A transparent runtime data distribution engine for OpenMP[1]

Dimitrios S. Nikolopoulos[a,*],
Theodore S. Papatheodorou[b],
Constantine D. Polychronopoulos[a],
Jesús Labarta[c] and Eduard Ayguadé[c]
[a]*Computer and Systems Research Laboratory, University of Illinois at Urbana-Champaign, 1308 West Main Street, Urbana, IL 61801, USA*
E-mail: {dsn,cdp}@csrd.uiuc.edu
[b]*Department of Computer Engineering and Informatics, University of Patras, GR26500, Patras, Greece*
E-mail: tsp@hpclab.ceid.upatras.gr
[c]*Department of Computer Architecture, Technical University of Catalonia, c/Jordi Girona 1-3, 08034, Barcelona, Spain*
E-mail: {jesus,eduard}@ac.upc.es

This paper makes two important contributions. First, the paper investigates the performance implications of data placement in OpenMP programs running on modern NUMA 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 contemporary NUMA architectures, reasonably balanced page placement schemes, such as round-robin or random distribution, incur modest performance losses. Second, the paper presents a transparent, user-level page migration engine with an ability to gain back any performance loss that stems from suboptimal placement of pages in iterative OpenMP programs. The main body of the paper describes how our OpenMP runtime environment uses page migration for implementing implicit data distribution and redistribution schemes without programmer intervention. Our experimental results verify the effectiveness of the proposed framework and provide a proof of concept that it is not necessary to introduce data distribution directives in OpenMP and warrant the simplicity or the portability of the programming model.

1. Introduction

Parallel processing is experiencing a convergence of both architectures and programming models into few well-established paradigms [10]. Parallel programming in particular seems to converge into two paradigms, namely message-passing and shared-memory. Message-passing is the programming model of choice for multiprocessors with distributed physical memory and disjoint address spaces, where the processors on a node[2] can access only the node's local memory. Shared-memory on the other hand, is the programming model of choice for single-address-space multiprocessors, in which cache coherence protocols or virtual memory mechanisms provide the programmer with the abstraction of a shared memory space, equally accessible by the processors.

The fundamental difference between message-passing and shared-memory is the means used by the programmer to express interprocessor communication. In message-passing, the programmer has to explicitly distribute data and computation among processors. It is the programmer's responsibility to implement the communication required to synchronize concurrent tasks and exchange data, via explicit send and receive statements. In the shared-memory programming paradigm, since all data lies in a single address space, processors communicate directly via loads and stores in shared memory. Parallelism is expressed by assigning code fragments to computational threads and inserting the synchronization commands needed to ensure correct execution. The scheduling of threads to processors is performed transparently to the programmer.

Both message-passing and shared-memory programming paradigms have been successfully standardized. Message-passing standards such as PVM and MPI have already reached a mature point of use [33]. More re-

*Corresponding author.

[1] An earlier version of this paper [29] appeared in the IEEE/ACM Supercompting'2000 Conference and won the Best Technical Paper Award.

[2] We use the general term *node* to denote the building block of a state-of-the-art multiprocessor. A node typically includes a few processors, memory and a communication assist.

Scientific Programming 8 (2000) 143–162
ISSN 1058-9244 / $8.00

cently, a joint effort of several hardware and software vendors resulted to the development of OpenMP [30], a flat shared-memory programming model which is implemented as an extension to FORTRAN, C and C++ and is portable across the whole spectrum of shared-memory multiprocessor architectures.

It is a common belief that shared-memory programming models are simpler and more intuitive to use compared to message-passing. This is true because the shared-memory paradigm hides the details of the architecture and the system support software from the programmer. This in turn enables the programmer to focus on the parallelism of the problem, rather than on the subtleties of the underlying hardware/software interface. On the flip side of the coin, it is also a fact that at medium and large processor scales, the level of performance attained by programs written with a shared-memory programming model is not nearly as good as the level of performance attained by their message-passing counterparts. This phenomenon is observed on non-uniform memory access (NUMA) multiprocessors, in which although the node memories are accessible by all processors, the actual memory access latency is non-uniform. A remote memory access costs several times as much as a local memory access.

Sustaining high performance on a NUMA architecture requires the localization of memory accesses. Briefly put, data should be placed in memory so that each processor accesses data from a local memory module (i.e., residing on the processor's node), whenever the processor misses in the cache.[3] Shared-memory programming paradigms in general lack the means to perform this task. The obvious way to accomplish that, is to extend the programming model with data distribution commands analogous to the commands used in data-parallel programming paradigms, such as HPF. Indeed, some researchers have shown that such an extension, along with some hand-crafted optimizations for reducing the cost of orchestrating parallelism, is sufficient to scale single-address-space programs to hundreds of processors [16,18].

It is interesting to note that vendors of commercial NUMA multiprocessors have realized the importance of data distribution and implemented HPF-like, platform-specific data distribution mechanisms, as extensions to FORTRAN and C [8]. Unfortunately, OpenMP, which is nowadays the de facto standard for programming shared-memory multiprocessors, provides no means for data distribution. As a consequence, some vendors are proposing the introduction of data distribution directives in OpenMP [4,21,23], as the way to achieve the desired levels of memory access locality in the programming model.

1.1. The problem

The thesis of this paper is that the introduction of data distribution directives contradicts the fundamental design principles of OpenMP and shared-memory programming models in general, by compromising their simplicity and portability.

The basic design principle behind OpenMP is that the details of the architecture and the operating system should be entirely hidden from the programmer and the same OpenMP program should be able to run without modifications on different shared-memory architectures manufactured by different vendors [30]. This principle is compromised to a significant extent with the introduction of data distribution directives. Data distribution is inherently platform-dependent and as such, hard to standardize and incorporate seamlessly in a programming model. 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 constitute dead code for non-NUMA architectures, a fact which raises an issue of code portability.

Regarding the ease of programming, data distribution has always been a burden for programmers. Data distribution commands are subtle and hard to understand and use. Defining flexible data distribution schemes to minimize communication is a daunting task, even for regular applications. A programmer is not likely to opt for a parallel programming model based on shared-memory, if the effort required to parallelize a code is similar to that of a programming model based on message-passing.

The problem addressed in this paper is how to introduce the functionality required to effectively distribute data in OpenMP, without any modification to the programming model. More specifically, the problem is to devise an automatic, transparent mechanism that performs the tasks of a manual data distribution tool, without exporting the actual data distribution interface to the programmer.

[3] Placing data for better utilization of the cache itself is also a prerequisite for efficient shared-memory parallel programming. However, the related issues are out of the scope of this paper.

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 on NUMA architectures. To answer this question, we conduct a thorough investigation of alternative data placement schemes in the OpenMP implementations of the NAS benchmarks [19] on the SGI Origin2000 [22]. These implementations are tuned specifically for the Origin2000 memory hierarchy and obtain maximum performance with the first-touch page placement strategy of IRIX, the Origin2000 operating system. Assuming that first-touch is the "optimal" page placement scheme for the OpenMP implementations of the NAS benchmarks, we assess the performance impact of three alternative, non-optimal schemes, namely round-robin, random and the worst-case page placement.

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 balanced distributions of pages among nodes. This result stems primarily from technological factors, since modern NUMA multiprocessors are equipped with communication networks and coherence protocols that achieve a very low remote-to-local memory access latency ratio [22].

The second contribution of this paper is a user-level runtime system that transparently injects data distribution capabilities in OpenMP programs. The system uses dynamic page migration, a technique with roots in the early dance-hall shared-memory architectures [1, 7,15]. The idea behind dynamic page migration 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 a direct analogue to multiprocessor cache coherence, with the virtual memory page serving as the coherence unit.

Page migration has been proposed merely as a kernel-level mechanism for improving the data locality of applications with dynamic memory reference patterns [15,34]. In this work, dynamic page migration is put in a radically different context. In particular, page migration is no longer considered as an optimization. It is rather used as the vehicle of a transparent data distribution engine.

The key for leveraging dynamic page migration as a data distribution technique is the exploitation of the iterative structure of the majority of 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 any performance loss. Data redistribution across phases with different memory access patterns can also be approximated transparently to the programmer, using our 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 dynamic 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 with dynamic page migration at runtime are novel. A second novelty is the implementation of these techniques, which is carried out entirely at user-level with the use of only a few operating system services. Our implementation not only enables the exploration of parameters in the page migration policies, but also makes our framework directly available to the community, without requiring any modifications to the architecture, the operating system, or the source code of OpenMP programs.

The remainder of this paper is organized as follows. A brief overview of OpenMP is given in Section 2. We present results that exemplify the sensitivity of OpenMP programs to data placement in Section 3. Our user-level page migration engine is outlined in Section 4. Section 5 provides detailed experimental results. We overview related work in Section 6 and conclude the paper in Section 7.

2. An overview of OpenMP

The OpenMP application programming interface (API) [30] provides a directive-based paradigm 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, simplicity and portability.

In OpenMP, parallelism is expressed with compiler directives that enclose loops or regions of code that can be executed in parallel. An OpenMP `PARALLEL` directive triggers the creation of a group of threads destined to execute concurrently the code enclosed between the `PARALLEL` and the corresponding `END PARALLEL`

directive. This computation can be divided among the threads that belong to the group via two worksharing constructs, expressed with the `DO` and `SECTIONS` directives. The `DO-END DO` pair of directives encapsulates parallel loops, the iterations of which are scheduled on different processors according to a scheduling scheme defined in the `SCHEDULE` clause of the directive. The `SECTIONS-END SECTIONS` pair of directives encapsulates disjoint blocks of code delimited by `SECTION` directives. Each block of code is assigned to a different thread for parallel execution.

OpenMP uses the fork/join execution model. An OpenMP program starts executing with one thread denoted as the master. Upon encountering a `PARALLEL` construct, the master creates a group of slave threads, to execute the code surrounded by the `PARALLEL-END PARALLEL` directives. The group of threads that participate in the execution of parallel loops and sections is transparently scheduled on multiple physical processors by the operating system. Upon completion of a parallel region, the master synchronizes with the slaves at an implicit barrier.

OpenMP programs are portable across the whole range of shared-memory architectures, including small-scale SMPs, scalable NUMA multiprocessors and clusters of workstations or SMPs [17,24,30]. The OpenMP API hides entirely the details of the architecture and the operating system from the programmer. The programmer does not need to worry about the actual implementation of shared memory (software or hardware), interprocessor communication, the location of data in shared memory, the topology of the system, the interface for creating, scheduling and synchronizing threads, or the internals of the operating system scheduler. These details are hidden behind the runtime backend of OpenMP. Therefore, 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.

2.1. OpenMP performance characteristics

OpenMP has recently become a subject of criticism because the simplicity of the programming model is often traded for performance. Experience with real codes suggests that it is difficult to scale an OpenMP program to tens or hundreds of processors and that complex OpenMP codes tend to scale worse than the corresponding MPI codes [2,13,31]. 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 minimum thread granularity that obtains speedup from parallel execution. One way to deal with this problem in OpenMP is to coarsen the granularity of parallel loops and sections, in order to minimize the frequency of expensive management operations, such as thread creation and synchronization [19]. A second performance limitation of OpenMP programs is the cache miss ratio. The layout of data in memory should be carefully optimized, in order to maximize the reuse of data blocks from the processor caches. Although sophisticated *cache-aware* programming is difficult, modern compiler technology provides several program transformations for exploiting various types of cache locality in shared-memory multiprocessors [35].

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

The overhead of managing parallelism and cache reuse are both important performance considerations. However, it has been shown that scaling the performance of shared-memory programs on a large number of processors requires some additional ad-hoc programmer interventions, the most important of which is proper distribution of data between nodes [16,18]. Data distribution is required to maximize the locality of references to main memory. This optimization is of vital importance on contemporary NUMA multiprocessors, in which remote memory accesses can increase memory latency by a factor of three to five. The OpenMP API provides no means for controlling the distribution and placement of data.

3. Sensitivity of OpenMP to data placement

Modern NUMA multiprocessors are characterized by their deep memory hierarchies. These hierarchies include at least four levels, namely the L1 cache, the L2 cache, local node memory and remote node memory.

The memory hierarchy can be logically subdivided further, if the remote node memory is classified according to the distance (in interconnection network hops) between the accessing processor and the accessed node.

Table 1 shows the contented memory access latencies to the memory hierarchy of the SGI Origin2000, a popular contemporary NUMA multiprocessor [10]. The numbers are taken from a 16-node system. The nodes of the Origin2000 are organized in a fat hypercube topology with two nodes on each edge. The L1 cache latency differs from the L2 cache latency by one order of magnitude. The difference between the latency of the L2 cache and local memory accounts for another order of magnitude. For each additional hop that a memory accesses traverses, the 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 NUMA 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 of the program are mapped to physical memory frames so that each thread is more likely to access local rather than remote memory upon L2 cache misses.

Page placement in NUMA systems is primarily a task of the operating system. Previous research came up with solutions for achieving satisfactory data locality at the page level, with simple page placement schemes implemented in the kernel [9,25]. However, the memory access traces of parallel programs do not and can not always conform to the page placement strategy of the operating system. The problem is pronounced in OpenMP, because the programming model is oblivious of the placement of data in memory. This section investigates the performance impact of theoretically inopportune page placement schemes on the performance of the NAS benchmarks.

3.1. Experimental setup

We used the OpenMP implementations of five of the NAS benchmarks, BT, SP, CG, MG and FT [19], to investigate the aforementioned issues. BT and SP are simulated CFD applications. Their main computational part solves Navier-Stokes equations in three dimensions. 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 specific implementations of the NAS benchmarks are well-tuned to exploit the characteristics of the memory system of the SGI Origin2000 and exhibit satisfactory scalability up to 32 processors [19].

The OpenMP implementations of the NAS benchmarks are optimized to achieve good data locality with the first-touch page placement scheme [25]. First-touch places each virtual memory page in the same node with the processor that reads or writes the page first during the course of execution. First-touch is the default page placement scheme used in IRIX. The NAS benchmarks conform 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 regions perform an implicit distribution of pages on a first-touch basis.

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 [32]. IRIX enables the user to virtualize the physical memory of the system and use a namespace for placing virtual memory pages to specific nodes. 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 an MLD with each node and 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 the shared arrays used in the benchmarks by calling `mprotect()`[4] with the `PROT_NONE` parame-

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

ter. We install a `SIGSEGV` signal handler to override the default handling of memory access violations in IRIX. Upon receiving a segmentation violation fault for a page, the handler maps the page to a randomly selected node, using the corresponding MLD as a handle. The NAS benchmarks with which we experimented have resident set sizes in the order of a few thousand pages,[5] therefore a simple random generator is sufficient to produce a well balanced distribution of pages.

The worst-case page placement is emulated by enabling first-touch and forcing the cold-start iteration of the parallel computation to run on one processor. With this modification, the virtual pages of the arrays 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 pages on one node is the exacerbation of contention. All processors except the ones on the node that hosts the data contend to access the memory modules of a single node throughout the execution of the program.

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

The IRIX kernel includes a competitive page migration engine which can be activated on a per-program basis [22] by setting the `_DSM_MIGRATION` environment variable. We use this option in the experiments and compare the results obtained with and without the IRIX 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 which are inferior to first-touch. The implementation of page migration in IRIX follows closely the design presented in [34] 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 and some additional logic to compare counters. The counters track the number of accesses from each node to each frame in 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. In that case, the hardware counters deliver an interrupt to a local processor. 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 of processors holding mappings of the page are invalidated with interprocessor interrupts. After moving the page, the operating system updates its internal mappings of the page. The valid TLB entries are reloaded upon TLB misses by the processors that reference the page after the migration.

[5] We used the class A problem sizes in the experiments.

3.2. Results

Figures 1 and 2 show the execution times of the OpenMP implementations of the NAS benchmarks on 16 and 32 idle processors of an SGI Origin2000, with different page placement strategies. The system on which we experimented has 64 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 uniformly distributed DRAM memory.

Each bar in the charts is the average of three independent experiments. Execution times are in seconds. The black bars illustrate the execution time with 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`). The straight line in the charts shows the performance of the native first-touch page placement mechanism of IRIX. The purpose of the experiment is merely to show the impact of data placement on performance. For the sake of completeness, we mention that the parallel efficiency (speedup divided by the number of processors) of the OpenMP implementations of the NAS benchmarks is equal to or greater than 68% on 16 and 32 processors, with the exception of MG which has an efficiency of slightly less than 40%.

Table 2
Minimum, maximum and average slowdown (percentage) of different page placement strategies, compared to first-touch

Strategy	16 procs.			32 procs.		
	min.	max.	avg.	min.	max.	avg.
round-robin	7.6	34.6	19.4	−4.8	48.6	25.0
random	2.4	45.4	22.6	−1.3	74.5	28.4
worst-case	23.7	147.5	70.2	62.4	110.9	81.3
round-robin + IRIX mig.	5.0	27.5	17.2	1.6	33.7	23.7
random + IRIX mig.	5.0	27.5	17.2	1.6	33.7	23.6
worst-case + IRIX mig.	17.5	87.2	47.3	38.9	66.6	57.3

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. The worst-case page placement incurs a significant slowdown, ranging from 50% to 148% on 16 processors and from 62% to 110% on 32 processors. The only exception is BT on 16 processors, where the slowdown is modest (24%). The average slowdown of the worst-case page placement is 70% on 16 processors and 81% on 32 processors (see Table 2). On the other hand, round-robin and random page placement have in most cases only a modest impact on performance. Round-robin incurs little slowdown in SP and CG (8% and 11% respectively) and modest slowdown in the rest of the benchmarks (22%–35%) on 16 processors. On 32 processors, CG enjoys a speedup of 5% with round-robin page placement, while the other benchmarks are slowed down modestly by 25% on average. Random page placement incurs almost no slowdown in BT and SP (2% and 12% respectively), modest slowdown in CG and MG (26% and 27%) and significant slowdown only in FT (45%) on 16 processors. On 32 processors, BT gains a speedup of 1.3% with random page placement, MG and SP have little slowdown (7% and 16%), while CG and FT are slowed down by more than 45%.

In general, balanced page placement schemes such as round-robin and random placement appear to affect modestly the performance of the benchmarks. This is attributed to the low ratio of remote to local memory access latency of the Origin2000. This important architectural property of the Origin2000 shows up in the experiments. A second reason is that balanced distributions of pages are highly effective in distributing evenly the message traffic incurred from remote memory accesses in the interconnection network. The experiments indicate that alleviating contention at the network interfaces appears to be a performance factor of increasing importance on modern NUMA architectures.

The use of the IRIX page migration engine has a negligible impact on performance with first-touch page placement. Activating dynamic page migration in the IRIX kernel provides only marginal speedups of 3% for CG and less than 2% for BT, SP and MG on 16 processors. On 32 processors, CG and FT are slowed down, while BT, SP and MG increase their execution speed by 9% to 15%. 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 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 very closely or even exceed slightly the performance of first-touch. Notice however that these are the cases in which the non-optimal static page placement schemes perform competitively to first-touch. 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 (more than 20%). Round-robin, random and worst-case page placement still incur a sizeable average slowdown (see Table 2).

To summarize, the page placement strategy can be harmful for programs parallelized with OpenMP. However, any reasonably balanced placement of pages makes the performance impact of mediocre memory access locality modest. In our experiments, this is possible due to the aggressive hardware and software optimizations of the Origin2000, which reduce the remote to local memory access latency ratio. It is also enabled by the reduction of contention achieved by balanced page placement schemes. The trends observed in the experiments are likely to hold for next-generation NUMA architectures, which include built-in hardware mechanisms for reducing both the number and the cost of remote memory accesses [11,14,26]. The impact of page placement would be more significant on NUMA

architectures with higher remote memory access latencies. It would be more significant also on truly large-scale NUMA systems (e.g. with more than 128 processors), in which some remote memory accesses would have to cross more than 5 interconnection network hops to reach the destination node. Unfortunately, access to a system of that scale was not possible for our experiments.

4. 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. 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 are hidden in system software. The question that remains to be answered is how can page migration emulate or approximate the functionality of a manual data distribution tool and if this is feasible, what is the level of performance achieved by a runtime data distribution mechanism based on dynamic page migration.

4.1. User-level dynamic page migration

We have developed a runtime system called *UPMlib* (user-level page migration library), which injects a dynamic page migration engine into OpenMP programs, through instrumentation performed by the compiler. The entire runtime system is implemented at user-level, using the IRIX memory management control interface (mmci) [32].

The hardware counters attached to the physical memory frames of the Origin2000 can be accessed with the `/proc` interface. At the same time, MLDs enable the migration of ranges of the virtual address space between nodes. These two services enable a straightforward implementation of a runtime system which acts 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. Simply put, a page migration requested at user-level may be rejected by the operating system, due to shortage of free memory in the target node. IRIX uses a best-effort strategy in this case and forwards the page to a 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 memory resources. Implementation details of our runtime system are given elsewhere [28].

Our earlier work on page migration identified the limited effectiveness of previously proposed kernel-level page migration engines, as a problem of poor timeliness and accuracy [27]. A dynamic page migration policy should migrate pages early enough to reduce the rate of remote memory accesses and at the same time, amortize 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.

We have shown that a technique for accurate and effective dynamic page migration stems from exploiting the iterative structure of most parallel codes [27]. 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 use this trace to make optimal decisions for migrating pages in subsequent iterations. This strategy works extremely well in codes with fairly coarse granularity and access pattern. The infrastructure requires limited support by the compiler, to identify areas of the virtual address space which are likely to concentrate remote memory accesses and instrument the program with invocations of 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 SECTIONS` constructs.

4.2. Emulating data distribution

In this section we show how recording reference traces at well-defined execution points can be applied in a page migration engine to approximate accurately and timely the functionality of manual data distribution 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 during the complete parallel computation, while the structure of the program ensures that the same reference trace will be repeated throughout the execution lifetime, unless the operating system intervenes and preempts or migrates threads.[6] The trace of the first iteration can be used to migrate each page to the node that minimizes the maximum latency due to remote memory accesses to the page, by applying a competitive migration criterion after the execution of the first iteration [27]. Page migration is used in place of static data distribution with a hysteresis of one iteration. The page migrations required are performed early and their cost is amortized well over the entire execution lifetime. The fundamental difference from an explicit data distribution tool, is that data placement is performed with implicit information encapsulated in the runtime system, rather than with explicit information provided by the programmer.

In the actual implementation, the page migration mechanism is invoked not only in the first, but also in subsequent iterations, as soon as the migration criterion detects pages to migrate. The mechanism is self-deactivated the first time it detects that no further 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 excessive page migrations. This is circumvented by freezing the pages that bounce between two or more nodes in consecutive iterations.

Figure 3 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()` function. After the execution of the first iteration, the program calls `upmlib_migrate_memory()`, which scans the reference counters of pages that belong to hot memory areas, applies a competitive page migration criterion for each page and migrates those pages that concentrate enough remote accesses to satisfy the criterion. The variable `num_migrations` stores the number of migrations executed by the mechanism in the last invocation of `upmlib_migrate_memory()` and deactivates the mechanism when set to 0.

[6] This case is not considered in this paper. The reader can refer to [29] for a treatment of the related issues.

```
...
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
...
call upmlib_end()
```

Fig. 3. Using page migration for data distribution in NAS BT.

4.3. Emulating data redistribution

Emulating data redistribution with dynamic page migration is more elaborate. Data redistribution is required when a *phase change* in the memory reference pattern distorts the memory access 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 also conforms well to the OpenMP programming paradigm, is a sequence of code blocks with a uniform interprocessor communication pattern. Not all communication patterns are recognizable by a compiler. However, simple cases like one-to-one or nearest neighbor can be relatively easily identified.

We use a technique called *record-replay*, in order to engage our page migration engine as a substitute for data redistribution. The compiler instruments the program to record the page reference counters at points of execution where 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 in the two recorded sets corresponding to 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, in order to improve memory access locality during the phase. Page migrations identified with this procedure are replayed in subsequent iterations. Each migration is replayed before the phase during which the associated 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 across phases and recovers the initial page placement.

More formally, assume that a program has n hot pages, $p_i, i = 1 \ldots n$. 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 at every 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 migration criterion using the values of the counters stored in $U_{i,j}$. If the counters in $U_{i,j}$ satisfy the criterion, p_i is migrated in every subsequent iteration at phase transition point $j-1$. For each page p_i that migrates at a phase 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 at 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 sensitive to the overhead of page migration. In the record-replay mechanism, page migrations must be performed on the critical path. Let $T_{nom,j}$ be the execution time of a phase j without page migration before the transition to the phase and $T_{m,j}$ the execution of the same phase with the record-replay mechanism enabled. Let $O_{m,j}$ be the overhead of 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 with page migration. 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 4 provides an example of using the record-replay mechanism in conjunction with the mechanism described in Section 4.2 in NAS BT. BT has a phase change at `z_solve`, due to the initial alignment of arrays in memory, which is performed to improve access 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`() in subsequent iterations. The function `upmlib_undo`() performs the replayed page migrations in the opposite direction.

5. Experimental results

We repeated the experiments presented in Section 3, after instrumenting the NAS benchmarks to use the page migration mechanisms of our runtime system. This section analyzes the results.

5.1. Performance of the iterative mechanism

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 existentially optimal initial data distribution scheme. Figures 5 and 6 repeat the results of Figs 1 and 2 and in addition, illustrate the performance of the iterative page migration mechanism of our runtime system with four different page placement schemes (labeled `ft-upmlib`, `rr-upmlib`, `rand-upmlib` and `wc-upmlib`).

```
...
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
...
call upmlib_end()
```

Fig. 4. Using the record-replay mechanism for data redistribution in NAS BT.

A first observation is that when first-touch is used, in all cases expect CG, user-level page migration provides sizeable reductions in execution time. The performance improvements range from 4% to 30% and average 9% on 16 processors and 17% on 32 processors. For the purposes of this paper, we consider this result as a second-order effect, attributed to the suboptimal placement of several pages by first-touch. We note however that this is probably the first experiment on a real system that shows meaningful performance improvements from the use of dynamic page migration.

The outcome of interest from the results in Figs 5 and 6 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, this slowdown is on average 3.5% for round-robin, 4.5% for random and 12% for worst-case page placement. Compared to the experiments presented in Section 3, the average slowdown of non-optimal page placement schemes is reduced by at least a factor of two and in some cases by as much as a factor of 8. There are also cases (e.g., SP and MG on 32 processors) in which the worst-case page placement combined with our page migration engine performs considerably better than first-touch.

Table 3 provides some additional statistics 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 on 16 processors.[7] 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 memory performance as an iterative parallel computation evolves in time.

The fifth, sixth and seventh column of Table 3 show the fraction of page migrations performed by our page migration engine in the first iteration of the parallel computation. In three out of five cases (CG, FT and MG), all page migrations are performed in the first iteration. In the case of BT and SP, some page-level false sharing forces page migrations in the second and third iterations. However, at least 78% of the migrations are performed in the first iteration. This result verifies that the page migration activity is concentrated at the begin-

[7] 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.

Table 3
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

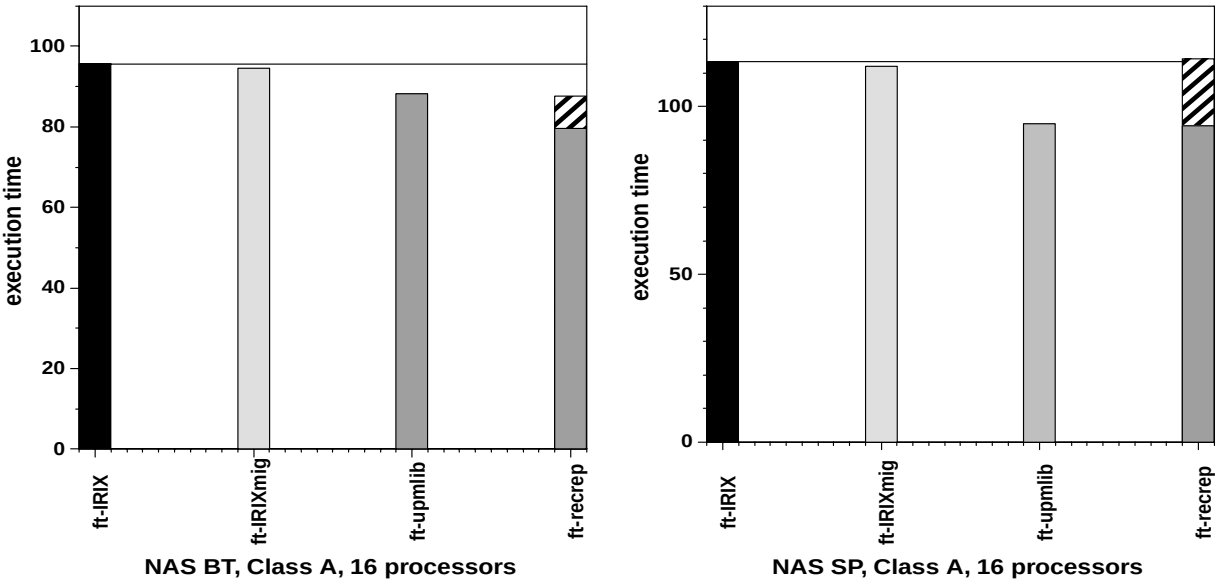

Fig. 7. Performance of the record-replay mechanism in NAS BT and SP. Execution times are on 16 idle processors of an Origin2000.

ning of execution and the associated cost is amortized well over the execution lifetime.

The conclusion from the results is that no matter what the initial placement of data is, our page migration engine achieves practically the same level of performance, which matches that of the theoretically best data placement scheme. At least for the class of strictly iterative parallel computations, our page migration engine makes OpenMP programs immune to the initial data placement scheme, or equivalently, relieves the programmer from the task of manual data distribution.

5.2. Performance of the record-replay mechanism

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 in `z_solve` as shown in Fig. 4.

Figure 7 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`. The striped part of the `ft-recrep` bar shows the non-overlapped overhead of 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 and 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 outweighs this speedup. When looking at the charts, one

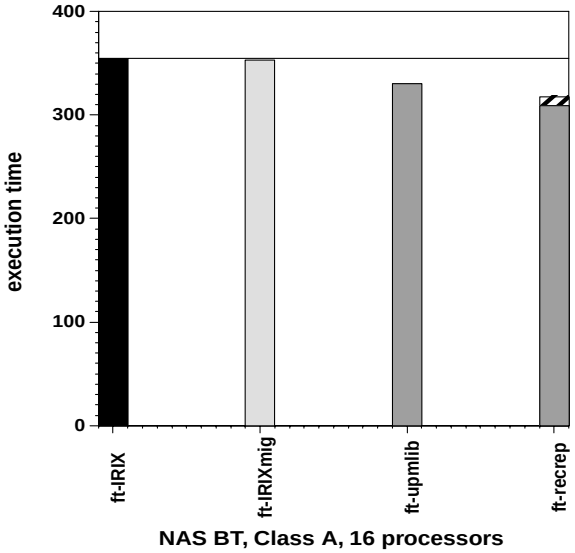

Fig. 8. Performance of the record-replay mechanism in the synthetic experiment with NAS BT. Execution times are on 16 idle processors of an Origin2000.

should bear in mind that the architectural characteristics of the Origin2000 bias the results. More specifically, the low remote-to-local memory access latency ratio and the high overhead of page migration due to the maintenance of TLB coherence by the operating system limit the gains from reducing the rate of remote memory accesses.

In order to overcome the aforementioned implication, we attempted to synthetically scale BT and increase the amount of computation performed in the benchmark. 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 benchmark as follows: we enclosed each function that comprises the main body of the parallel computation in a sequential loop with 4 iterations. In this way, we were able to increase 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 shown in Fig. 5.2 verify our intuition. The overhead of page migration accounts only for a very small fraction of execution time and the reduction of remote memory accesses shows up. In this experiment, the record-replay mechanism provides a measurable improvement of 5% over the version of the benchmark that uses page migration only for data distribution.

6. Related work

The idea of dynamic page migration has been developed since the appearance of the first commercial NUMA architectures more than a decade ago. Aside from several theoretical foundations on the algorithmic side of page migration [5,6], mechanisms for dynamic page migration in the operating system have been implemented on systems like the BBN Butterfly Plus and the IBM RP3 [7,15]. These systems had no hardware support for cache coherence and the cost of shared memory accesses was determined solely by the location of pages. Different schemes were investigated, such as migrating a page on every remote write, 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 was a reasonable choice, because the relative cost of page migrations was not so high compared to the cost of memory accesses. The effectiveness of these dynamic page migration mechanisms varied considerably and was affected significantly by architectural implications.

With the appearance of cache coherent NUMA multiprocessors, dynamic page migration became a harder problem. On the ccNUMA architecture, accesses to shared data are filtered from the caches and the memory performance of parallel programs depends heavily on the cache hit ratio. In the first detailed study of the related issues, Verghese et al. [34] 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 insufficient, because it does not reflect the frequency of accesses to pages. The same work proposed a complete kernel-level implementation of a dynamic page migration engine and evaluated it using accurate machine-level simulation. 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 of a multiprogrammed ccNUMA system running sequential jobs. The page migration engine of the Origin2000 is largely based on this work, although it has not been able to achieve the same level of performance improvements [18,27]. It is important to note that previous work investigated only the potential of dynamic page migration as a locality optimizer.

The context in which page migration is applied in our work is entirely different.

This paper is among the first to conduct a comprehensive evaluation of static page placement schemes on an actual ccNUMA multiprocessor. Such schemes were investigated via simulation in [3,25]. The study of Marchetti et al. [25] identified first-touch as an effective solution for simple parallel codes. Bhuyan et al. [3] have recently explored the impact of several page placement schemes in conjunction with alternative interconnection network switch designs, on the performance of parallel programs running 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 Bhuyan et al. agree with ours in the sense that non-optimal page placement schemes perform 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 [16,18]. However, these studies focused on analyzing the locality and load balancing characteristics of the specific nodes.

The idea of recording shared-memory accesses and use the recorded information to implement on-the-fly optimizations was exploited in the tapes mechanism [20]. This mechanism is designed for software distributed shared-memory systems, in which all accesses to shared memory are handled in software. 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 case 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 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, although some comparative results can be inferred from the performance of an existing HPF implementation of the NAS benchmarks on the Origin2000 [12]. HPF is very expressive with respect to data distribution and providing a one-to-one correspondence between HPF functionality 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 NUMA multiprocessors can be implemented with dynamic page migration mechanisms.

7. Conclusion

This paper raised a dilemma of whether manual data distribution 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 NUMA multiprocessors is aggressively optimized to reduce the remote-to-local memory access latency ratio to a point where any reasonably balanced automatic page placement scheme is expected to perform within a small fraction off the optimum. This trend is expected to persist in next-generation architectures, since all the related research efforts attack the problem of reducing the number of remote memory accesses. 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 poorly placed data at runtime. The synergy of architectural factors and advances in system software enables plain shared-memory programming models like OpenMP to maintain a competitive position against message-passing, by preserving 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] BBN Advanced Computers, Inside the Butterfly Plus, Version 1.0, October 1987.

[2] J. Berthou and E. Fayolle, Defining the Best Parallelization Strategy for a Diphasic Compressible Fluid Mechanics Code, in: *Proc. of the Second European Workshop on OpenMP* (*EWOMP'2000*), Edinburgh, Scotland, September 2000.

[3] 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*, **11**(3) (March 2000), 230–246.

[4] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. Nelson and C. Offner, Extending OpenMP for NUMA Machines. in: *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference* (*SC'2000*), Dallas, Texas, November 2000.

[5] D. Black, A. Gupta and W. Weber, Competitive Management of Distributed Shared Memory. in: *Proc. of the 34th IEEE Computer Society International Conference* (*COMPCON'89*), San Francisco, California, February 1989, pp. 184–191.

[6] 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, November 1989.

[7] W. Bolosky, R. Fitzgerald and M. Scott, Simple but Effective Techniques for NUMA Memory Management. in: *Proc. of the 12th ACM Symposium on Operating System Principles* (*SOSP'89*), Litchfield Park, Arizona, December 1989, pp. 19–31.

[8] R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedelijkovic and J. Anderson, Data Distribution Support on Distributed Shared Memory Multiprocessors. in: *Proc. of the 1997 ACM Conference on Programming Languages Design and Implementation* (*PLDI'97*), Las Vegas, Nevada, June 1997, pp. 334–345.

[9] R. Chandra, S. Devine, A. Gupta and M. Rosenblum, Scheduling and Page Migration for Multiprocessor Compute Servers. in: *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS-VI*), San Jose, California, October 1994, pp. 12–24.

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

[11] B. Falsafi and D. Wood, Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. in: *Proc. of the 24th International Symposium on Computer Architecture* (*ISCA'97*), Denver, Colorado, June 1997, pp. 229–240.

[12] M. Frumkin, H. Jin and J. Yan, Implementation of NAS Parallel Benchmarks in High Performance FORTRAN, Technical Report NAS-98-009, NASA Ames Research Center, September 1998.

[13] W. Gropp, A User's View of OpenMP: The Good, The Bad and the Ugly, in: *Workshop on OpenMP Applications and Tools* (*WOMPAT'2000*), San Diego, California, July 2000.

[14] E. Hagersten and M. Koster, WildFire: A Scalable Path for SMPs, in: *Proc. of the 5th International Symposium on High Performance Computer Architecture* (*HPCA-5*), Orlando, Florida, January 1999, pp. 171–181.

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

[16] C. Holt, J. P. Singh and J. Hennessy, Application and Architectural Bottlenecks in Large-Scale Distributed Shared Memory Machines, in: *Proc. of the 23rd International Symposium on Computer Architecture* (*ISCA'96*), Philadelphia, Pennsylvania, June 1996, pp. 134–145.

[17] Y. Hu, H. Lu, A. Cox and W. Zwaenepoel, OpenMP on Networks of SMPs, in: *Proc.of the 13th International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing* (*IPPS/SPDP'99*), San Juan, Puerto Rico, April 1999, pp. 302–310.

[18] D. Jiang and J.P. Singh, Scaling Application Performance on a Cache-Coherent Multiprocessor, in: *Proc. of the 26th International Symposium on Computer Architecture* (*ISCA'99*), Atlanta, Georgia, May 1999, pp. 305–316.

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

[20] P. Keleher, Tapeworm: High Level Abstractions of Shared Accesses, in: *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (*OSDI'99*), New Orleans, Louisiana, February 1999, pp. 201–214.

[21] D. Kuck, OpenMP: Past and Futures, in: *Proc. of the Workshop on OpenMP Applications and Tools* (*WOMPAT'2000*), San Diego, California, July 2000.

[22] J. Laudon and D. Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server, in: *Proceedings of the 24th International Symposium on Computer Architecture* (*ISCA'97*), Denver, Colorado, June 1997, pp. 241–251.

[23] J. Levesque, The Future of OpenMP on IBM SMP Systems, in: *Proc. of the First European Workshop on OpenMP* (*EWOMP'99*), Lund, Sweden, October 1999, pp. 5–6.

[24] H. Lu, Y. Hu and W. Zwaenepoel, OpenMP on Networks of Workstations. in: *Proc. of the IEEE/ACM Supercomputing'98: High Performance Networking and Computing Conference* (*SC'98*), Orlando, Florida, November 1998.

[25] M. Marchetti, L. Kontothanassis, R. Bianchini and M. Scott, Using Simple Page Placement Schemes to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems, in: *Proc. of the 9th IEEE International Parallel Processing Symposium* (*IPPS'95*), Santa Barbara, California, April 1995, pp. 380–385.

[26] A. Moga and M. Dubois, The Effectiveness of SRAM Network Caches in Clustered DSMs, in: *Proc. of the 4th International Symposium on High Performance Computer Architecture* (*HPCA-4*), Las Vegas, Nevada, January 1998, pp. 103–112.

[27] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta and E. Ayguadé, A Case for User-Level Dynamic Page Migration, in: *Proc. of the 14th ACM International Conference on Supercomputing* (*ICS'2000*), Santa Fe, New Mexico, May 2000, pp. 119–130.

[28] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta and E. Ayguadé. UPMlib: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared-Memory Multiprocessors, in: *Proc. of the 5th ACM Workshop on Languages, Compilers and Runtime Systems for Scalable Computers* (*LCR'2000*), *LNCS Vol. 1915*, Rochester, New York, May 2000, pp. 85–99.

[29] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta and E. Ayguadé, User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors, in: *Proc. of the 2000 International Conference on Parallel Processing* (*ICPP'2000*), Toronto, Canada, August 2000, pp. 95–103.

[30] OpenMP Architecture Review Board, OpenMP Fortran Application Programming Interface, Version 1.2, http://www.openmp.org, November 2000.

[31] M. Resch and B. Sander, A Comparison of OpenMP and MPI for the Parallel CFD Case, in: *Proc. of the First European Workshop on OpenMP*, Lund, Sweden, October 1999.

[32] Silicon Graphics Inc., IRIX 6.5 Man Pages, http://techpubs.sgi.com, November 1999.
[33] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI: The Complete Reference*, MIT Press, Cambridge, Massachusetts, 1996.
[34] B. Verghese, S. Devine, A. Gupta and M. Rosenblum, Operating System Support for Improving Data Locality on CC-NUMA Compute Servers, in: *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS-VII*), Cambridge, Massachusetts, October 1996, pp. 279–289.
[35] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, California, 1996.

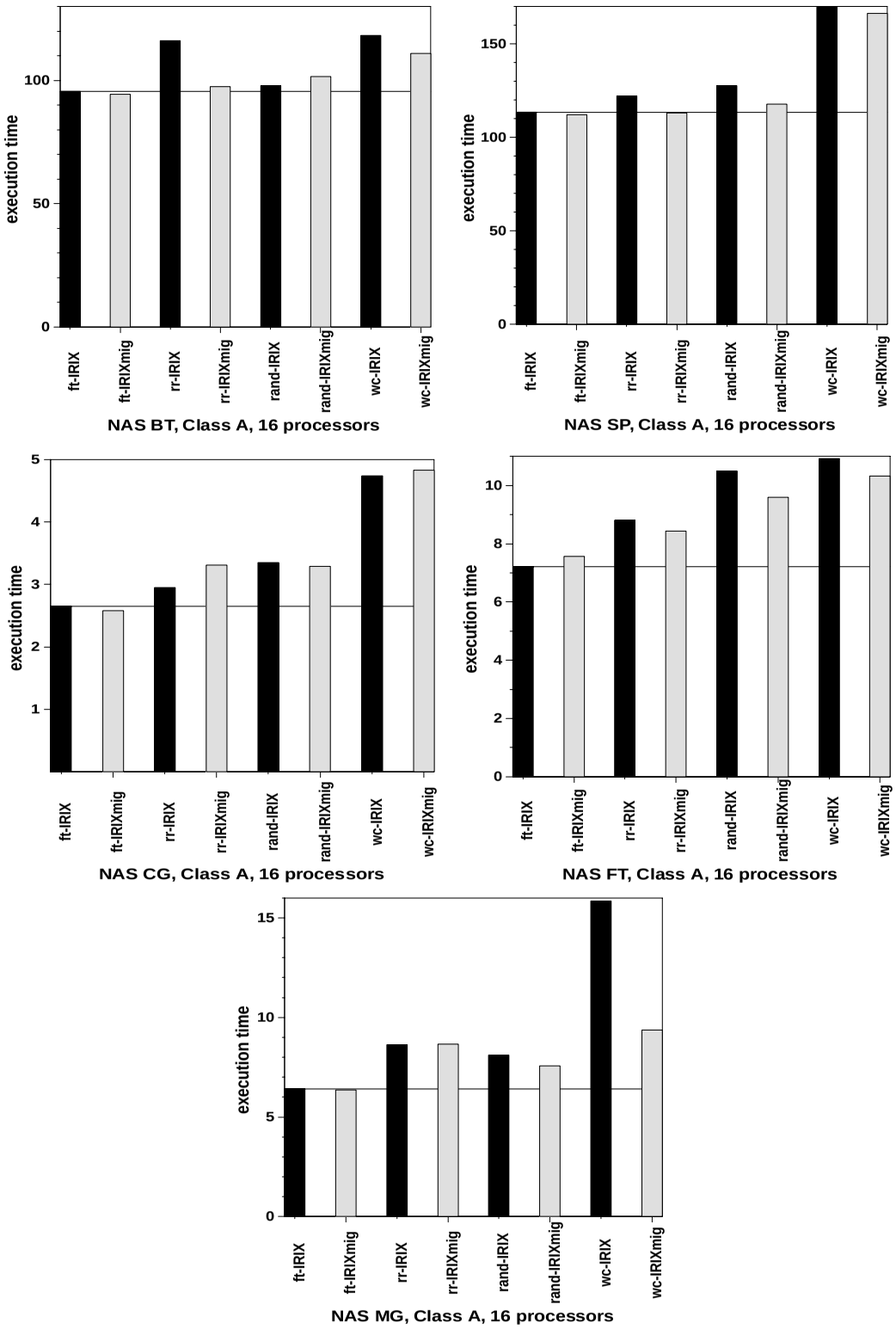

Fig. 1. Impact of page placement on the performance of the OpenMP implementations of the NAS benchmarks, executed on 16 idle processors of an Origin2000.

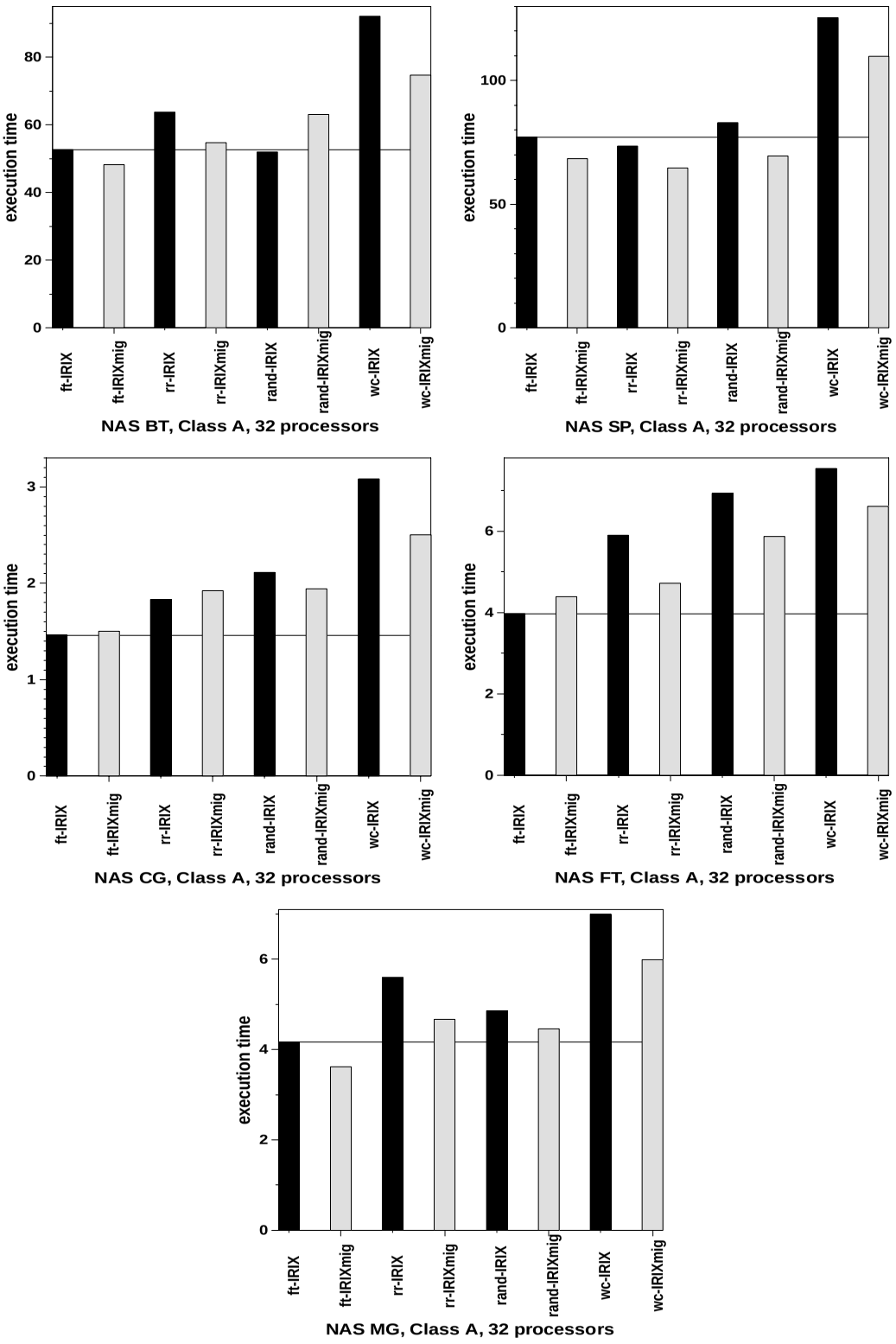

Fig. 2. Impact of page placement on the performance of the OpenMP implementations of the NAS benchmarks, executed on 32 idle processors of an Origin2000.

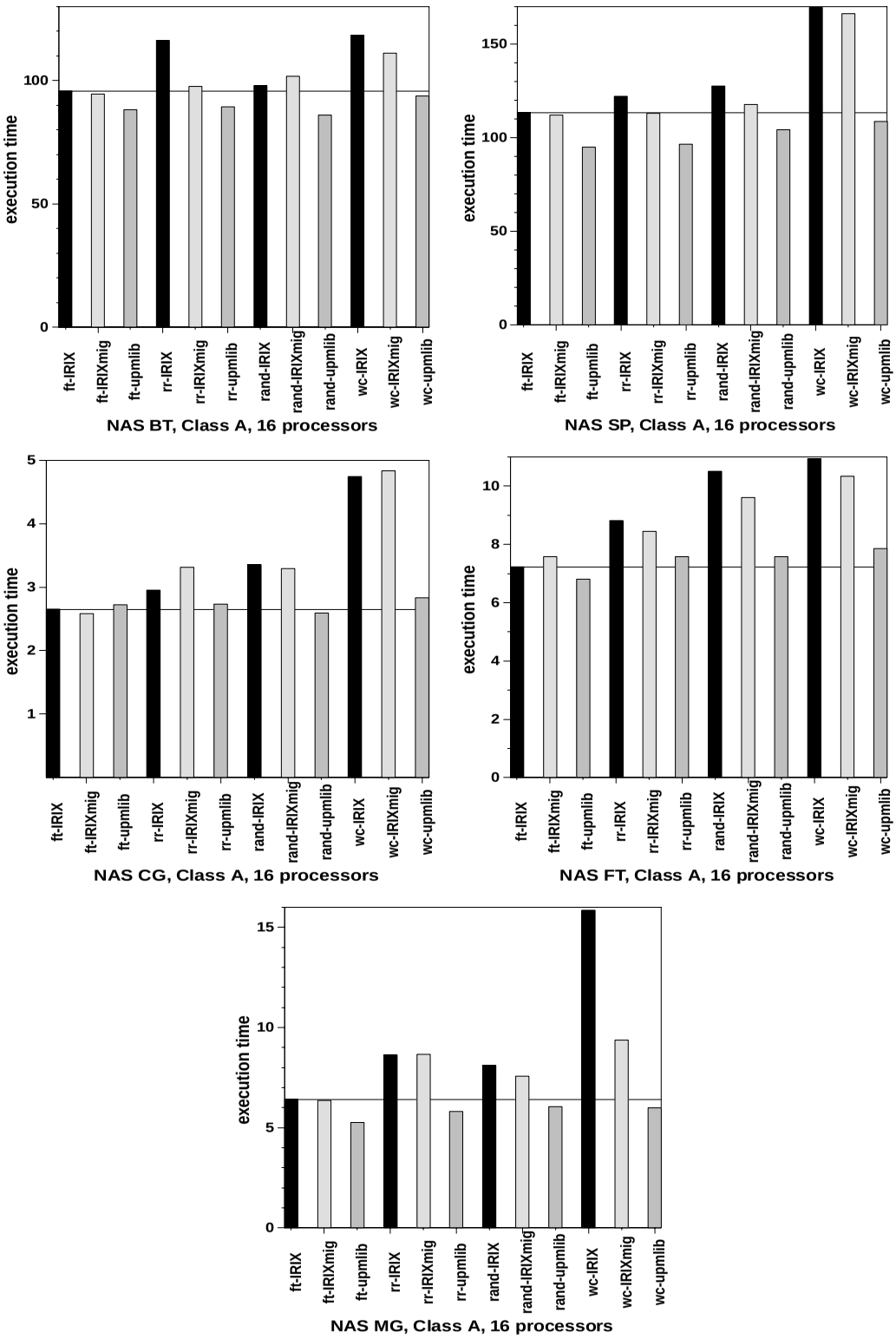

Fig. 5. Performance of our page migration runtime system with different page placement schemes in the NAS benchmarks, executed on 16 idle processors of an Origin2000.

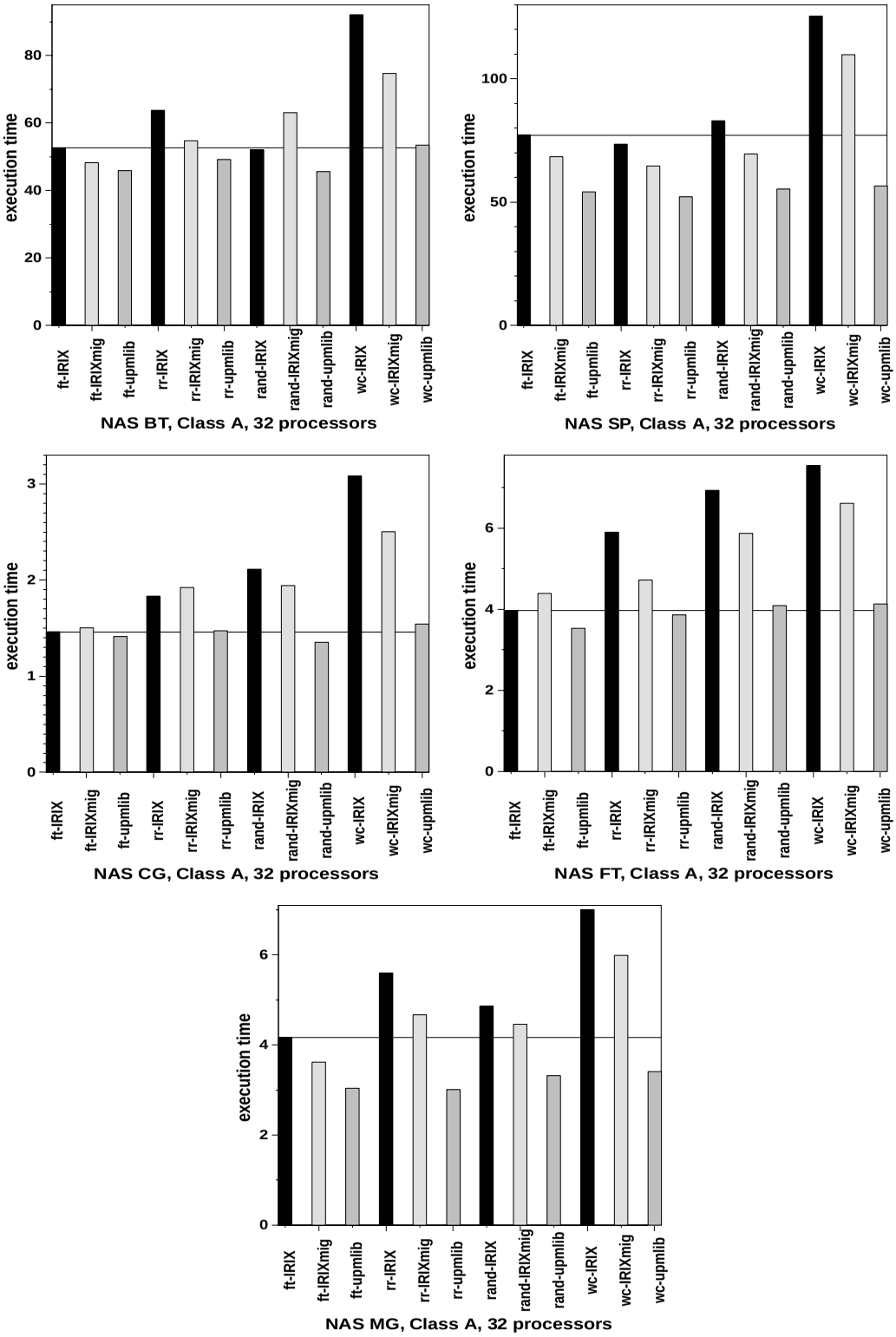

Fig. 6. Performance of our page migration runtime system with different page placement schemes in the NAS benchmarks, executed on 32 idle processors of an Origin2000.