Exploiting Operating System Support for Dynamic Page Placement on a NUMA Shared Memory Multiprocessor*

Richard P. LaRowe Jr.
James T. Wilkes
Carla Schlatter Ellis

Department of Computer Science
Duke University
Durham, NC 27706

Abstract

Shared memory multiprocessors are attractive because they are programmed in a manner similar to uniprocessors. The UMA class of shared memory multiprocessors is the most attractive, from the programmer's point of view, since the programmer need not be concerned with the placement of code and data in the physical memory hierarchy. Scalable shared memory multiprocessors, on the other hand, tend to present at least some degree of non-uniformity of memory access to the programmer, making the NUMA class an important one to consider. In this paper, we investigate the role that DUnX, an operating system supporting dynamic page placement on a BBN GP1000, might play in simplifying the memory model presented to the applications programmer. We consider a case study of psolu, a real scientific application originally targeted for a NUMA architecture. We find that dynamic page placement can dramatically improve the performance of a simpler implementation of psolu targeted for an UMA memory architecture. The addition of a phase change hint to the UMA version of psolu enables the operating system to further improve performance, approaching that of the hand-tuned NUMA implementation.

*This research was supported in part by NSF grants CCR-8721781 and CCR-8821809, ONR contracts N00014-85-K-0487 and N00014-89-J-1644, and AT&T contract #248953.

1 Introduction

Shared memory multiprocessors are among the most popular parallel architectures because the programming interface to these machines bears close resemblance to that of typical uniprocessors. The addition of a few primitives for synchronization and process control to standard uniprocessor programming environments is generally sufficient for implementing applications on shared memory machines. For small-scale bus-based multiprocessors, which support an UMA (Uniform Memory Access time) memory model, these environments have been proved successful.

As shared memory multiprocessors scale to large numbers of processors, non-uniformity of memory access becomes an almost inevitable feature. Distributing the memory of a large-scale multiprocessor among the processors so that each location can be considered close to some processor(s) while being distant from others has significant price/performance advantages. Examples of current NUMA (Non-Uniform Memory Access time) machines include the BBN Butterfly family [2, 3, 4], Cedar at the University of Illinois [13, 29], the IBM RP3 [24], and Cm* and PLUS at Carnegie-Mellon University [5, 17, 26].

An important implication of the NUMA architectural design decision is that the placement and movement of code and data become crucial to performance. In order to achieve good performance, the programmer is typically forced to explicitly manage the locality issues in the execution of the program. This involves trying to place data initially in the memory of the node where it is expected to be most used, scattering data among memories in order to reduce contention, making copies of read-only or slowly changing data to encourage local access, worrying about the consistency of any copies made, and moving data when reference patterns are known to change during execution. Unfortunately, this effort represents a significant burden placed on the programmer, and severely limits the attractiveness of the programming interface provided by most NUMA shared memory multiprocessors.

Ideally, we wish to provide an environment resembling the UMA programming model supported by small-scale shared memory multiprocessors on the large-scale NUMA architectures. One approach to achieving this goal is to design a memory hierarchy based on transparent caches with some hardware-supported consistency protocol. There are several recent proposals for scalable shared memory architectures based upon this approach [11, 14, 22, 28]. However, the potential performance gains from making concessions to the actual non-uniformity of the memory architecture become more evident as these systems grow. For example, the benefits of taking advantage of locality in Encore's cluster architecture (now known as the Gigamax) are acknowledged in [28]. Lessons learned in the explicitly NUMA environments are likely to become increasingly relevant in all scalable shared memory designs.

As memory architectures become more complex and the non-uniformity becomes less well hidden, system software must assume a larger role in providing memory management support for the programmer. The NUMAtic project at Duke has been addressing this problem by investigating the role of the operating system [15, 16, 18, 19, 20]. This is an area of active research by other groups as well [5, 6, 7, 8, 9, 12, 21, 23, 25].

In this paper, we describe a recent experience of designing an application to use the memory management capabilities provided in DUnX (Duke University nX - pronounced "ducks"), our locally-developed kernel (based on BBN's nX) for the BBN GP1000. DUnX supports dynamic page placement in an effort to provide a more UMA-like memory model on the GP1000. The application we consider is called `psolu` [27]. It uses block chaotic relaxation to solve the linear system $Ax = b$. The questions we attempt to address include the following: How can an application program best exploit the dynamic placement capabilities of DUnX? To what extent can the user adopt an UMA programming model without sacrificing performance significantly? Are there *minor* concessions to the NUMA nature of the architecture that can be provided in the OS interface to allow the programmer to assist the memory management decision making?

It is important to emphasize that this paper reports on a single case study of using the dynamic page placement support in DUnX. It is not intended to serve as justification for dynamic page placement in general or for DUnX-specific features in particular. These issues have already been discussed by the authors [19] and others (e.g., [8, 12]). What distinguishes this paper from others in the literature is that it provides an in-depth account of how dynamic page placement can be used to improve the performance of a real UMA application, and compares it to the performance obtained through hand-tuning of a NUMA version of the same application. The paper also considers the use of source-level hints to the operating system, which can be exploited for further performance improvement.

In the next section of this paper we discuss dynamic page placement in general, and then one of the DUnX policies in particular. Section 3 describes three implementations of `psolu`. In section 4 we evaluate performance of our three `psolu` implementations under different page placement strategies. Section 5 considers the relationship of our work to the work of others. Finally, we conclude with a summary of our results and a discussion of the virtues and limitations of dynamic page placement towards simplifying the programmer interface to NUMA shared memory multiprocessors.

2 Dynamic Page Placement and DUnX

Traditional page placement policies are *static*, in the sense that once a decision is made about the placement of a page, that decision remains in effect until the page is selected for replacement by the replacement policy. In a uniprocessor or UMA multiprocessor environment, this is an acceptable way to handle placement since there is no reason to change the placement of a page (all physical memory frames are created equal).

In a NUMA shared memory multiprocessor environment, however, changing reference patterns often affect the desired placement of pages. A page initially used by one processor may subsequently become more frequently used by another, changing the desired placement of that page from a frame on the first node to a frame on the second. Thus, it becomes desirable to use a *dynamic* page placement policy that periodically reevaluates earlier page placement decisions and makes the adjustments necessary to improve performance.

NUMA multiprocessor page placement policies also differ from uniprocessor and UMA multiprocessor policies in that it is often desirable to maintain multiple physical copies of a single virtual page. While allowing multiple copies of a page necessitates some page coherence mechanism, it can greatly improve performance in situations where several processors are sharing pages in a read-only or read-mostly fashion. This is similar in intent to the hardware caching of shared data commonly found in UMA multiprocessors, with the exception that in the NUMA environment, it is possible for one processor to directly reference data in another node, thus avoiding the creation of an additional copy. This is a feature that can be exploited for further performance improvements.

The DUnX kernel was developed to serve as a platform for studying memory management policies for NUMA shared memory multiprocessors. It supports dynamic, user-level policy selection on a per-cluster basis (a cluster, in BBN terminology, is a dedicated subset of processor nodes). We have studied and compared a wide assortment of different page placement strategies (both static and dynamic, single-copy and multiple-copy) using measurements of application performance under DUnX [19].

In this paper, we consider one static single-copy policy and one dynamic multiple-copy policy. The purpose is to look at how dynamic page placement can improve upon the programming model currently available on most NUMA shared memory multiprocessors.

The static policy places each virtual page in a frame on the processor that first references that page or in a frame on the processor explicitly specified by the application programmer when the virtual memory is allocated. Other processors that wish to share access to the page create mappings to the same physical copy (with the exception of text pages, which are always replicated to each node using them), and the placement of a page does not change unless it is selected for replacement by the replacement policy and later paged in again. This policy is identical to that currently provided in BBN's nX and Mach-1000 operating systems.

The dynamic policy that we consider uses the same initial placement as the static policy, but arranges to periodically reevaluate earlier decisions and allows multiple physical copies of a single virtual page. The policy is comprised of two parts; the fault handler subpolicy and the page scanner subpolicy.

When a fault is taken on a page that is resident in some remote frame, the fault handler subpolicy must decide whether to use the remote copy, migrate the remote copy to a local frame (invalidating the source copy), or replicate the page to a local frame. If the page has been recently migrated or invalidated due to a coherency fault, or if the page has been marked *frozen*, the remote copy of the page is used to service the fault and it is frozen. Otherwise, the page is either migrated or replicated. If a write reference triggered the fault or if the page has been recently modified the page is migrated. On a read reference to a page not recently modified, the page is replicated. Note that a write reference to a page that has been replicated always results in the invalidation of all remote copies of that page, so that data coherence is assured.

Each processor has a page scanner daemon that periodically runs on that node to help support memory management. One of the scanners' many jobs is to trigger the reevaluation of earlier page placement decisions either by invalidating all virtual to physical mappings to a page or by invalidating all remote mappings to a page. The resulting page faults provide opportunities for the fault handler subpolicy to change page placements.

The scanners are primarily interested in frozen pages, as they are the ones that processors have been forced to reference remotely. Whenever a scanner detects that a frozen page has been referenced by remote processors as often as local processors for some number of scanner runs, the page is *defrosted.* If the page has been recently modified, the page is defrosted by invalidating all virtual to physical mappings for it and clearing the frozen flag. Otherwise, it is defrosted by simply invalidating remote mappings to the page and clearing the frozen flag. The two different defrost mechanisms correspond to whether the fault handler subpolicy is likely to migrate or replicate the page on subsequent faults.

This dynamic multiple-copy page placement policy is highly tunable, controlled by seven parameters that are user-level adjustable at run-time. The parameters control such things as the frequency of page scanner daemon runs and the definitions of "recently migrated or invalidated" and "recently modified". The ability to tune DUnX policies to an application is coupled with facilities, built into the kernel, for monitoring virtual memory activity and extracting information during program execution (e.g., shared memory access patterns and faulting behavior).

One final DUnX feature of importance to this paper is the ability to respond to a *phase change hint* from an application program. A programmer may insert a phase change hint (which takes the form of a system call directed at a particular cluster) in an application whenever shared reference patterns are known to change. In response to such a hint, the page scanners running in that cluster invalidate all remote virtual mappings to their physical frames. In addition, all frozen pages are defrosted. This essentially "resets" the page placement policy, so that the desired placement of all shared pages can easily be reevaluated as the new program phase is entered. As we shall see later, this feature can easily be exploited to improve the performance of at least one application.

Due to space limitations, this is an admittedly high-level description of the dynamic policy considered in this paper. The reader is referred to [19] for a more detailed description of the policies and mechanisms provided in the DUnX kernel.

3 The psolu Application

The psolu application solves the sparse linear system $Ax = b$ using block chaotic relaxation [27]. Block chaotic relaxation is a parallel iterative method developed to overcome the synchronization problems encountered when trying to parallelize traditional iterative methods [1, 10].

The "block" in block chaotic relaxation comes from the fact that the input matrix A is partitioned into blocks, which form the unit of parallelism. A static load balancing scheme uses the number of operations per block per iteration to estimate the processing time required for each block. Blocks are assigned to the processors so that the estimated processing time is minimized. The method is called "chaotic" since the processors taking part in the computation do not synchronize between iterations. Each processor iterates on the blocks for which it is responsible until the computation is complete. A special convergence-checking process continuously computes $||Ax - b||$ and compares it to zero. When the solution is sufficiently accurate, computation ceases.

Three versions of psolu have been implemented. The heart of the computation for all three versions is computing iterates. To compute the iterates for a particular block, the processor responsible for that block must access the block row (think of the input matrix A as being comprised of rows of blocks), the diagonal block factors, the x and b vectors, and the shared iteration control data. Once the iteration phase of the computation begins, most of the shared data structures are referenced in a read-only fashion. Only the x vector is regularly modified throughout the computation. The primary difference between the three psolu implementations is the allocation of these shared data.

The UMA version was written ignoring the difference in remote and local memory access times, so data space is allocated without regard to physical location or pagination. This is the style in which the application would likely have been written had the target machine been an UMA shared memory multiprocessor, such as the bus-based machines manufactured by Encore and Sequent. To provide easy access to the data needed for computation, the UMA version uses the blocking assignment determined by the load balancing algorithm to create lists, one per processor, of iteration data. Each list contains one entry per block assigned to that processor with each entry containing pointers to the block row, pointers to the factors, and iteration control data. Access to x and b is through sharing original copies of these contiguous arrays.

As we shall see later, the UMA version of psolu performs quite poorly when run without dynamic page placement. Since no such support was available to us when psolu was first implemented, hand optimization of the application code was essential to obtaining reasonable performance. We refer to this hand-tuned version of psolu as the NUMA version. The NUMA version creates the same lists of iteration data as the UMA version, but the memory for each list is allocated locally on the processor using the list. Also, instead of using the original copy of the block rows and factors, the NUMA version makes another copy of A and the factors, with block rows and corresponding factors manually placed in the memory of the node responsible for processing that row. A second copy of x is also made; this is a distributed copy made up of a list of pointers, one per subvector, with each subvector allocated in the memory of the processor responsible for computing that subvector. Since each processor must access the entire x vector, the array of pointers to the subvectors is manually replicated across the processors. Manual replication of other read-only data structures, such as the b vector, is also employed to further improve performance.

The hand-tuning of psolu for a particular NUMA architecture has been a nontrivial task, as it represents a significant portion of the total development time required for the application. The extra programming effort required is not the only drawback to the hand-tuning approach, however. As a result of architecture dependent code, the NUMA implementation is more complicated and less portable than the UMA implementation. If the issue of performance could be ignored, the UMA implementation is certainly more desirable.

The hand-tuned NUMA psolu and the implementation based upon a pure UMA memory model are two endpoints on a continuum of possible levels of programmer involvement in NUMA memory management. There are many potentially attractive intermediate points requiring different amounts of investment on the part of the programmer, including the tuning of policy parameters, providing hints to the operating system about the program's memory access patterns, and paying attention to the page layout of data during memory allocation. We have considered only the OS interface, ignoring possible contributions by a sophisticated NUMA-izing compiler. Monitoring preliminary executions of the UMA implementation and tuning the parameters defining the dynamic policy were used to determine the best policy choice for our experiments. The next obvious step beyond policy tuning for the UMA program was to try to identify minimal code changes that could productively assist the OS-level memory management. This led to the third psolu implementation, which we call the UMA+ version. The UMA+ version differs from the UMA version by a single line of source code. Monitoring of the virtual memory behavior of the UMA psolu indicated that the dynamic policy was slow in responding to a particular phase transition. Thus, immediately prior to the start of the iteration phase of the computation, which follows initialization and factorization of the diagonal blocks of the matrix, the UMA+ psolu supplies a hint to DUnX that a phase transition is about to occur.

4 Experiments and Results

In order to evaluate the performance of our three psolu implementations under static and dynamic page placement policies, we solve the matrix equation generated by solving the Dirichlet problem, $\nabla^2 u = 0$ with u defined on the boundary, on a square grid using different numbers of processor nodes. The completion time results (measured in seconds) for four implementation/policy combinations (NUMA/static, UMA/static, UMA/dynamic, and UMA+/dynamic) are presented in figure 1.

It is readily apparent that the NUMA/static results are much better than the UMA/static results. At two processors, the smallest number of processors on which the applications can be run, the UMA/static completion time is nearly three times that of the NUMA/static completion time. With five processors, the difference is about the same, but after five processors, the UMA/static times grow steadily slower. Clearly, performance of the UMA **psolu** implementation is unacceptable under the static page placement policy.

The UMA/dynamic results are much more encouraging, however. While slower than the NUMA/static combination in all cases, the difference is not nearly as great as with the UMA/static combination. This suggests that it may be difficult to justify the extra programmer effort required to obtain the improved performance of the NUMA implementation. Performance of the UMA+/dynamic combination lies between that of the NUMA/static and UMA/dynamic results. Since the effort required to create the UMA+ version is substantially less than that for the NUMA version, the programmer time required to achieve this performance gain is more easily justified.

If we define the speedup of an implementation running with $n-1$ iterating processors (n processors total) to be the completion time of that same implementation running with one iterating processor divided by the completion time running the implementation with $n-1$ iterating processors, we can compare the implementations' performance qualitatively. In figure 2, we plot speedup versus the total number of processor nodes for each of our four implementation/policy combinations.

The speedup curves emphasize the qualitative performance difference of the four implementation/policy combinations. As we could see from the completion time results, the UMA/static combination is unacceptable after just five processors (where it does achieve a speedup of nearly three with four iterating processors). The other combinations, however, derive increased performance with additional processors through 33, though the speedup curves do appear to be leveling off in all three cases (dropping slightly in the UMA/dynamic case). Clearly, the NUMA/static combination is the most successful, with the UMA+/dynamic and UMA/dynamic combinations following in that order.

It is important to point out the reason for relatively low quantitative speedup values for even the NUMA/static combination. The primary reason for this less than linear speedup is the convergence behavior of the chaotic relaxation. As the number of iterating processors increases, the average number of iterations (per-processor) required to converge also increases. For example, the number of iterations required for the NUMA/static combination (results for the other combinations are similar) can be approximated by the line $\mathit{Iterations}(n) = 2713.44 + 74.977n$, which is a least-squares fit of the measured data points with a mean error of less than one and a half percent.

The increase in the number of iterations required as the number of processors is increased is related solely to the solution method being employed, and is unrelated to the memory management policies under which the implementations are run. Though a detailed explanation of the cause of this behavior is beyond the scope of this paper, it is helpful to consider the performance of two traditional methods, Gauss-Siedel and Jacobi iteration, for solving the Dirichlet problem. For this problem, Gauss-Siedel typically converges in about half the iterations as the Jacobi method. Though an exact correspondence cannot be made, the block chaotic relaxation method used by **psolu** more closely resembles the Gauss-Siedel method when run with fewer processors and behaves more like the Jacobi method when the number of iterating processors gets larger. This is primarily due to the increased probability, with an increased number of iterating processors, of using an old x value when computing the values for the next iteration.

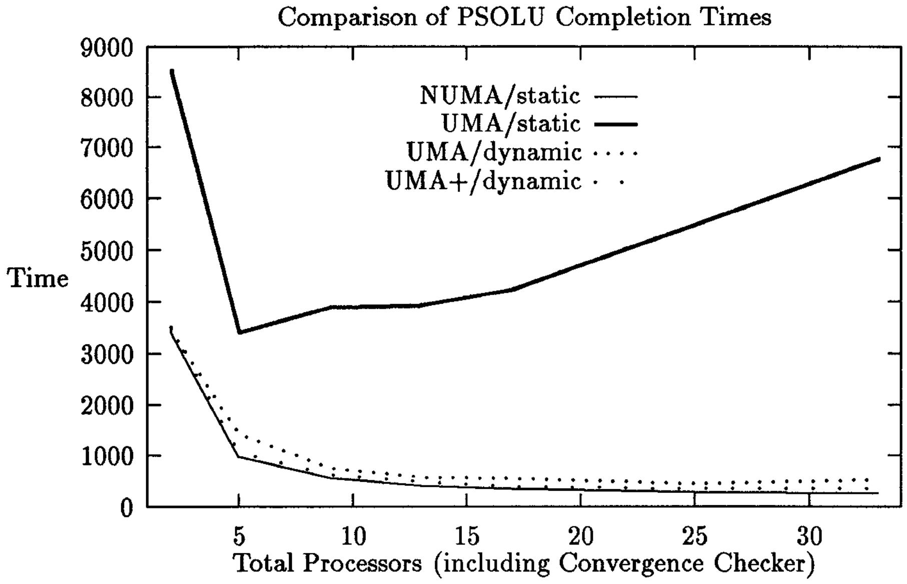

Figure 1: Completion Time versus Number of Processors

This is discussed more thoroughly in [27].

An important question concerns the performance difference between the UMA+/dynamic combination and the UMA/dynamic combination. Since only one line of source code distinguishes the two, such a significant performance difference is at first surprising. In table 1, completion times and the numbers of page faults, page migrations, page replications, and coherency faults (the total number of replicas invalidated is shown in parentheses) are given for the two combinations when run on 13 processor nodes.

The additional page faults experienced by the UMA+ implementation are a result of the DUnX response to the phase change hint, which triggers the reevaluation of all earlier page placement decisions to use remote frames. Both versions perform approximately the same number of page migrations, but the UMA+ version performs a fair number of additional page replications. The data do not show, however, whether the additional replications contribute significantly to the noted performance difference.

We believe that much of the performance gain of the UMA+ version over the UMA version is a result of desired page replications (primarily pages containing blocks of A, the diagonal block factors, b, and other assorted data referenced in a read-only fashion throughout the iteration phase of the computation) being performed by the UMA+ implementation sooner than with the UMA implementation. When monitoring the behavior of the UMA/dynamic combination, we see that throughout the iterating phase of the computation, pages are periodically defrosted and replicated. No coherency faults or migrations occur during this part of the computation, indicating that the pages being replicated were frozen prior to the beginning of the iteration phase. This is the observed behavior that prompted us to add the phase change hint to the UMA `psolu` to derive the UMA+ implementation.

When we monitor the UMA+/dynamic combination, the memory management behavior we see is somewhat different than with the UMA/dynamic case. As soon as the iteration phase begins, a far greater number of page replications occur. Soon after, however, nearly all dynamic page placement activity ceases. This indicates that the desired replications occur immediately after the phase change, so that the iterating processors are able to take advantage of the local references as soon as possible.

We suspect that the primary reason for the greater number of replications in the UMA+/dynamic case than in the UMA/dynamic case is related to the "refreezing" of some set of pages (probably containing the x vector,

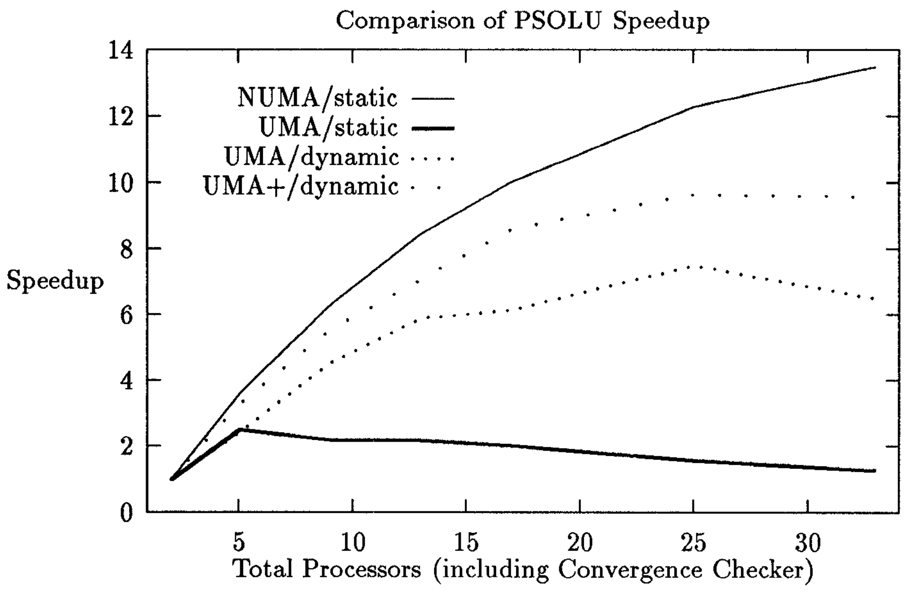

Figure 2: Speedup versus Number of Processors

Test	Time	Faults	Migrations	Replications	Coherency
UMA/dynamic	638.1*s*.	3145	69	605	69 (240)
UMA+/dynamic	497.9*s*.	4024	79	757	110 (362)

Table 1: Operation Counts for 13-Node UMA and UMA+ Dynamic Tests

which ideally should be frozen to prevent page bouncing from node to node) that are initially frozen prior to the start of the iteration phase of the computation. Since the UMA+ phase change hint causes all frozen pages to be defrosted, those pages may again be replicated to other nodes before a series of coherency faults (caused by regular updates to the x vector by the iterating processors) causes those pages to be frozen once more.

Another question of importance concerns the performance difference between the UMA+/dynamic combination and the hand-tuned NUMA/static combination. The overhead of running the dynamic policy is one important cause. The far fewer number of page faults that occur for the NUMA/static case (e.g. only 2255 faults in the 13 processor case versus 4024 for UMA+/dynamic), the lack of overhead for migrating and replicating pages, and the savings that result from not running the page scanners all contribute to the savings in overhead. Even attributing excessively high (unrealistic) costs for each of these factors (30 *ms.* per fault, 5% scanner overhead), they amount to only about 80% of the total performance difference, indicating that some other factor also plays a significant role.

The likely cause of some of the performance difference is the placement of pages containing the x vector. The NUMA **psolu** implementation breaks the x vector into subvectors corresponding to the blocking of the A matrix. Each subvector is manually placed on the processor responsible for the corresponding block of A, so that each element of x is local to the processor that references it most often. The UMA+ implementation, on the other hand, uses a single contiguous array to hold the x vector. Since several subvectors of x fit within a single page, and each of those subvectors may be the responsibility of a different processor, the best placement for each x page is unclear. In addition, the dynamic policy is faced with references to the entire x vector from the convergence checker process. Of course, it is impossible for the operating system to know, without being told, that there is little reason to place pages of x on the node running the convergence checker. Thus, at least some of the iterating processors are faced with a far greater number of remote x vector references under the UMA+/dynamic case than under the NUMA/static case.

We suspect that the performance of the UMA+ `psolu` implementation could be improved by ensuring that each x subvector is allocated in its own virtual page. Further improvements would likely be possible if the kernel could be told not to place pages containing x vector data in frames on the node running the convergence checker. We have not yet investigated these more extensive changes, however.

5 Relationship to Other Work

Although this kind of case study could have been performed in other OS environments supporting dynamic page placement (e.g., [7, 8, 12]), we are unaware of similar experiences based on an extensively studied NUMA program. The relationship of DUnX to those other systems can be explained in terms of the goals set forth at the start of the DUnX project. DUnX has been designed as a vehicle for comparing a wide range of memory management alternatives. The emphasis has been more on evaluation than on proposing particular policies. Each of the related projects has focussed on a relatively small set of original policy proposals. Black, Gupta, and Weber [7] introduced provably competitive page migration and replication policies that require hardware support not commonly available. Bolosky, Scott, and Fitzgerald [8] introduced a simple dynamic page placement policy based on allowing only a small number of invalidations before permanently fixing the placement of a page. A more dynamic policy based on the freezing and defrosting of pages was introduced by Cox and Fowler [12]. The flexibility built into DUnX allows us to specify at least approximations of those policies and, thus, to consider them in our experimental evaluations. A side-effect of our providing a range of policy options in a single system is that it permits the user to tune the policy to be used to a particular application.

6 Conclusions

Shared memory multiprocessors are attractive because they are programmed in a manner similar to uniprocessors. The UMA class of shared memory multiprocessors is the most attractive, from the programmer's point of view, since the programmer need not be concerned with the placement of code and data in the physical memory hierarchy. Scalable shared memory multiprocessors, on the other hand, tend to present at least some degree of non-uniformity of memory access to the programmer, making the NUMA class an important one to consider.

In this paper, we considered `psolu`, a real scientific application originally targeted for a NUMA architecture (a BBN Butterfly GP1000). A significant amount of the development time for the original `psolu` (the hand-tuned NUMA version) was spent dealing with the problem of managing the memory hierarchy efficiently. The UMA `psolu` is a much simpler implementation of the same application, but was found to perform quite poorly when run under a traditional static page placement policy. We found that dynamic page placement, however, was able to greatly improve the performance of that same program, with no expenditure of effort on the part of the application programmer.

The addition of a *phase change hint* to the UMA implementation source code proved to be quite successful in further improving performance. The operating system's dynamic page placement policy used the hint to better respond to the shared memory reference patterns that accompanied the phase change, significantly reducing the time spent readjusting the placement of shared pages.

Though neither of the UMA implementations of `psolu` achieved performance quite as good as the NUMA version, there was substantial improvement in performance over the UMA version when run under a static page placement policy. The difference in performance between the NUMA implementation/static policy combination and the UMA+/dynamic combination would have to carefully be weighed against the cost of the programmer effort required to achieve that performance improvement.

We have demonstrated one example of an application for which operating system support can improve the performance of UMA programs. We must caution the reader, however, that our experience has shown that there exist applications for which this has not yet been achieved [19]. The false sharing problem, in particular, poses problems for operating systems level solutions to the NUMA problem. We suspect that any complete solution to the NUMA problem will require compiler and/or user library support in addition to the support provided by the operating system kernel.

References

[1] G. M. Baudet. Asynchronous iterative methods for multiprocessors. *J. Assoc. Comput. Mach.*, 25:226–244, 1978.

[2] BBN. *Butterfly Parallel Processor Overview.* Cambridge, MA, June 1985.

[3] BBN. *Inside the Butterfly GP1000.* Cambridge, MA, October 1988.

[4] BBN. *Inside the BBN TC2000.* Cambridge, MA, February 1990.

[5] R. Bisiani and M Ravishankar. PLUS: A distributed shared-memory system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, June 1990.

[6] R. Bisiani and M Ravishankar. Programming the PLUS distributed-memory system. In *Proceedings of the 5th Annual Distributed Memory Computing Conference*, pages 115–124, April 1990.

[7] D. Black, A. Gupta, and W-D Weber. Competitive management of distributed shared memory. In *Spring COMPCON 89 Digest of Papers*, pages 184–190, 1989.

[8] W. Bolosky, M. Scott, and R. Fitzgerald. Simple but effective techniques for NUMA memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

[9] M. Brorsson. Performance impact of code and data placement on the IBM RP3. Technical Report Research Report RC 14651 (#65734), IBM Research Division, June 1989.

[10] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Its Applications*, 2:199–222, 1969.

[11] D. Cheriton, H. Goosen, and P. Boyle. Multi-level shared chaching techniques for scalability in VMP-MC. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 16–24, June 1989.

[12] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with Platinum. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 32–43, December 1989.

[13] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. Cedar—a large scale multiprocessor. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 524–529, August 1983.

[14] J.R. Goodman and P.J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, Honolulu, HI, 1988.

[15] M. Holliday. Page table management in local/remote architectures. In *ACM SIGARCH Int. Conf. on Supercomputing*, pages 1–8, July 1988.

[16] M. Holliday. Reference history, page size, and migration daemons in local/remote architectures. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, pages 104–112, April 1989.

[17] A. K. Jones, R. J. Chansler Jr., I. Durham, P Feiler, and K. Schwans. Software management of Cm* - a distributed multiprocessor. In *1977 National Computer Conference*, volume 46, pages 657–663, 1977.

[18] R. P. LaRowe Jr. and C. S. Ellis. Dynamic page placement in a NUMA multiprocessor virtual memory system. Technical Report CS-1989-21, Duke University, October 1989.

[19] R. P. LaRowe Jr. and C. S. Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. Technical Report CS-1990-10, Duke University, April 1990. Submitted.

[20] R. P. LaRowe Jr. and C. S. Ellis. Page placement policies for NUMA multiprocessors. *Journal of Parallel and Distributed Computing*, to appear.

[21] T. LeBlanc, M. Scott, and B. Marsh. Memory management for large-scale multiprocessors. Technical Report 311, Dept. of Computer Science, Univ. of Rochester, March 1989.

[22] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessey. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[23] R. McGrath and P. Emrath. Using memory in the Cedar system. Technical Report CSRD rpt. no. 655, Center for Supercomputing Research and Development, U of Illinois, June 1987. Also in Proceedings of Int'l Conf. on Supercomputing, Athens, Greece, June 1987.

[24] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[25] C. Scheurich and M. Dubois. Dynamic page migration in multiprocessors with distributed global memory. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 162–169, June 1988.

[26] R. J. Swan, S. H. Fuller, and D. P. Siewiorek. Cm* - a modular, multi-microprocessor. In *1977 National Computer Conference*, volume 46, pages 637–644, 1977.

[27] J. T. Wilkes. BLITPAK II: A parallel implementation of block iterative methods for sparse systems of linear algebraic equations. Master's thesis, Duke University, December 1989.

[28] A. W. Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987.

[29] Pen-Chung Yew. Architecture of the Cedar parallel supercomputer. Technical Report CSRD 609, Center for Supercomputing Research and Development, Univ. of Illinois, August 1986.