Is Data Distribution Necessary in OpenMP ?

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

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

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

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

Abstract

This paper investigates the performance implications of data placement in OpenMP programs running on modern ccNUMA multiprocessors. Data locality and minimization of the rate of remote memory accesses are critical for sustaining high performance on these systems. We show that due to the low remote-to-local memory access latency ratio of state-of-the-art ccNUMA architectures, reasonably balanced page placement schemes, such as round-robin or random distribution of pages incur modest performance losses. We also show that performance leaks stemming from suboptimal page placement schemes can be remedied with a smart user-level page migration engine. The main body of the paper describes how the OpenMP runtime environment can use page migration for implementing implicit data distribution and redistribution schemes without programmer intervention. Our experimental results support the effectiveness of these mechanisms and provide a proof of concept that there is no need to introduce data distribution directives in OpenMP and warrant the portability of the programming model.

1. Introduction

The OpenMP application programming interface [1] provides a simple and flexible means for programming parallel applications on shared memory multiprocessors. OpenMP has recently attracted major interest from both the industry and the academia, due to two strong inherent advantages, namely portability and simplicity.

OpenMP is portable across a wide range of shared memory platforms, including small-scale SMP servers, scalable ccNUMA multiprocessors and recently, clusters of workstations and SMPs [1, 2]. The OpenMP API uses a directive-based programming paradigm. The programmer annotates sequential code with directives that enclose blocks of code that can be executed in parallel. The programmer does not need to worry about subtle details of the underlying architecture and the operating system, such as the implementation of shared memory, the threads subsystem or the internals of the operating system scheduler. These details are entirely hidden from the programmer. OpenMP offers an intuitive, incremental approach for developing parallel programs. Users can begin with an optimized sequential version of their code and start adding manually or semi-automatically parallelization directives, up to a point at which they get the desired performance benefits from parallelism.

OpenMP follows the fork/join execution model. An OpenMP `PARALLEL` directive triggers the creation of a group of threads destined to execute in parallel the code enclosed between the `PARALLEL` and the corresponding `END PARALLEL` clause. This computation can be divided among the threads of the group via two worksharing constructs, denoted by the `DO` and `SECTIONS` directives. The `DO-END DO` directives encapsulate parallel loops, the iterations of which are scheduled on different processors according to a scheduling scheme defined in the `SCHEDULE` clause. The `SECTIONS-END SECTIONS` directives encapsulate disjoint blocks of code delimited by `SECTION` directives, which are assigned to individual threads for parallel execution. The group of threads that participate in the

0-7803-9802-5/2000/$10.00 © 2000 IEEE.

execution of a PARALLEL region is transparently scheduled on multiple physical processors by the operating system.

1.1. OpenMP and data distribution

OpenMP has recently become a subject of criticism because the simplicity of the programming model is often traded for performance. It is generally difficult to scale an OpenMP program to tens or hundreds of processors. Some researchers have pin-pointed this effect as a problem of the overhead of managing parallelism in OpenMP, which includes thread creation and synchronization. This overhead is an important performance limitation because it determines the *critical task size*, that is, the finest thread granularity that obtains speedup with parallel execution.

Although one could argue that the overhead of managing parallelism is a problem of the shared memory programming paradigm in general, it has been shown that programs parallelized for shared memory architectures can achieve satisfactory scaling up to a few hundreds of processors [3, 4]. This is possible with reasonable scaling of the problem size to increase the granularity of threads and reduce the frequency of synchronization. Nevertheless, scaling the performance of shared memory programs on a large number of processors requires also some ad-hoc programmer interventions, the most important of which is proper distribution of data among processing nodes. Data distribution is required to maximize the locality of references to main memory. This optimization is of vital importance on modern ccNUMA architectures, in which remote memory accesses can increase memory latency by factors of three to five.

Effective data distribution is what we consider to be the main performance optimization for OpenMP programs on contemporary ccNUMA multiprocessors. Briefly speaking, each page in the memory address space of a program should be placed on the same node with the threads that tend to access the page more frequently upon cache misses. Unfortunately, the OpenMP API provides no means to the programmer for controlling the distribution of data among processing nodes. It is interesting to note that some vendors of commercial ccNUMA systems have realized the importance of data distribution and implemented HPF-like, platform-specific data distribution mechanisms in their FORTRAN and C compilers [5]. Since OpenMP has become the de facto standard for parallel programming on shared memory multiprocessors, some vendors are seriously considering the incorporation of data distribution facilities in the OpenMP API [6, 7].

The introduction of data distribution directives in OpenMP contradicts some fundamental design goals of the OpenMP programming interface. Data distribution is inherently platform-dependent and thus hard to standardize and incorporate seamlessly in shared memory programming models like OpenMP. It is more likely that each vendor will propose and implement its own set of data distribution directives, customized to specific features of the in-house architecture, such as the topology, the number of processors per node, the available intra and internode bandwidth, intricacies of the system software and so on. Furthermore, data distribution directives will be essentially dead code for non-NUMA architectures such as desktop SMPs, a fact which raises an issue of code portability. Finally, data distribution has always been a burden for programmers. A programmer would not opt for a parallel programming model based on shared memory, if its programming requirements are similar to those of a programming model based on message passing.

1.2. Contributions of the paper

This first question that this paper comes to answer is up to what extent can data distribution affect the performance of OpenMP programs. To answer this question, we conduct a thorough investigation of alternative data placement schemes in the OpenMP implementations of the NAS benchmarks on the SGI Origin2000 [8]. These implementations are tuned specifically for the Origin2000 memory hierarchy and obtain maximum performance with the first-touch page placement strategy of the Origin2000. Assuming that first-touch is the "optimal,, data distribution scheme for the OpenMP implementations of the NAS benchmarks, we assess the performance impact of three alternative data distribution schemes, namely round-robin, random and worst-case page placement, which coincides with the page placement performed by a buddy system.

Our findings suggest that data distribution can indeed have a significant impact on the performance of OpenMP programs, although this impact is not as pronounced as expected for reasonably balanced distributions of pages among processors, like round-robin and random distribution. This result stems primarily from technological factors, since state-of-the-art ccNUMA systems such as the SGI Origin2000 have very low remote-to-local memory access latency ratios [9].

Since data distribution can have a significant impact on performance, the next question that rises naturally is how can data distribution be incorporated in OpenMP without modifying the application programming interface. The second contribution of this paper is a user-level framework for transparently injecting data distribution capabilities in OpenMP programs. The framework is based on dynamic page migration, a technique which has its roots in the earlier dance-hall shared memory architectures without hardware cache-coherence [10, 11, 12]. The idea is to track the reference rates from each node to each page in memory and

move each page to the node that references the page more frequently. Read-only pages can be replicated in multiple nodes. Page migration and replication are the direct analogue to multiprocessor cache coherence with the virtual memory page serving as the coherence unit.

Page migration was proposed merely as a kernel-level mechanism for improving the data locality of applications with dynamic memory reference patterns, initially on non cache coherent and later on cache coherent NUMA multiprocessors [12, 13]. In this work, we apply dynamic page migration in an entirely new context, namely data distribution. In this context, page migration is no longer considered as an optimization. It is rather used as the mechanism for approximating implicitly the functionality of a simple data distribution system.

The key for leveraging dynamic page migration as a data distribution vehicle is the exploitation of the iterative structure of most parallel codes, in conjunction with information provided by the compiler. We show that at least in the case of popular codes like the NAS benchmarks, a smart page migration engine can be as effective as a system that performs accurate initial data distribution, without losses in performance. Data redistribution across phases with uniform communication patterns can also be approximated transparently to the programmer by the page migration engine. The runtime overhead of page migration needs to be carefully amortized in this case, since it may easily outweigh the earnings from reducing the number of remote memory accesses. This problem would occur in any data distribution system, therefore we do not consider it as a major limitation.

To the best of our knowledge, the techniques presented in this paper for approximating data distribution and redistribution via dynamic page migration are novel. A second novelty is the implementation of these techniques entirely at user-level, with the use of only a few operating system services. The user-level implementation not only enables the exploration of parameters for the page migration policies, but also makes our infrastructure directly available to the user community.

The remainder of this paper is organized as follows: we present results that exemplify the degree of sensitivity of OpenMP programs to alternative page placement schemes in Section 2. We then describe briefly our user-level page migration engine in Section 3. Section 4 contains detailed experimental results. We overview related work in Section 5 and conclude in Section 6.

2. Sensitivity of OpenMP to page placement

Modern ccNUMA multiprocessors are characterized by their deep memory hierarchies. These hierarchies include typically four levels, namely the L1 cache, the L2 cache, the local node memory and the remote node memory. The memory hierarchy is logically expanded further if the remote node memory is classified according to the distance in hops between the accessing processor and the accessed node.

Table 1. Access latency to the different levels of the Origin2000 memory hierarchy.

Level	Distance in hops	Contented latency (ns.)
L1 cache	0	5.5
L2 cache	0	56.9
local memory	0	329
remote memory	1	564
remote memory	2	759
remote memory	3	862

Table 1 shows the base contented memory access latency by one processor to the different levels of the Origin2000 memory hierarchy on a 16-node system [14]. The nodes of the Origin2000 are organized in a fat hypercube topology with two nodes on each edge. The difference in the access latency between the L1 and the L2 caches is one order of magnitude. The difference between the access latency of the L2 cache and local memory accounts for another order of magnitude. For each additional hop that the memory accesses traverses, the memory latency is increased by 100 to 200 ns. The ratio of remote to local memory access latency ranges between 2:1 and 3:1.

The non-uniformity of memory access latency demands locality optimizations along the complete memory hierarchy of ccNUMA systems. These optimizations should take into account not only cache locality, but also locality of references to main memory. The latter can be achieved if the virtual memory pages used by a parallel program are mapped to physical memory frames so that each thread is more likely to access local rather than remote memory upon a miss in the L2 cache.

Page placement in ccNUMA systems is considered as a task of the operating system and previous research came up with simple solutions for achieving satisfactory data locality at the page level with page placement schemes implemented entirely in the operating system [15, 16]. However, the memory access traces of parallel programs do not and can not always conform to the memory management strategy of the operating system. The problem is pronounced in OpenMP because the programming model is oblivious to the distribution of data in the system. This section investigates the performance impact of theoretically inopportune page placement schemes on the performance of the NAS benchmarks.

2.1. Experimental setup

We used the OpenMP implementations of five benchmarks, namely BT, SP, CG, MG and FT, from the NAS benchmarks suite [8]. BT and SP are simulated CFD applications. Their main computational part solves Navier-Stokes equations and the programs differ in the factorization method used in the solvers. CG, MG and FT are computational kernels from real applications. CG approximates the smallest eigenvalue of a large sparse matrix using the conjugate-gradient method. MG computes the solution of a 3-D Poisson equation, using a V-cycle multigrid method. FT computes a 3-D Fast Fourier Transform. All codes are iterative and repeat the same parallel computation for a number of iterations corresponding to time steps. The implementations are well-tuned by the providers to exploit the characteristics of the memory system of the SGI Origin2000 and exhibit very good scalability up to 32 processors [8].

The OpenMP implementations of the NAS benchmarks are optimized to achieve good data locality with a first-touch page placement strategy [16]. This strategy places each virtual memory page in the same node with the processor that reads or writes it first during the execution of the program. First-touch is the default page placement scheme used by cellular IRIX, the Origin2000 operating system. The NAS benchmarks are customized to first-touch, by executing a cold-start iteration of the complete parallel computation before the main time-stepping loop. The calculations of the cold-start iteration are discarded, but the executed parallel constructs enable the distribution of pages between nodes with the first-touch strategy.

We conducted the following experiment to assess the impact of different page placement schemes. Assuming that first-touch is the best page placement strategy for the benchmarks, we ran the codes using three alternative page placement schemes, namely round-robin, random and worst-case page placement.

Round-robin page placement can be activated by setting the `_DSM_PLACEMENT` variable of the IRIX runtime environment. To emulate random page placement, we utilized the user-level page placement and migration capabilities of IRIX [17]. IRIX enables the user to virtualize the physical memory of the system and use a namespace for placing virtual memory pages to specific nodes in the system. The namespace is composed of entities called Memory Locality Domains (MLDs). A MLD is the abstract representation of the physical memory of a node in the system. The user can associate one MLD with each node and then place or migrate pages between MLDs to implement application-specific memory management schemes.

Random page placement is emulated as follows. Before executing the cold-start iteration, we invalidate the pages of all the shared arrays by calling `mprotect()` [1] with the `PROT_NONE` parameter. We install a `SIGSEGV` signal handler to override the default handling of memory access violations in the system. Upon receiving a segmentation violation fault for a page, the handler maps the page to a randomly selected node in the system, using the corresponding MLD as a handle. For benchmarks with resident set size in the order of a few thousand pages [2], a simple random generator is sufficient to produce a fairly balanced distribution of pages.

The worst-case page placement is emulated by enabling first-touch page placement and forcing the cold-start iteration of the parallel computation to run on one processor. With this modification, all virtual pages of the arrays which are accessed during the parallel computation are placed on a single node. This placement maximizes the number of remote memory accesses. Assuming a uniform distribution of secondary cache misses among processors and a system with n nodes, a fraction of secondary cache misses equal to $\frac{n-1}{n}$ is satisfied from remote memory modules. For a system with 8 nodes this amounts to 87.5% of the memory accesses and for a system with 16 nodes to 93.75% of the memory accesses. A second important, albeit implicit, effect of placing all the pages on one node is the maximization of contention. All processors except the ones on the node that hosts the data are contending to access the memory modules of one node throughout the execution of the program.

Note that the worst-case page placement scheme described previously is not totally unrealistic. On the contrary, it corresponds to the allocation performed by a buddy system which would allocate the pages with a best-fit strategy on a node with sufficient free memory resources. Many existing compilers make use of this memory allocation scheme.

The IRIX kernel includes a competitive page migration engine which can be activated on a per-program basis [9] by setting the `_DSM_MIGRATION` environment variable. We use this option in the experiments and compare the results obtained with and without the page migration engine. This is done primarily to investigate if the IRIX page migration engine is capable of improving the performance of page placement schemes inferior to first-touch. The implementation of page migration in IRIX follows closely the scheme presented in [13] for the Stanford FLASH multiprocessor. Each physical memory frame is equipped with a set of 11-bit hardware counters. Each set of counters contains one counter per node in the system and some additional logic to compare counters. The counters track the number of accesses from each node to each page frame in

[1] `mprotect` is the UNIX system call for controlling access rights to memory pages.

[2] We used the Class A problem sizes in the experiments.

memory. The additional circuitry detects when the number of accesses from a remote node exceeds the number of accesses from the node that hosts the page by more than a predefined threshold and delivers an interrupt in that case. The interrupt handler runs a page migration policy, which evaluates if migrating the page that caused the interrupt satisfies a set of resource management constraints. If the constraints are satisfied the page is migrated to the more frequently accessing node and the TLB entries with the mappings of the page are invalidated with interprocessor interrupts. After moving the page, the operating system updates its mappings of the page internally. The valid TLB entries for the page are reloaded upon TLB misses by processors that reference the page after its migration.

2.2. Results

Figure 1 shows the results from executing the OpenMP implementations of the NAS benchmarks on 16 idle processors of an SGI Origin2000. The system on which we experimented had MIPS R10000 processors with a clock frequency of 250 Mhz, 32 Kbytes of split L1 cache per processor, 4 Mbytes of unified L2 cache per processor and 8 Gbytes of memory, uniformly distributed between the nodes of the system. Each bar in the charts is an average of three independent experiments with insignificant variation. All execution times are in seconds. The black bars illustrate the execution time with the different page placement schemes, labeled `ft-`, `rr-`, `rand-` and `wc-`, for first-touch, round-robin, random, and worst-case page placement respectively. The gray bars illustrate the execution time with the same page placement schemes and the IRIX page migration engine enabled during the execution of the benchmarks (labeled `ft-IRIXmig`, `rr-IRIXmig`, `rand-IRIXmig` and `wc-IRIXmig` respectively). The straight line in each chart shows the baseline performance with the native first-touch page placement scheme of IRIX.

The primary observation from the results is that using a page placement scheme other than first-touch does have an impact on performance, although the magnitude of this impact is non-uniform across different benchmarks and page placement schemes. In general, worst-case page placement incurs a significant slowdown (50%–248%) for all benchmarks except BT, in which the slowdown is modest (24%). The average slowdown with worst-case page placement is 90%. On the other hand, round-robin and random page placement have generally a modest impact. Round-robin incurs little slowdown in SP and CG (8% and 11% respectively), and modest slowdown in the rest of the benchmarks (22%–35%). Random page placement incurs almost no slowdown for BT and SP (2% and 12% respectively), modest slowdown for CG and MG (26% and 27%) and significant slowdown only for FT (45%).

In general, balanced page placement schemes such as round-robin and random appear to affect modestly the performance of the benchmarks, compared to the best static page placement scheme. This is attributed to the low ratio of remote to local memory access latency on the SGI Origin2000, which is no more than 2:1 on the 16-processor scale. This important architectural property of the Origin2000 shows up in the experiments. A second reason is that any balanced page placement scheme, such as round-robin and random can be effective in distributing evenly the message traffic incurred from remote memory accesses in the interconnection network.

The results show that the IRIX page migration engine has in general negligible impact on performance with first-touch page placement. Activating dynamic page migration in the IRIX kernel provides only marginal gains of 3% for CG and less than 2% for BT, SP and MG. Page migration is harmful for FT because it introduces false-sharing at the page level. With the other three page placement schemes, dynamic page migration generally improves performance, with only a few exceptions (BT with random page placement and CG with round-robin page placement). In three cases, BT with round-robin and SP with round-robin and random placement, the IRIX page migration engine is able to approximate the performance of first-touch. Notice however that these are the cases in which the static page placement schemes perform competitively to first-touch and the performance losses are less than 12%. Dynamic page migration from the IRIX kernel is unable to close the performance gap between first-touch and the other page placement schemes in the cases in which the difference is significant. Round-robin, random and worst-case page placement schemes still incur a sizeable average slowdown (16%, 17% and 61% respectively). Only in one case, MG with worst-case page placement, the IRIX page migration engine is able to improve performance drastically, without approaching however the performance of first-touch.

To summarize, the page placement scheme can be harmful for programs parallelized with OpenMP. However, any reasonably balanced page placement scheme makes the performance impact of mediocre page-level locality modest. In our experiments, this is possible due to the aggressive hardware and software optimizations of the SGI Origin2000, which reduce the remote to local memory access latency ratio to 2:1. It is also enabled by the reduction of contention achieved by balanced page placement schemes. The impact of page placement would be more significant on ccNUMA architectures with higher remote memory access latencies. It would be also more significant on truly large-scale Origin2000 systems (e.g. with 128 processors or more), in which some remote memory accesses would have to cross up to 5 interconnection network hops and then a meta-router to reach the destination node. Unfortunately,

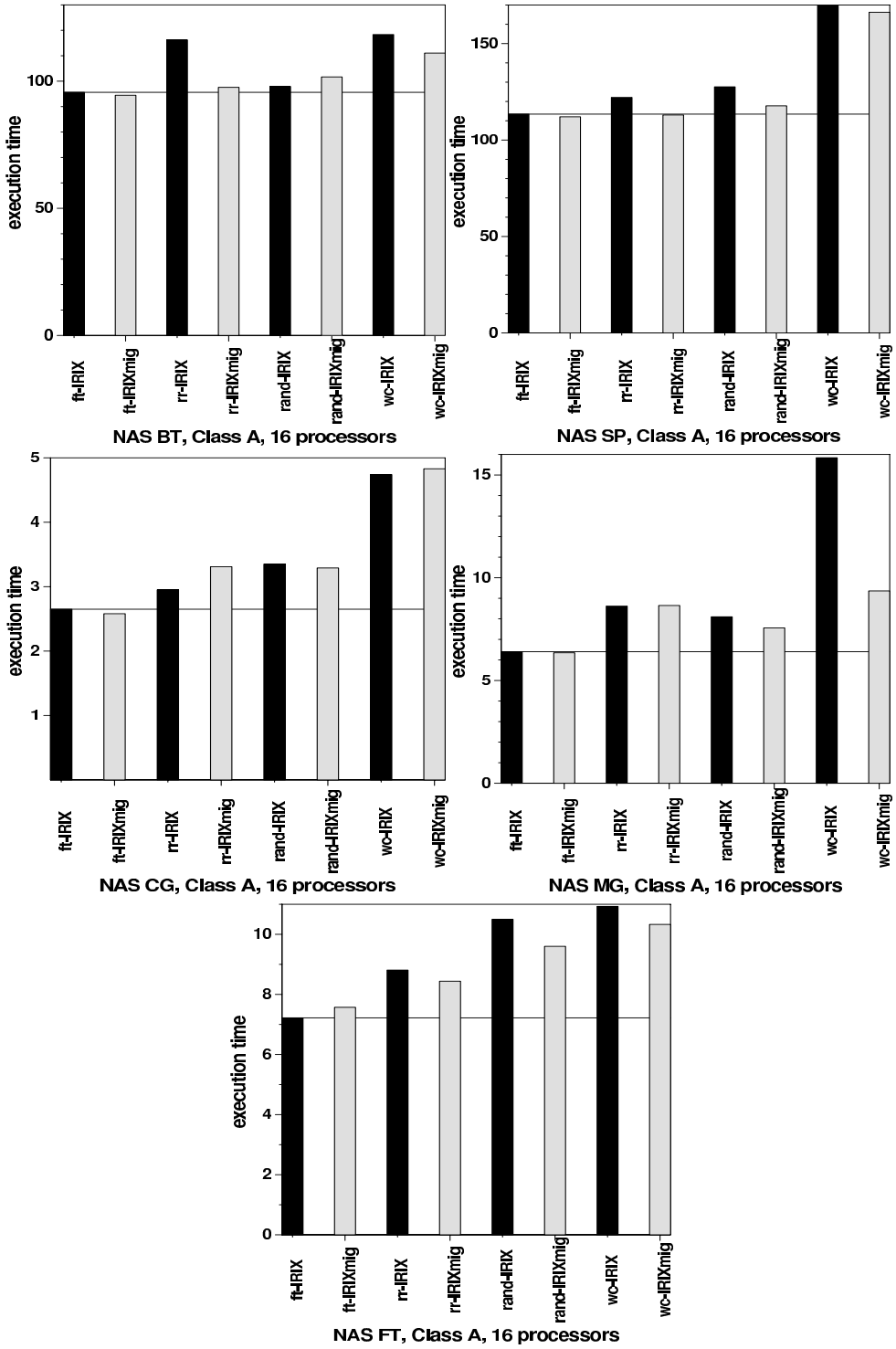

Figure 1. Impact of page placement on the performance of the OpenMP implementations of the NAS benchmarks.

access to a system of that scale was impossible for our experiments.

3. Using dynamic page migration in place of data distribution

The position of this paper is that dynamic page migration can transparently alleviate the problems introduced from poor page placement in OpenMP. In particular, we investigate the possibility of using dynamic page migration as a substitute for data distribution and redistribution in OpenMP programs. Intuitively, this approach has the advantages of transparency and seamless integration with OpenMP, because dynamic page migration is a runtime technique and the associated mechanisms reside in the system software. The questions that remain to be answered is how can page migration emulate or approximate the functionality of data distribution and if this is feasible, what is the level of performance achieved by a data distribution mechanism based on dynamic page migration.

3.1. User-level dynamic page migration

To investigate the issues stated before, we have developed a runtime system called *UPMlib* (user-level page migration library), which injects a dynamic page migration engine to OpenMP programs, through instrumentation performed by the compiler. A pure user-level implementation was possible, because the operating system services needed to implement a page migration policy in IRIX are available at user-level.

The hardware counters attached to the physical memory frames of the Origin2000 can be accessed via the `/proc` interface. At the same time, MLDs enable the migration of ranges of the virtual address space between nodes in the system. These two services allow for a straightforward implementation of a runtime system which can act in place of the operating system memory manager in a local scope. The only subtle detail is that the page migration service offered at user-level is subject to the resource management constraints of the operating system. Briefly speaking, a user-requested page migration may be rejected by the operating system due to shortage of available memory in the target node. IRIX uses a best-effort strategy in this case and forwards the page to another node as physically close as possible to the target node. This restriction is necessary to ensure the stability of the system in the presence of multiple users competing for shared resources. Implementation details of our runtime system are given elsewhere [18].

Our earlier work on page migration identified the ineffectiveness of previously proposed kernel-level page migration engines as a problem of poor timeliness and accuracy [19]. A page migration mechanism should migrate pages early enough to reduce the rate of remote memory accesses while amortizing effectively the high cost of coherent page movements. Furthermore, the page migration decisions should be based on accurate page reference information and not biased by transient effects in the parallel computation. If page migration is to be used as a means for data distribution, timeliness and accuracy are paramount.

In the same work [19], we have shown that an effective technique for accurate and effective dynamic page migration stems from exploiting the iterative structure of most parallel codes. If the code repeats the same parallel computation for a number of iterations, the page migration engine can record the exact reference trace of the program as reflected in the hardware counters after the end of the first iteration and subsequently use this trace to make nearly optimal decisions for migrating pages. This strategy works extremely well in codes with fairly coarse-grain computations and access patterns. The infrastructure requires limited support by the compiler for identifying areas of the virtual address space which are likely to concentrate remote memory accesses and instrumenting the program to invoke the page migration engine. The compiler identifies as *hot* memory areas the shared arrays which are both read and written in disjoint sets of OpenMP `PARALLEL DO` and `PARALLEL SECTION` constructs.

3.2. Emulating data distribution

In this paper we show how the technique of recording reference traces at well-defined execution points can be applied in a page migration engine to approximate accurately and effectively the functionality of manual data distribution and redistribution in iterative parallel codes.

The mechanism for emulating data distribution is straightforward to implement. Assume any initial placement of pages. The runtime system records the memory reference trace of the parallel program after the execution of the first iteration. This trace indicates accurately which processor accesses each page more frequently, while the structure of the program ensures that the same reference trace will be repeated throughout the execution of the program, unless the operating system intervenes and preempts or migrates threads[3]. The trace of the first iteration can be used to migrate each page to the node that will minimize the maximum latency due to remote memory accesses to this page, by applying a competitive page migration criterion after the execution of the first iteration. Page migration is used in place of static data distribution with a hysteresis of one iteration. The necessary migrations of pages are performed early and their cost is amortized well over the entire execution time. The fundamental difference with an explicit data

[3] This case is not considered in this paper, but the reader can refer to a related paper [20] for details.

```
...
call upmlib_init()
call upmlib_memrefcnt(u, size)
call upmlib_memrefcnt(rhs,size)
call upmlib_memrefcnt(forcing,size)
...
do step=1,niter
    call compute_rhs
    call x_solve
    call y_solve
    call z_solve
    call add
    if ((step .eq. 1) .or.
      (num_migrations .gt. 0)) then
      call upmlib_migrate_memory()
    endif
enddo
```

Figure 2. Using page migration for data distribution in NAS BT.

distribution mechanism is that data placement is performed with implicit information encapsulated in the runtime system, rather than with explicit information provided by the user.

In the actual implementation, the page migration mechanism is invoked not only in the first, but also in subsequent iterations of the parallel program, as soon as it detects at least one page to migrate. The mechanism is self-deactivated the first time it detects that no further page migrations are required. In practice, this happens usually in the second iteration, however there are some cases in which page-level false sharing might incur some excessive page migrations. This is circumvented by freezing the pages that bounce between two nodes in consecutive iterations.

Figure 2 provides an example of using the previously described mechanism in NAS BT. Calls to the page migration runtime system are prefixed by `upmlib_`. The OpenMP compiler identifies three arrays (`u`,`rhs` and `forcing`) as hot memory areas and activates page reference monitoring for these areas by invoking the `upmlib_memrefcnt(addr,size)` function. After the execution of the first iteration, the program calls `upmlib_migrate_memory()`, which scans the reference counters of the pages that belong to hot memory areas, applies a competitive page migration for each page and migrates those pages that concentrate enough remote memory accesses to satisfy the migration criterion. The variable `num_migrations` stores the number of page migrations executed by the mechanism in the last invocation of `upmlib_migrate_memory()` and deactivates the mechanism when set to 0.

3.3. Emulating data redistribution

Emulating data redistribution with dynamic page migration is a more elaborate procedure. In general, data redistribution is needed when a phase change in the memory reference pattern distorts the data locality established by the initial page placement scheme. Data redistribution needs some additional compiler support to identify *phases*. A simple definition of a phase, which conforms also to the OpenMP programming paradigm, is a sequence of basic blocks of parallel code with a uniform communication pattern between processors. Not all communication patterns are recognizable by a compiler. However, simple cases like one-to-one, nearest neighbor, broadcast and all-to-all can be relatively easily identified.

We use a technique known as record-replay in order to use our page migration engine as a substitute for data redistribution. The compiler instruments the program to record the page reference counters at the points of execution at which phase transitions occur. The recording is performed during the first iteration of the parallel program. After the recording procedure is completed, each phase is associated with two sets of hardware counters, one recorded before the beginning of the phase and one before the transition to the next phase.

For each page in the hot memory areas, the runtime system obtains the reference trace during the phase in isolation, by comparing the values of the counters attached to the page in the two recorded sets of the phase. The runtime system applies the competitive page migration criterion using the isolated reference trace of the phase and decides what pages should be moved before the transition to the phase, to improve data locality during the phase. The page migrations identified with this procedure are replayed in subsequent iterations. Each page migration is replayed before the phase during which the page satisfied the competitive criterion in the first iteration. The recording procedure is not used for the transition from the last phase of the first iteration to the first phase of the second iteration. In this case, the runtime system simply *undoes* all page migrations performed between phases and recovers the initial page placement.

More formally, assume that a program has n hot pages, denoted as $p_i, i = 1 \ldots n$. Assume also that one iteration of the program has k distinct phases. There are $k-1$ phase transition points, $j = 1 \ldots k-1$. The runtime system is invoked in each transition point and records for each page a vector of page reference counters $V_{i,j}, \forall i, j, i = 1 \ldots n, j = 1 \ldots k-1$. After the execution of the first iteration, the runtime system computes the difference $U_{i,j} = V_{i,j} - V_{i,j-1} \forall i, j, i = 1 \ldots n, j = 2 \ldots k-1$. The runtime system applies the competitive criterion using the values of the counters stored in $U_{i,j}$. If the counters in $U_{i,j}$ satisfy the competitive criterion, page p_i is migrated

in every subsequent iteration at the phase transition point $j-1$. For each page p_i that migrates at some transition point for the first time during an iteration, the home node of the page before the migration is recorded, in order to migrate the page back to it before the beginning of the next iteration.

The record-replay mechanism is accurate in the sense that page migration decisions are based on complete information on the reference trace of the program. However, the mechanism is also sensitive to the overhead of page migration. In the record-replay mechanism, page migrations must be performed on the critical path of the execution. Let $T_{nom,j}$ be the execution time of a phase j without page migration before the transition to the phase and $T_{m,j}$ be the execution of the same phase with the record-replay mechanism enabled. Let $O_{m,j}$ be the overhead of the page migrations performed before the transition to phase j by the record-replay mechanism. It is expected that $T_{m,j} < T_{nom,j}$ due to the reduction of remote memory accesses achieved by the page migration engine. The record-replay mechanism should satisfy the condition $\sum_{j=1}^{k}(T_{m,j}+O_{m,j}) < \sum_{j=1}^{k} T_{nom,j}$. In practice, this means that each phase should be computationally coarse enough to balance the cost of migrating pages with the earnings from reducing memory latency.

To limit the cost of page migrations in the record-replay mechanism, we use an environment variable which instructs the mechanism to move only the n *most critical* pages, in each iteration, where n is a tunable parameter. The n most critical pages are determined as follows: the pages are sorted in descending order according to the ratio $\frac{racc_{max}}{lacc}$, where *lacc* is the number of accesses from the node that hosts the page and $racc_{max}$ is the maximum number of remote accesses from any of the other nodes. The pages that satisfy the inequality $\frac{racc_{max}}{lacc} > thr$, where thr is a predefined threshold, are considered as eligible for migration. Let m be the number of these pages. If $m > n$, the n pages with the highest ratios $\frac{racc_{max}}{lacc}$ are migrated. Otherwise, the m candidate pages are migrated.

Figure 3 provides an example of using the record-replay mechanism in conjunction with the mechanism described in Section 3.2 in NAS BT. BT has a phase change in the `z_solve` function, due to the initial alignment of arrays in memory, which is performed to improve locality along the `x` and `y` directions. After the first iteration, `upmlib_migrate_memory()` is called to approximate the best initial data distribution scheme. In the second iteration, the program invokes `upmlib_record()` before and after the execution of `z_solve`. The function `upmlib_compare_counters()` is used to identify the reference trace of the phase and the pages that should migrate before the transition to the phase. These migrations are replayed by calling `upmlib_replay()` before each

```
...
call upmlib_init()
call upmlib_memrefcnt(u, size)
call upmlib_memrefcnt(rhs,size)
call upmlib_memrefcnt(forcing,size)
...
do step=1,niter
    call compute_rhs
    call x_solve
    call y_solve
    if (step .eq. 2) then
      call upmlib_record()
    else if (step .gt. 2) then
      call upmlib_replay()
    endif
    call z_solve
    if (step .eq. 1) then
        call upmlib_migrate_memory()
    else if (step .eq. 2) then
      call upmlib_record()
      call upmlib_compare_counters()
    else
      call upmlib_undo()
    endif
enddo
```

Figure 3. Using the record-replay mechanism for data redistribution in NAS BT.

subsequent execution of `z_solve`. The function `upmlib_undo()` performs all the replayed page migrations in the opposite direction.

4. Experimental results

We repeated the experiments presented in Section 2, after instrumenting the NAS benchmarks to use the page migration mechanisms of our runtime system.

In the first set of experiments presented in this section, we evaluate the ability of our page migration engine to relocate pages early in the execution of the program, in order to approximate the best possible initial data distribution scheme. Figure 4 repeats the results from Figure 1 and in addition, illustrates the performance of the iterative page migration mechanism of our runtime system, with the four different page placement schemes (labeled `ft-upmlib`, `rr-upmlib`, `rand-upmlib` and `wc-upmlib`).

A first observation is that with first-touch page placement, in all cases expect CG, user-level page migration provides sizeable reductions of execution time (6%–22%), compared to the native codes with or without page migration from the IRIX kernel. For the purposes of this paper, we consider this result as a second-order effect, attributed to the suboptimal placement of several pages in

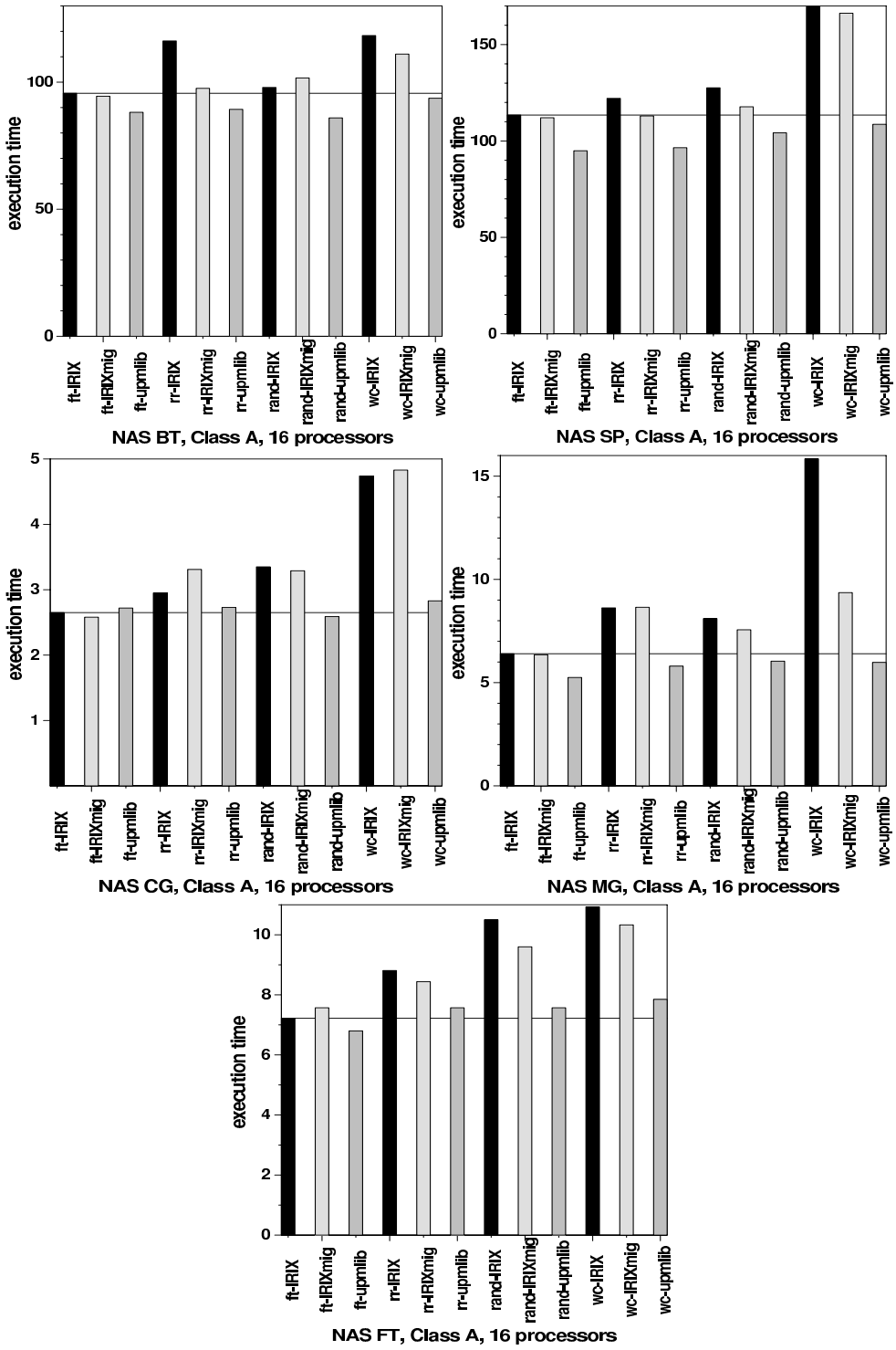

Figure 4. Performance of our page migration runtime system with different page placement schemes.

each benchmark by the first-touch strategy. We note however that this is probably the first experiment on a real system which shows some meaningful performance improvements achieved by dynamic page migration over the best static page placement scheme.

The outcome of interest from the results in Figure 4 is that with non-optimal page placement schemes, the slowdown compared to first-touch is almost imperceptible. When the page migration engine of our runtime system is enabled, the slowdown compared to first-touch is on average 5% for round-robin, 6% for random and 14% for worst-case page placement. In the experiments presented in Section 2 the average slowdowns incurred from round-robin, random and worst-case page placement without page migration were 22%, 23% and 90% respectively. The slowdowns of the same page placement schemes with page migration enabled in the IRIX kernel were 16%, 17% and 61% respectively.

Table 2 provides some additional statistics which were collected by manually inserting event counters in the runtime system. The second, third and fourth columns of the table report the slowdown of the benchmarks in the last 75% of the iterations of the main parallel computation for round-robin, random and worst-case page placement respectively[4]. This slowdown was always measured less than 2.7%, while in most cases it was less than 1%. The results indicate that the page migration engine achieves robust and stable memory performance as the iterative computations evolve.

The fifth, sixth and seventh column of Table 2 show the fraction of page migrations performed by our page migration engine after the first iteration of the parallel computation. In three out of five cases, CG, FT and MG, all page migrations were performed after the first iteration of the program. In the case of BT and SP some page-level false sharing forced page migrations after the second and third iterations. However, 78% or more of the migrations were performed after the first iteration. This result verifies that the page migration activity and the associated overhead are concentrated at the beginning of the execution of the programs and are amortized well over the execution lifetime.

The overall results show that due to the effectiveness of the iterative page migration mechanism described in Section 3.2, the performance of the OpenMP implementations of the NAS benchmarks is not sensitive to the initial page placement scheme. Equivalently, the iterative page migration mechanism can approximate closely the performance achieved by the best static page placement scheme and therefore be effectively used as a substitute for data distribution.

[4]The fraction 75% was somewhat arbitrarily selected, because MG has only 4 iterations. The number of iterations for BT,CG,FT and SP are 200,400,6 and 15 respectively.

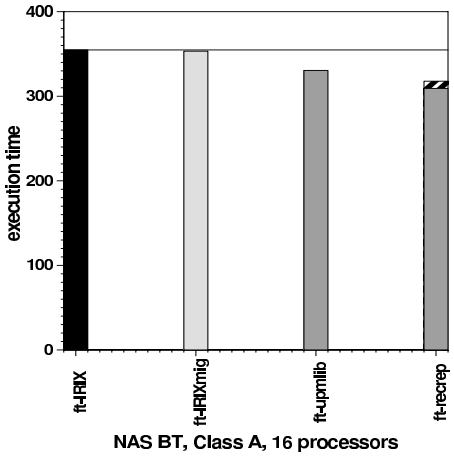

Figure 6. Performance of the record-replay mechanism in the synthetic experiment with NAS BT.

We conducted a third set of experiments, in which we evaluated the record-replay mechanism. In these experiments, we instrumented BT and SP to use record-replay in order to deal with the phase change the `z_solve` function, as shown in Figure 3.

Figure 5 illustrates the performance of the record-replay mechanism with first-touch page placement and the page migration mechanism for data distribution enabled only in the first iteration. This scheme is labeled `ft-recrep` in the charts. The striped part of the `ft-recrep` bar shows the non-overlapped overhead of the page migrations performed by the record-replay mechanism. In these experiments we set the number of critical pages to 20, in order to limit the cost of replaying page migrations at phase transition points. For the sake of comparison, the figure shows also the execution time of BT and SP with first-touch with/without the IRIX page migration engine, as well as the execution time with our page migration engine enabled only for data distribution.

The results show that the record-replay mechanism achieves some speedup in the execution of useful computation, marginal in the case of SP, up to 10% in the case of BT. Unfortunately, the overhead of page migrations performed by the record-replay mechanism seems to outweigh this speedup. When looking at these experiments, one should bear in mind that the specific architectural characteristics of the Origin2000 bias significantly the results. More specifically, the low remote-to-local memory access latency ratio of the system and the high overhead of page migration due to the maintenance of TLB coherence, limit the gains from reducing the rate of remote memory accesses.

In order to overcome the aforementioned implications, we attempted to synthetically scale the experiment and in-

Table 2. Statistics from the execution of the NAS benchmarks with different page placement schemes and our page migration engine.

Benchmark	Slowdown in the last 75% of the iterations			% of Migrations in the first iteration		
	round-robin	random	worst-case	round-robin	random	worst case
BT	0.3%	0.2%	0.9%	87%	82%	93%
CG	1.1%	1.0%	2.7%	100%	100%	100%
FT	0.5%	0.4%	0.8%	100%	100%	100%
MG	0.5%	0.6%	0.5%	100%	100%	100%
SP	0.8%	0.9%	1.4%	78%	81%	88%

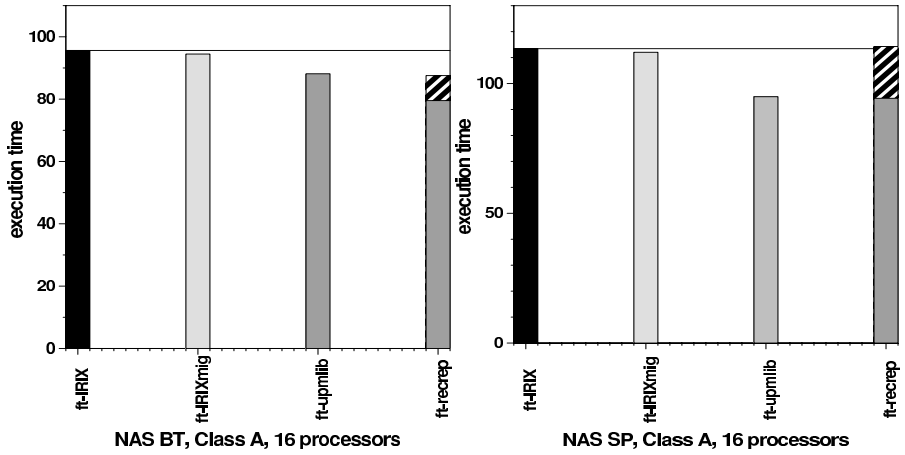

Figure 5. Performance of the record-replay mechanism in NAS BT and SP

.

crease the amount of computation performed during each phase in the benchmarks. The purpose was to enable the record-replay mechanism to amortize the cost of page migrations over a longer period of time. We did this modification without changing the memory access pattern and the locality characteristics of the benchmarks as follows: we enclosed each function that comprises the main body of the parallel in a sequential loop with 4 iterations. In this way, we were able to expand the parallel execution time of `z_solve` from 130 ms to approximately 520 ms on 16 processors. What we expected to see in this experiment was a much lower relative cost of page migration and some earnings from activating the record-replay mechanism between phases. The results from this experiment with NAS BT (shown in Figure 6) verify our intuition. The overhead of page migrations accounts for a small fraction of the execution time and the reduction of remote memory accesses shows up, since it is exploited over a longer time period. In this experiment the record-replay mechanism provides an improvement of 5% over the version of the benchmark that uses page migration only for data distribution.

5. Related work

The idea of dynamic page migration has been employed since the appearance of the first commercial NUMA architectures more than a decade ago. Aside from several important theoretical foundations on page migration [21, 22] mechanisms for automatic page migration by the operating system have been implemented in systems like the BBN Butterfly Plus and the IBM RP3 [11, 12]. These systems had no hardware-supported cache coherence and the cost of shared memory accesses was solely determined by the location of pages in shared memory. Different schemes were investigated, such as migrating a page on every write by a different processor, migration based on complete reference information, or migration based on incomplete reference information collected by the operating system. Applying fairly aggressive page migration and replication strategies on these systems was feasible, because the relative cost of page migrations was not so high compared to the cost of memory accesses. The effectiveness of dynamic page migrations in these schemes varied radically and it was af-

fected significantly by the subtleties of the architecture and the underlying operating system.

With the appearance of cache coherent NUMA multiprocessors, dynamic page migration became a trickier problem. On the ccNUMA architecture, accesses to shared data go through the caches and the memory performance of parallel programs depends heavily on cache locality. In the first detailed study of the related issues, Verghese et.al. [13] have shown that it is necessary to collect accurate page reference information in order to implement an effective dynamic page migration scheme. Partial information like TLB misses is not sufficient. The same work proposed a complete kernel-level implementation of dynamic page migration and evaluated it using accurate machine-level simulation of a ccNUMA multiprocessor. The results have shown that dynamic page migration can improve the response time of programs with irregular memory access patterns, as well as the throughput on a mulitprogrammed system. The page migration engine of the Origin2000 is largely based on this work, however it has not been able to achieve the same level of performance improvements thus far [4, 19]. The previous work on dynamic page migration investigated in general the potential of the technique as a locality optimizer. In our work dynamic page migration is placed in a new context and used as a tool for implicit data distribution in OpenMP.

This paper is among the first to conduct a comprehensive evaluation of static page placement schemes on an actual ccNUMA multiprocessor. Static page placement schemes for cache coherent NUMA multiprocessors were investigated via simulation in [16, 23]. The study of Marchetti et.al. [16] identified first-touch as the most effective static page placement scheme for simple parallel codes. Bhuyan et.al. [23] have recently explored using simulation the impact of several page placement schemes in conjunction with alternative interconnection network switch designs on the performance of parallel applications on ccNUMA multiprocessors. Their study is oriented towards identifying how can better switch designs improve the performance of suboptimal page placement schemes that incur contention. The study provides also some useful insight on the relative performance of three of the page placement schemes evaluated in this paper, namely first-touch, round-robin and buddy allocation. The quantitative results of the study of Bhuyan et.al. and ours resemble in the sense that non-optimal page placement schemes perform often quite close to first-touch under certain architectural circumstances. Some quantitative assessment of page placement schemes appeared also in papers that evaluated the performance of the SPLASH-2 benchmarks on ccNUMA multiprocessors [3, 4], however these studies focused on analyzing the locality characteristics of the specific programs.

The idea of recording shared memory accesses and use the recorded information to implement on-the-fly locality optimizations was exploited in the tapes mechanism [24]. This mechanism is designed for software distributed shared memory systems, in which all accesses to shared memory are handled by the runtime system. The tapes mechanism is used as a tool to predict future consistency protocol actions which are likely to require communication between nodes. The domain in which the recording mechanism is applied in this paper is quite different. However, both the tapes mechanism and the record-replay mechanism presented in this paper exploit the iterative structure of parallel programs.

Data distribution is a widely and thoroughly studied concept, mainly in the context of data-parallel programming languages like HPF. A direct comparison between HPF and OpenMP is out of the scope of this paper. HPF is very expressive with respect to data distribution and providing a one-to-one correspondence between HPF functionalities and page migration mechanisms would be rather unrealistic. What this paper emphasizes, is that some data distribution capabilities which are critical for sustaining high performance on distributed shared memory multiprocessors can be replaced by dynamic page migration mechanisms.

6. Conclusion

The title of this paper raised the question if data distribution facilities should be introduced in OpenMP or not. The answer given to this dilemma by the experiments presented in this paper is no. This position is supported by two arguments. First, the hardware of state-of-the-art ccNUMA systems is aggressively optimized to reduce the remote-to-local memory access latency ratio so much, that any reasonably balanced page placement scheme used by the operating system is expected to perform within a small fraction of the optimum. This trend is expected to persist in future architectures, since all the related research is attacking the problem of minimizing remote memory accesses or reducing their cost. Second, in cases in which the page placement scheme is a critical performance factor, system software mechanisms like dynamic page migration can remedy the problem by relocating accurately and timely during the execution of the program the poorly placed pages. The synergy of architectural factors and advances in system software enables plain shared memory programming models like OpenMP to retain a competitive position in the user community by preserving the fundamental properties of programming with shared memory, namely simplicity and portability.

Acknowledgments

This work was supported by the E.C. through the TMR Contract No. ERBFMGECT-950062, the Greek Secretariat of Research and Technology (contract No. E.D.-99-566) and the Spanish Ministry of Education through projects No.

TIC98-511 and TIC97-1445CE. The experiments were conducted with resources provided by the European Center for Parallelism of Barcelona (CEPBA).

References

[1] OpenMP Architecture Review Board. *OpenMP Specifications*. http://www.openmp.org, accessed April 2000.

[2] H. Lu, Y. Charlie Hu, and W. Zwaenepoel. *OpenMP on Networks of Workstations*. Proc. of Supercomputing'98: High Performance Networking and Computing Conference. Orlando, FL, November 1998.

[3] C. Holt, J. Pal Singh and J. Henessy. *Application and Architectural Bottlenecks in Large Scale Shared Memory Machines*. Proc. of the 23rd Annual International Symposium on Computer Architecture, pp. 134–145. Philadelphia, PA, June 1996.

[4] D. Jiang and J. Pal 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.

[5] R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. Anderson. *Data Distribution Support for Distributed Shared Memory Multiprocessors*. Proc. of the 1997 ACM Conference on Programming Languages Design and Implementation, pp. 334–345. Las Vegas, NV, June 1997.

[6] J. Levesque. *The Future of OpenMP on IBM SMP Systems*. Invited Talk. First European Workshop on OpenMP, Lund, Sweden, October 1999.

[7] J. Harris. *Extending OpenMP for NUMA Architectures*. Invited Talk. Second European Workshop on OpenMP. Edinburgh, Scotland, October 2000.

[8] 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. October 1999.

[9] J. Laudon and D. Lenoski. *The SGI Origin: A ccNUMA Highly Scalable Server*. Proc. of the 24th International Symposium on Computer Architecture, pp. 241–251. Denver, CO, May 1997.

[10] BBN Advanced Computers Inc. *Inside the Butterfly Plus*. 1987.

[11] W. Bolosky, R. Fitzgerald and M. Scott. *Simple but Effective Techniques for NUMA Memory Management*. Proc. of the 12th ACM Symposium on Operating Systems Principles, pp. 19–31. Lichfield Park, AZ, December 1989.

[12] M. Holliday *Reference History, Page Size, and Migration Daemons in Local/Remote Architectures*. Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 104–112. Boston, MA, April 1989.

[13] 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.

[14] D. Culler, J. Pal Singh and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, August 1998.

[15] R. Chandra, S. Devine, A. Gupta and M. Rosenblum. *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.

[16] 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*. Proceedings of the 9th International Parallel Processing Symposium, pp. 480–485. Santa Barbara, CA, April 1995.

[17] SGI Inc. IRIX 6.5 Man Pages. `mmci` - Memory Management Control Interface. http://techpubs.sgi.com, accessed October 1999.

[18] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta and E. Ayguadé. *UPMlib: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared Memory Multiprocessors*. Proc. of the 5th ACM Workshop on Languages, Compilers and Runtime Systems for Scalable Computers. Rochester, NY, May 2000.

[19] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta and E. Ayguadé. *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.

[20] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta and E. Ayguadé. *User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors*. Proc. of the 29th International Conference on Parallel Processing. Toronto, Canada, August 2000.

[21] 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.

[22] D. Black, A. Gupta and W. Weber. *Competitive Management of Distributed Shared Memory*. Proc. of the IEEE COMPCON'89 Conference. San Francisco, CA, February 1989.

[23] L. Bhuyan, R. Iyer, H. Wang and A. Kumar. *Impact of CC-NUMA Memory Management Policies on the Application Performance of Multistage Switching Networks*. IEEE Transactions on Parallel and Distributed Systems, Vol. 11(3), pp. 230–245, March 2000.

[24] P. Keleher. *Tapeworm: High Level Abstractions of Shared Accesses*. Proc. of the 3rd Symposium on Operating Systems Design and Implementation, pp. 201–214. New Orleans, LA, February 1999.