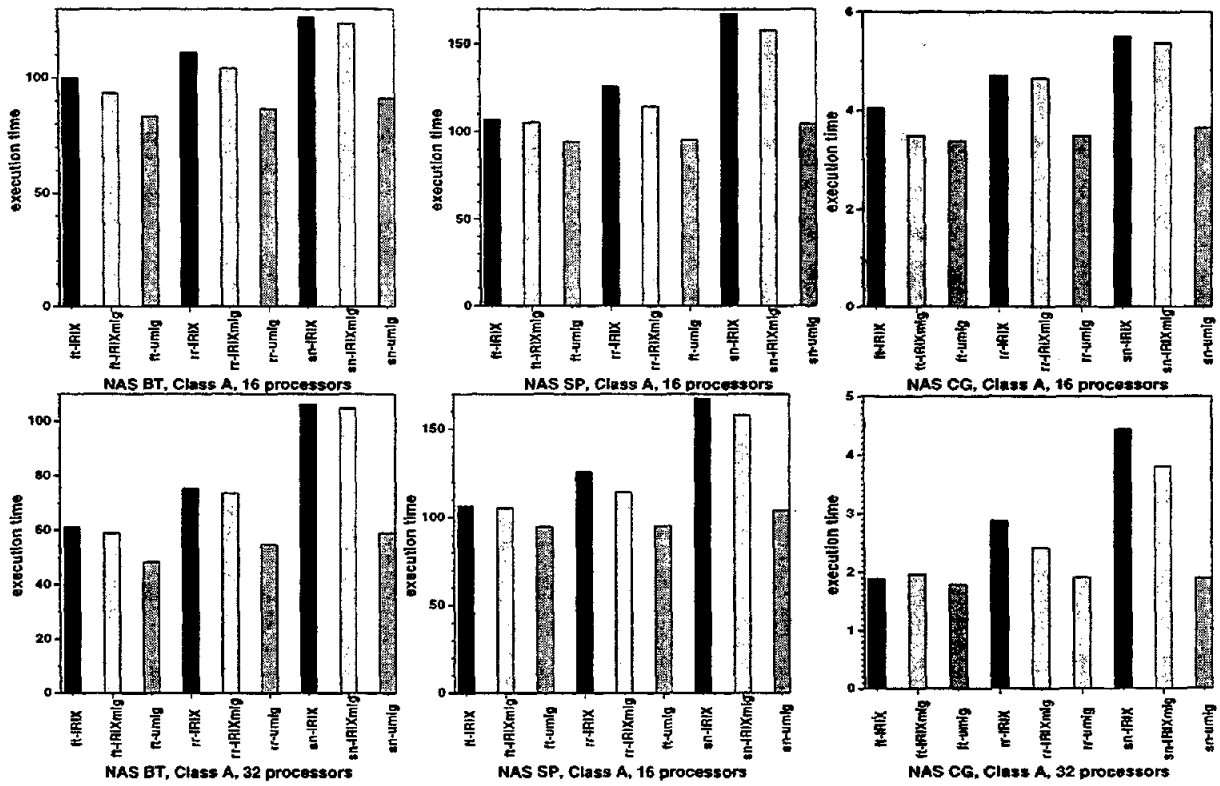

Figure 2: Experiments with page placements and page migration for the NAS BT, SP and CG benchmarks running on an idle system. All execution times are in seconds.

on 32 processors. Each bar in the histograms corresponds to the total number of remote accesses to an array, during an interval of 20 iterations for BT and 40 iterations for SP. The values were collected by recording the reference counters before each page migration performed by the runtime system and polling the counters every 20 and 40 iterations respectively. The same experiment could not be repeated with the IRIX page migration engine because we could not intercept the page migrations performed by the IRIX kernel at runtime. The histograms illustrate how user-level page migration reduces the number and hence the latency of remote memory accesses as the computation evolves. We note that in both cases the user-level page migration engine reduces drastically the number of remote accesses after the first invocations of the runtime system and that the benchmarks reach very soon an optimal execution point at which the number of remote accesses is practically minimized.

Table 1 gives some more statistics for the two page migration engines. The statistics in the table were collected from executions of BT and SP on 32 processors. BT and SP presented the greatest challenge for the page migration systems as their memory reference counts indicated that with first-touch page placement at least 1000 pages were always misplaced with respect to the optimal placement. In CG the corresponding number was between 10 and 20.

We instrumented the page migration calls in our runtime system to collect statistics. We used the `nstats -r` command to collect post-mortem statistics for the IRIX page migration engine. The table from left to right reports: the size of the resident set of each benchmark in pages; the number of pages in the hot arrays identified by the OpenMP compiler; the number of pages migrated on average by the IRIX kernel over the total number of pages that were considered eligible for migration; the same ratio for our user-level page migration engine with the competitive migration algorithm; and the percentage of page migrations that were performed by our runtime system in the first two iterations of the benchmarks. The latter information could not be obtained for IRIX, again because we were not able to intercept page migrations performed by the IRIX kernel during the execution of the benchmarks.

The statistics show that the user-level page migration runtime system migrates about one order of magnitude more pages compared to IRIX, although both the IRIX kernel and the runtime system identify approximately the same number of pages as candidates for migration. IRIX migrates only 24% of the pages considered eligible for migration. The rest of the pages are not migrated due to the multitude of constraints in the IRIX page migration engine, including the dampening, bouncing, and memory pressure filters (cf. Section 2). The runtime system migrates on average 84% of the pages considered eligible for migration. This is a good indication of the high accuracy of our page migration mechanism and the effectiveness of ping-pong prevention. Only 16% of the pages selected for migration were frozen due to actual ping-pong (i.e. migrated more than twice) by our runtime system. Witness that with the worst-case page placement the runtime system migrates 90% and 85% of the hot pages for BT and SP respectively. Considering that the benchmark executed on 16 nodes of the Origin2000 and 93.75% of the hot pages should migrate, it appears that the runtime system comes very close to the optimal point. In BT, the

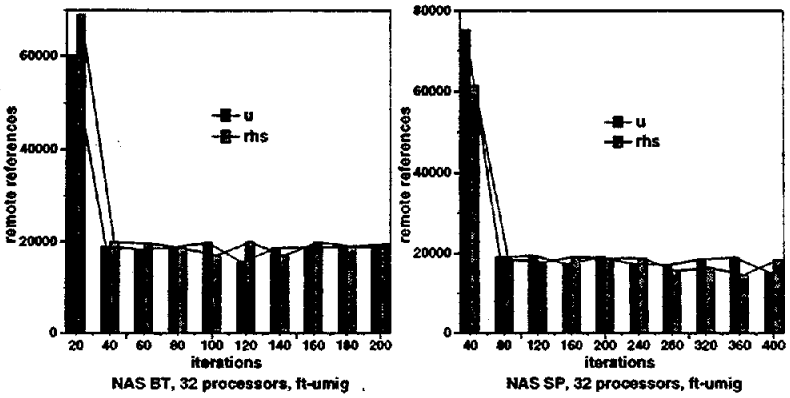

Figure 3: Histogram of remote accesses to two heavily referenced shared arrays in NAS BT and SP.

Table 1: Page migration statistics for executions of BT and SP on 32 processors.

Benchmark	Res. set	Hot pages	IRIX migr.	User-Level Migr.	% Migr. in first 2 iter.
BT, first-touch	3107	2688	255/1052 (24%)	1002/1233 (81%)	83%
BT, single-node	3107	2688	466/2607 (18%)	2405/2759 (87%)	73%
SP, first-touch	3055	2688	258/1010 (25%)	818/1014 (81%)	88%
SP, single-node	3055	2688	743/2670 (27%)	2260/2641 (86%)	78%

runtime system identifies more pages than the total number of hot pages as candidate for migration. This is a consequence of excessive ping-pong of some pages between more than two nodes, which was not detected by our ping-pong prevention algorithm.

The page migration overhead is not a major consideration since the implementation overlaps this overhead with useful computation. We measured with microbenchmarks that the amortized cost per page migration was approximately 1.1 ms on the system on which we experimented and we used this time as an estimation of migration latency in our page migration criteria. The machine on which we experimented did not utilize the Block Transfer Engine and the lazy TLB shootdown algorithm used in Origin2000 systems to reduce the cost of page migrations [15]. Although the high latency of migrations is overlapped by the runtime system, there is an implicit cost associated with page migrations, since page migration requests trigger operations such as TLB flushes, which interrupt the processors while executing application code. However, even with this restriction, the implicit cost of page migrations is well amortized over time by the runtime system. Our statistics show that 73%–88% of the page migrations are executed in the first 2 iterations of the benchmarks.

5.5 Experiments with Multiprogrammed Workloads

We conducted a second set of experiments to evaluate the effectiveness of user-level dynamic page migration in a multiprogrammed execution environment. In these experiments we executed three workloads with the NAS BT and SP benchmarks. The first workload included two copies of BT, the second workload included two copies of SP, and the third workload included one copy of BT and one copy of SP. We call the workloads `wl-1`, `wl-2` and `wl-3` respectively. We executed the workloads using first-touch page placement and the IRIX page migration enabled (`ft-IRIXmig`) and then first-touch page placement and our predictive page migration mechanism enabled (`ft-predmig`).

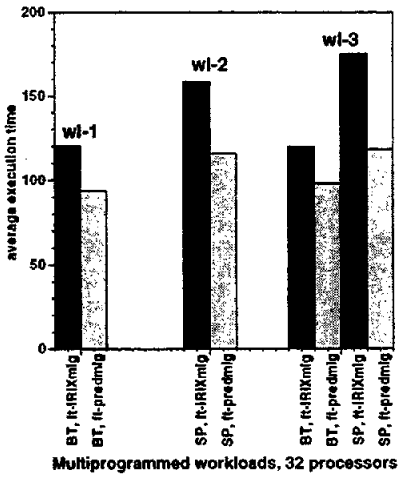

Figure 4: Execution time of BT and SP in the multiprogrammed workloads.

All benchmarks in the workloads requested 32 processors to execute. The experiments were conducted on a 32-processor partition of the SGI Origin2000, using the `miser` scheduler to establish an isolated 32-processor set [22]. Each program in the workload started executing with 32 threads therefore each processor in the partition was initially time-shared between two threads. In the course of execution the programs adjusted their number of running threads according to the multiprogramming load and the observed processor utilization. Automatic thread adjustment was accomplished by setting the `OMP_SET_DYNAMIC` variable to true, thus letting the IRIX kernel take control of the number of threads that execute parallel constructs in coordination with the IRIX MP parallelization library, which serves as the SGI OpenMP compiler backend [22].

Figure 4 illustrates the average execution times of BT and

SP in the three workloads, obtained from 5 independent executions of each workload in which the variance was less than 10%. The predictive dynamic user-level page migration mechanism reduces the execution times in all three workloads by 28% to 49%. The improvement originates mainly from the acceleration of the iterations executed after the benchmarks adjusted their number of threads, due to migrations triggered by the predictive page migration algorithm, as soon as the runtime system detected that the number of running threads in the benchmark has changed and some threads were migrated. Our experiment merely demonstrates the ability of our runtime system to react to scheduling events that may degrade application's performance in multiprogrammed environments. Further experimentation with multiprogrammed workloads is required to extract more accurate conclusions.

6. RELATED WORK

Dynamic page placement triggered by TLB misses was studied for NUMA multiprocessors with and without hardware cache coherence [3; 6; 18]. The related work motivated the introduction of first-touch, as the page placement policy of choice in multiprocessors with distributed shared memory. Dynamic page migration based on reference counters was introduced in [2] and its first implementation in a real operating system was presented in [25]. Our work extends these previous works towards the direction of making dynamic page migration more accurate, flexible and precise in time. We also differentiate from previous works with respect to the perception of dynamic page migration. Instead of using dynamic page migration as merely an optimization, we consider it as the means to introduce self-adaptability in parallel programs that execute in dynamic environments, given that in practically all production systems optimal execution conditions are hard to guarantee for parallel programs and this problem is likely to become worse in the near future. A third issue that differentiates our work is the study of dynamic page migration with the popular OpenMP standard, which demonstrates that page migration can tackle what is considered to be the major drawback of OpenMP i.e. the negligence of memory locality.

The integration of runtime support for improving data locality on cache-coherent NUMA multiprocessors has been explored in the COOL language [7]. COOL included affinity hints for associating threads with data and dynamically migrating data close to the threads for which they have affinity. COOL was primarily relying on programmer's knowledge for distributing tasks and data in a manner that exploits efficiently the memory hierarchy. Our infrastructure is transparent and relies on the compiler for driving the page migration engine. Although our first prototype does not utilize truly advanced compiler techniques to improve data locality at runtime, our approach is convenient, since it preserves the simplicity of the OpenMP programming standard and avoids exporting the subtle details of data distribution to the programmer.

The idea of exploiting the iterative nature of parallel applications and performance feedback mechanisms for applying runtime techniques that self-tune performance was exploited in several contexts such as dynamic processor allocation for multiprogrammed multiprocessors [19] and adaptive synchronization techniques [9]. Our work applies the idea of runtime performance monitoring and self-tuning in the context of page migration. A distinctive feature of our work is that runtime performance monitoring is applied to tune both the performance of the program and the parameters of the page migration algorithms, that is, we use dynamic performance feedback to implement two levels of adaptability in the runtime system.

7. CONCLUSIONS AND FUTURE WORK

This paper presented the design, algorithms, implementation details and a preliminary evaluation of a runtime system for user-level dynamic page migration. Our runtime system provides an infrastructure which enhances the memory performance of OpenMP programs on NUMA multiprocessors and arms the programs with immunity to the anomalies of the underlying execution environment with respect to data placement and locality. We analyzed how user-level page migration policies with compile time and runtime information can improve the accuracy and the timeliness of page migrations. We also presented several new approaches for solving critical problems related to page migration, including new cost-based competitive and predictive page migration algorithms and a ping-pong prevention scheme. Our results have exemplified that OpenMP programs executed with dynamic user-level page migration have indistinguishable performance with best-case and worst-case initial page placement strategies, without using explicit data placement directives. Furthermore, our runtime system provided sizeable improvements over the IRIX 6.5.5 memory placement and migration mechanisms, for single parallel benchmarks and multiprogrammed workloads.

Several of the issues outlined in this paper offer opportunities for further investigation. Our current prototype exploits the compiler solely to activate reference counting for shared arrays and instrument the programs. However, the runtime system is designed to exploit more advanced compiler knowledge and hints for further improving the accuracy of page migrations. For example, the compiler could annotate the program at specific points of execution at which the program exhibits a phase change in the memory access pattern and would benefit from activating an aggressive page migration mechanism. Mechanisms for migrating pages between phase changes in the reference patterns merit further investigation, since they can potentially obviate the need for data redistribution directives. Sensitivity analysis of our page migration algorithms is required to extract conclusions on the relative importance of parameters such as latency, contention and number of remote accesses. More experimentation with multiprogrammed workloads is also required to investigate the interaction between our runtime system and different job scheduling strategies. User-level page migration for non-iterative as well as irregular applications is an open issue as well. Finally, porting our infrastructure to other NUMA hardware platforms, including non tightly-coupled clusters with software shared memory and investigating the associated tradeoffs between page and thread migration are within our current plans.

Acknowledgments

This work was supported by the European Commission, through the TMR Contract ERBFMGECT-950062 and in part through the ESPRIT IV Project No. 21907 (NANOS). The experiments were conducted with resources provided by

the European Center for Parallelism of Barcelona (CEPBA). The authors would like to gratefully acknowledge the help of Xavier Martorell and the CEPBA systems support staff.

8. REFERENCES

[1] E. Ayguadé et al. NanosCompiler: A Research Platform for OpenMP Extensions. In *Proc. of the First European Workshop on OpenMP*, pages 27–31, October 1999.

[2] D. Black and D. Sleator. Competitive Algorithms for Replication and Migration Problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[3] W. Bolosky, M. Scott, R. Fitzgerald, and A. Cox. NUMA Policies and their Relationship to Memory Architecture. In *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, 1991.

[4] T. Brewer and G. Astfalk. The Evolution of the HP/Convex Exemplar. In *Proc. of the COMPCON Spring'97 Conference*, pages 81–96, Februrary 1997.

[5] L. Brieger. HPF to OpenMP on the Origin2000: A Case Study. In *Proc. of the First European Workshop on OpenMP*, pages 19–20, October 1999.

[6] 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*, pages 12–24, October 1994.

[7] R. Chandra, A. Gupta, and J. Hennessy. COOL: An Object-Based Language for Parallel Programming. *IEEE Computer*, 27(8):13–26, 1994.

[8] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, 1998.

[9] P. Diniz and M. Rinard. Eliminating Synchronization Overhead in Automatically Parallelized Programs Using Dynamic Feedback. *ACM Transactions on Computer Systems*, 17(2):89–132, 1999.

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

[11] C. Hristea, D. Lenoski, and J. Keen. Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks. In *Proc. of Supercomputing'97*, November 1997.

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

[13] 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.

[14] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proc. of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[15] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc. of the 24th International Symposium on Computer Architecture*, pages 171–181, June 1997.

[16] J. Levesque. The Future of OpenMP on IBM SMP Systems. In *Proc. of the First European Workshop on OpenMP*, pages 5–6, October 1999.

[17] T. Lovett and R. Clapp. STiNG : A CC-NUMA Computer System for the Commercial Marketplace. In *Proc. of the 23rd International Symposium on Computer Architecture*, pages 308–317, May 1996.

[18] 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. In *Proc. of the 9th International Parallel Processing Symposium*, April 1995.

[19] T. Nguyen, R. Vaswani, and J. Zahorjan. Maximizing Speedup Through Self-Tuning of Processor Allocation. In *Proc. of the 10th International Parallel Processing Symposium*, April 1996.

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

[21] M. Resch and B. Sander. A Comparison of OpenMP and MPI for the Parallel CFD Test Case. In *Proc. of the First European Workshop on OpenMP*, October 1999.

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

[23] Silicon Graphics Inc. *Origin2000 and Onyx2 Performance Tuning and Optimization Guide*. http://techpubs.sgi.com, 1999.

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

[25] 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*, pages 279–289, October 1996.